A Duality of Sorts

Ralf Hinze, José Pedro Magalhães, and Nicolas Wu*

Department of Computer Science, University of Oxford
{ralf.hinze,jose.pedro.magalhaes,nicolas.wu}@cs.ox.ac.uk

Abstract. Sorting algorithms are one of the key pedagogical foundations of computer science, and their properties have been studied heavily. Perhaps less well known, however, is the fact that many of the basic sorting algorithms exist as a pair, and that these pairs arise naturally out of the duality between folds and unfolds. In this paper, we make this duality explicit, by showing how to define common sorting algorithms as folds of unfolds, or, dually, as unfolds of folds. This duality is preserved even when considering optimised sorting algorithms that require more exotic variations of folds and unfolds, and intermediary data structures. While all this material arises naturally from a categorical modelling of these recursion schemes, we endeavour to keep this presentation accessible to those not versed in abstract nonsense.

1 Introduction

Sorting, described in great detail by Knuth (1998), is one of the most important and fundamental concepts in computer science. In one form or another, sorting appears in nearly every domain of computer science. As such, there are many different implementations of sorting algorithms, with varying performance and complexity.

One of the simplest sorting algorithms is insertion sort, which revolves around the idea of inserting a single element in an already sorted list. To sort a list of elements using this strategy, we take the next element in the list that is to be considered, insert it into an accumulated sorted list that is initially empty, and proceed recursively until all elements have been inserted. In Haskell (Peyton Jones et al. 2003), insertion sort is concisely expressed using the *foldr* operation on lists, defining it as the application of the *insert* operation to each element of the input list, producing a result starting with the empty list:

$$
\begin{aligned}
&\mathit{insertSort} :: [\mathit{Integer}] \to [\mathit{Integer}] \\
&\mathit{insertSort} = \mathit{foldr}\ \mathit{insert}\ [\,]
\end{aligned}
$$

The *insert* function takes one element and inserts it in an already sorted list. It does this using *span* to break the sorted list into two segments according to the pivot element we want to insert, which is then introduced in between the two parts:

$$
\begin{aligned}
&\mathit{insert} :: \mathit{Integer} \to [\mathit{Integer}] \to [\mathit{Integer}] \\
&\mathit{insert}\ y\ ys = xs +\!\!+ [y] +\!\!+ zs \\
&\quad \mathbf{where}\ (xs, zs) = \mathit{span}\ (\leqslant y)\ ys
\end{aligned}
$$

* This work has been funded by EPSRC grant number EP/J010995/1.

P. Achten and P. Koopman (Eds.): Plasmeijer Festschrift, LNCS 8106, pp. 151–167, 2013.
© Springer-Verlag Berlin Heidelberg 2013

The use of *span* relies on the fact that the list *ys* is already sorted.

Another basic sorting algorithm, selection sort, can be easily expressed in terms of an unfold. Unfolds are a recursion scheme dual to folds, and are used to produce data, instead of consuming data. Unfolds are often given less attention than folds (Gibbons and Jones 1998), but that is not the case in this paper: we assure the reader that we will maintain proportional representation, and show an unfold for every fold. In Haskell, the unfold operation on lists is defined as *unfoldr*:

$$unfoldr :: (b \to Maybe\ (a,b)) \to (b \to [a])$$

The first argument to *unfoldr* defines how to produce lists from a seed: the *Nothing* case corresponds to the empty list, whereas *Just* (a,b) corresponds to a list with element a and new seed b. Using this function, and a starting seed, *unfoldr* produces a complete list.

Selection sort works on a list by recursively picking the smallest element of the input list and adding this element to the result list. It can be defined as the unfold of a *select* operation:

```
selectSort :: [Integer] → [Integer]
selectSort = unfoldr select
```

This *select* operation picks the smallest element from the input list using *minimum*, removes it using *delete*, and continues recursively by returning the smallest element together with the remaining list.

```
select :: [Integer] → Maybe (Integer, [Integer])
select [ ] = Nothing
select xs = Just (x, xs′)
  where x   = minimum xs
        xs′ = delete x xs
```

For practical reasons, the types of *foldr* and *unfoldr* in Haskell are not clearly dual. This contributes to obscuring the inherent duality in our *insert* and *select* functions. The purpose of this paper is to explicitly highlight the duality in sorting algorithms, so that we can provide a unified definition for both *insertSort* and *selectSort*. We do this by exploring a type-directed approach to algorithm development, where the types dictate most of the behaviour of functions.

The remainder of this paper is structured as follows. We first introduce a framework of functors, folds, and unfolds in Section 2, which we use in Section 3 to implement two exchange sorts in one go. To define more efficient insertion and selection sorts, we start by introducing more exotic variants of folds and unfolds, called paramorphisms and apomorphisms, in Section 4. We use these morphisms in Section 5, revisiting the two sorting algorithms shown in this introduction. In Section 6 we turn our attention to mergesort, in order to show how these recursion schemes can be applied to create more efficient sorting algorithms. We conclude our discussion in Section 7.

This paper is built on our earlier work on this same subject (Hinze et al. 2012), but we have rewritten the exposition entirely, simplifying many aspects and removing all the

category theory jargon. The theoretically-inclined reader is referred to the earlier work for a deeper understanding of bialgebras and distributive laws in sorting, but this is not required for the comprehension of this paper; the entire development arises naturally out of a type-directed approach to programming, without need for appealing to category theory for the justification of design choices.

2 Functors, Folds, and Unfolds

In this paper we focus on the duality of folds and unfolds, and how these recursion schemes can be used in sorting. The standard functions *foldr* and *unfoldr* are particularly useful since they allow us to express a whole class of recursive functions. Their utility draws from the fact that folds and unfolds allow us to abstract away from using functions with direct recursion. In the case of folds, the exact site of the recursive step is handled by the *foldr* function, and the non-recursive component is described by its parameters, which constitute a so-called algebra. Dually, unfolds are considered in terms of a corecursive step and a non-recursive coalgebra.

This pattern is mirrored at the level of data and is not unique to lists, where recursive datatypes are described as two-level types (Sheard and Pasalic 2004): one level describes the fact that the data is recursive, and the other is non-recursive and describes the shape of the data. Using this representation we can decompose the list datatype into two parts. First, consider the non-recursive component:

```
data List list = Nil | Cons Integer list
```

We call a datatype such as *List* the *base functor* of the recursive type. For simplicity we consider lists of elements of type *Integer*; our development generalises readily to polymorphic lists with an *Ord* constraint on the element type.

Note that the type of *List* is intrinsically not recursive, but instead uses a parameter where one might expect the recursive site. We can retrieve the usual lists from our non-recursive *List* datatype using the fixed-point combinator *Fix*, which builds recursion into datatypes:

```
newtype Fix f = In {out :: f (Fix f)}
```

Combining these two parts into a two-level type yields *Fix List*, which is isomorphic to the predefined type of integer lists [*Integer*].

The recursive component of this data structure is marked by the *Functor* instance of the base functor, and is key to providing a generalised definition of a fold:

```
instance Functor List where
  fmap f Nil          = Nil
  fmap f (Cons k x)   = Cons k (f x)
```

Note that this is not the same functoriality as the one typically used to express a mapping over the elements in a list.

The advantage of representing lists by their base functor becomes evident when we define the fold and unfold operations. Datatypes defined by abstracting over the recursive positions, like *List*, enjoy a single fold operator:

$$fold :: (Functor\, f) \Rightarrow (f\ a \rightarrow a) \rightarrow (Fix\, f \rightarrow a)$$
$$fold\ a = a \cdot fmap\ (fold\ a) \cdot out$$

The *fold* function takes an argument *a*, called the algebra, that is able to crush one level of the data structure. The definition works by first exposing the top level of the structure by using *out*, which is then crushed at its recursive sites using *fmap*, and finally crushed at the top level using *a*.

There is also a single, generic definition of unfold, which is now clearly dual to fold:

$$unfold :: (Functor\, f) \Rightarrow (a \rightarrow f\ a) \rightarrow (a \rightarrow Fix\, f)$$
$$unfold\ c = In \cdot fmap\ (unfold\ c) \cdot c$$

The first argument to *unfold*, the coalgebra *c*, defines how to expand a seed into some functorial type *f* with seeds at the leaves. The coalgebra is applied by *unfold* recursively until a complete structure is built. Again the recursive site is marked by the *Functor* instance of the structure in question.

Unlike Haskell lists, which have *foldr* and *unfoldr* operations specialised to their type, our *fold* and *unfold* operations work on any datatype with a *Functor* instance, and we will soon make use of this generality. We have only sketched the details of base functors and their recursive morphisms; a more detailed presentation, including relevant category theory background, can be found in Meijer et al. (1991). Another categorical treatment that generalises folds and unfolds to operate on an even wider class of recursive types can be found in Hinze (2011).

3 Sorting by Swapping

In this section we will look at our first sorting algorithms expressed in terms of the generalised folds and unfolds introduced in the previous section, and show how duality naturally arises in this setting. To ease the understanding of the algebras and coalgebras that we will see, which generally perform "one step" of sorting, we introduce a datatype of sorted lists, together with its *Functor* instance:

$$\textbf{data}\ \underline{List}\ list = \underline{Nil} \mid \underline{Cons}\ Integer\ list$$
$$\textbf{instance}\ Functor\ \underline{List}\ \textbf{where}$$
$$\quad fmap\ f\ \underline{Nil} \qquad\qquad = \underline{Nil}$$
$$\quad fmap\ f\ (\underline{Cons}\ k\ list) = \underline{Cons}\ k\ (f\ list)$$

Note that $\underline{List}$ is entirely isomorphic to *List*. The only difference lies in the names used: the fact that a list is sorted is indicated by the underlining on its type and constructor names. The compiler will not be able to enforce the condition that $\underline{List}$ always represents sorted lists for us, but we keep this invariant throughout our development.

A sorting algorithm, in general, takes arbitrary lists to sorted lists:

$$sort :: Fix\ List \rightarrow Fix\ \underline{List}$$

By looking at its type, we can interpret *sort* as either a fold, that consumes a value of type *Fix List*, or as an unfold that produces a value of type *Fix* $\underline{List}$. If *sort* is a *fold*,

its algebra will have type $\mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}) \to \mathit{Fix}\ \underline{\mathit{List}}$. This algebra can then be defined as an *unfold* which produces a value of type $\mathit{Fix}\ \underline{\mathit{List}}$. These observations are summarised in the following type signatures:

$$\begin{aligned}
\mathit{fold}\ (\mathit{unfold}\ c) &:: \mathit{Fix}\ \mathit{List} \to \mathit{Fix}\ \underline{\mathit{List}} \\
\mathit{unfold}\ c &:: \mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}) \to \mathit{Fix}\ \underline{\mathit{List}} \\
c &:: \mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}))
\end{aligned}$$

Using this approach brings an additional benefit: the analysis of the complexity of our algorithms can be framed in terms of the cost of the *fold* and *unfold* functions. The running time of a fully evaluated result of a *fold* is proportional to the depth of its input structure multiplied by the cost of one step of the algebra. Dually, the running time of an *unfold* is proportional to the depth of its output structure multiplied by the cost of one step of the coalgebra. This property will become useful when evaluating the performance of our algorithms.

Let us then write a sorting function as a fold of an unfold:

$$\begin{aligned}
&\mathit{naiveInsertSort} :: \mathit{Fix}\ \mathit{List} \to \mathit{Fix}\ \underline{\mathit{List}} \\
&\mathit{naiveInsertSort} = \mathit{fold}\ (\mathit{unfold}\ \mathit{naiveInsert}) \\
&\mathit{naiveInsert} :: \mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}})) \\
&\mathit{naiveInsert}\ \mathit{Nil} = \underline{\mathit{Nil}} \\
&\mathit{naiveInsert}\ (\mathit{Cons}\ a\ (\mathit{In}\ \underline{\mathit{Nil}})) = \underline{\mathit{Cons}}\ a\ \mathit{Nil} \\
&\mathit{naiveInsert}\ (\mathit{Cons}\ a\ (\mathit{In}\ (\underline{\mathit{Cons}}\ b\ x))) \\
&\quad \mid a \leqslant b \quad\ \ = \underline{\mathit{Cons}}\ a\ (\mathit{Cons}\ b\ x) \\
&\quad \mid \mathit{otherwise} = \underline{\mathit{Cons}}\ b\ (\mathit{Cons}\ a\ x)
\end{aligned}$$

Most of the behaviour of *naiveInsert* follows naturally from its type. The empty and single element unsorted lists are trivially converted into sorted variants. For an unsorted list with at least two elements, we compare the elements, reordering if necessary. What we obtain is a form of "naive" insertion sort, since it does not make use of the fact that the list where an element is being inserted in is already sorted. Instead, the traversal is continued, even though there is no more work to be done. Indeed, the analysis of the time complexity of this algorithm is simple: the input size of the fold is linear, and the output size of the inner unfold is also linear, so we should expect quadratic behaviour. We will see how to make use of the fact that the inner list is already sorted in Section 5.

Recall now that we can also see a sorting function as an unfold of a fold. In that case, the type of the inner algebra can be derived as in the following type signatures:

$$\begin{aligned}
\mathit{unfold}\ (\mathit{fold}\ a) &:: \mathit{Fix}\ \mathit{List} \to \mathit{Fix}\ \underline{\mathit{List}} \\
\mathit{fold}\ a &:: \mathit{Fix}\ \mathit{List} \to \underline{\mathit{List}}\ (\mathit{Fix}\ \mathit{List}) \\
a &:: \mathit{List}\ (\underline{\mathit{List}}\ (\mathit{Fix}\ \mathit{List})) \to \underline{\mathit{List}}\ (\mathit{Fix}\ \mathit{List})
\end{aligned}$$

The sorting algorithm that we obtain as an unfold of a fold is a version of bubble sort:

$$\begin{aligned}
&\mathit{bubbleSort} :: \mathit{Fix}\ \mathit{List} \to \mathit{Fix}\ \underline{\mathit{List}} \\
&\mathit{bubbleSort} = \mathit{unfold}\ (\mathit{fold}\ \mathit{bubble})
\end{aligned}$$

$$
\begin{aligned}
&bubble :: List\ (\underline{List}\ (Fix\ List)) \to \underline{List}\ (Fix\ List)\\
&bubble\ Nil \qquad\qquad = \underline{Nil}\\
&bubble\ (Cons\ a\ \underline{Nil}) = \underline{Cons}\ a\ (In\ Nil)\\
&bubble\ (Cons\ a\ (\underline{Cons}\ b\ x))\\
&\quad \mid a \leqslant b \qquad = \underline{Cons}\ a\ (In\ (Cons\ b\ x))\\
&\quad \mid otherwise = \underline{Cons}\ b\ (In\ (Cons\ a\ x))
\end{aligned}
$$

This algorithm proceeds by continually comparing adjacent elements, swapping them if they are in the wrong order, which is the principal idea behind a bubble sort. The similarity between *bubble* and *naiveInsert* is striking; they differ only in the placement of the fixed-point constructor *In*. This becomes clear if we look at their types, after expanding one definition of *Fix* in each of them:

$$
\begin{aligned}
&naiveInsert :: List\ (\underline{List}\ (Fix\ \underline{List})) \to \underline{List}\ (List\ (Fix\ \underline{List}))\\
&bubble \qquad :: List\ (\underline{List}\ (Fix\ List)) \to \underline{List}\ (List\ (Fix\ List))
\end{aligned}
$$

The only difference is in the inner type of lists at the third level of depth. However, this third level is in some sense redundant, since these algorithms only inspect elements in the first two levels. It is this observation that allows *naiveInsert* and *bubble* to be safely generalised to a *step function* of the following type:

$$swap :: List\ (\underline{List}\ x) \to \underline{List}\ (List\ x)$$

Such a step function is sometimes called a *distributive law*, since it captures an abstract notion of distributivity. The definition of this new function, which we call *swap* since it simply swaps adjacent elements based on their order, is entirely similar to the definitions of both *naiveInsert* and *bubble*:

$$
\begin{aligned}
&swap\ Nil \qquad\qquad = \underline{Nil}\\
&swap\ (Cons\ a\ \underline{Nil}) = \underline{Cons}\ a\ Nil\\
&swap\ (Cons\ a\ (\underline{Cons}\ b\ x))\\
&\quad \mid a \leqslant b \qquad\quad = \underline{Cons}\ a\ (Cons\ b\ x)\\
&\quad \mid otherwise \qquad = \underline{Cons}\ b\ (Cons\ a\ x)
\end{aligned}
$$

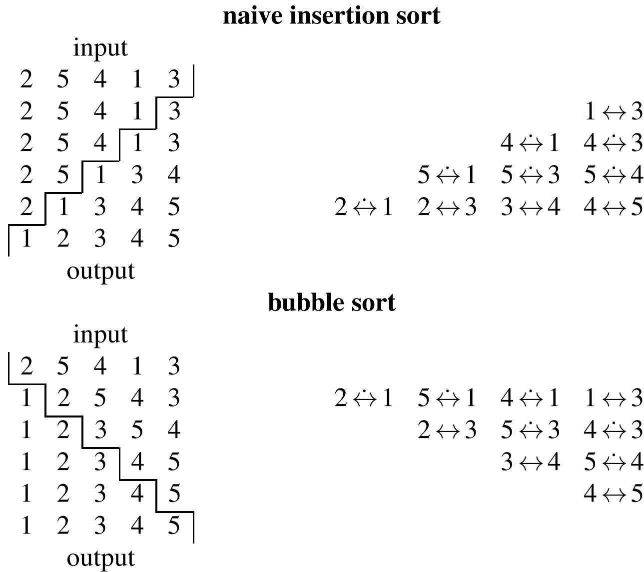
The *swap* function can be understood as a distributive law between the types *List* and *List*, and is a generalisation that captures the essence of both *naiveInsert* and *bubble*. We can use *swap* to define both $naiveInsertSort'$ and $bubbleSort'$:

$$
\begin{aligned}
&naiveInsertSort', bubbleSort' :: Fix\ List \to Fix\ \underline{List}\\
&naiveInsertSort' = fold \quad (unfold\ (swap \cdot fmap\ out))\\
&bubbleSort' \qquad = unfold\ (fold \quad (fmap\ In \cdot swap))
\end{aligned}
$$

The use of *fmap out* in $naiveInsertSort'$, and, dually, *fmap In* in $bubbleSort'$, reflects our expansion of the *Fix* datatype in the type of the (co)algebra. What we have obtained is a single definition for two conceptually distinct sorting algorithms, in terms of a function that expresses how to perform one step of the computation.

At this point it is worth reinforcing our intuition for how these algorithms work. The duality of these sorting algorithms can be seen visually when we assume a call-by-value evaluation order of the definitions. The diagrams below emphasise that the actual

comparisons performed by *swap* are the same, and that the algorithms only differ in the order in which these comparisons are performed:



For both of these algorithms, it is the outer recursion scheme that drives the computation. In the case of *naiveInsertSort′* this is a *fold*, and the progression is depicted by the vertical line that separates sorted from unsorted data working its way from the end of the unsorted list until only a sorted list remains. Since *bubbleSort′* is expressed as an *unfold*, the corresponding diagram is pleasingly dual. The computation starts from the beginning of a sorted list, beginning with an empty list and gradually bubbling values to the front. Notice how for *naiveInsertSort′*, the input remains stable, whereas the output does not, whereas for *bubbleSort′*, it is the output that remains stable, whereas the input does not.

On the right side of these diagrams we have presented the comparisons that take place in the *swap* function, which are effectively determined by the inner recursion. The arrows correspond to comparisons that are made between two elements, and dotted arrows indicate comparisons that result in a swap (this is the case when the element on the left is greater than that on the right). The arrows that correspond to each line of *naiveInsertSort′* show the comparisons that are needed to insert the element immediately to the left of the vertical line, and should be read from left to right. On the other hand, the swaps that correspond to each line of *bubbleSort′* show the comparisons that are needed to select the value to the left of the vertical line, and should be read from right to left.

4 Paramorphisms and Apomorphisms

In the previous section we defined two sorting algorithms with performance $\Theta(n^2)$, where both work in quadratic time regardless of the input. In particular, the unfolding step in the insertion sort will continue to traverse a sorted structure long after a new element has been inserted into its appropriate place. This is unfortunate, as it fails to

make use of the fact that the output that has already been constructed is sorted. If this information was taken into consideration, the inner traversals could be interrupted for better performance. However, using folds and unfolds as recursion schemes prohibits us from meddling with the recursion, and no such interruption is possible. In order to gain explicit control of when an unfold should stop traversing a structure, we turn to a slightly more exotic version of unfold, namely the apomorphism (Vene and Uustalu 1998), which gives us the required ability to abort recursion. Dually, we will also make use of paramorphisms (Meertens 1992), which give us the power to use a part of the input data in an algebra. Paramorphisms and apomorphisms can be understood as the counterparts to folds and unfolds, and enjoy aspects of duality.

Before diving into the details of these recursion schemes, however, it is informative to first consider the duality that exists between so-called product and sum types. The *product* of types is simply a synonym for a pair of values, and the *sum* of types is a synonym for the *Either* type (a choice between two values):

$$\textbf{type}\ a \times b = (a, b)$$
$$\textbf{type}\ a + b = \mathit{Either}\ a\ b$$

In a sense, these operators encode a form of arithmetic on types. Assuming a is a type with m inhabitants, and b a type with n inhabitants, the product type $a \times b$ is inhabited by $m \times n$ values, as we can choose one element of b for each element of a. Similarly, the sum type $a + b$ has $m + n$ inhabitants, as we have to pick either an element from a or an element from b.

The duality between these types can be understood in terms of two operators: one which constructs products, and another which deconstructs sums. The operator $\triangle$, or *split*, constructs a pair by applying two functions with a common source type:

$$(\triangle) :: (x \to a) \to (x \to b) \to (x \to a \times b)$$
$$(f \triangle g)\ x = (f\ x, g\ x)$$

The dual operator ∇, or *join*, deconstructs a sum by applying two functions with a common target type:

$$(\nabla) :: (a \to x) \to (b \to x) \to (a + b \to x)$$
$$(f \nabla g)\ (\mathit{Left}\ \ a) = f\ a$$
$$(f \nabla g)\ (\mathit{Right}\ b) = g\ b$$

Using these operators, we can extend the duality that folds and unfolds enjoy, and define paramorphisms and apomorphisms.

A paramorphism is defined as a variation on *fold* which makes use of a product in the source of its algebra. The product is used to “remember” the original *Fix f* structure:

$$\mathit{para} :: (\mathit{Functor}\ f) \Rightarrow (f\ (\mathit{Fix}\ f \times a) \to a) \to (\mathit{Fix}\ f \to a)$$
$$\mathit{para}\ f = f \cdot \mathit{fmap}\ (\mathit{id} \triangle \mathit{para}\ f) \cdot \mathit{out}$$

The first argument to *para* now has access to both the original *Fix f* structure and the computed result for this same structure. This argument is not an algebra in the categorical sense, but we shall name it so for simplicity, since it serves essentially the same purpose. We will render the product constructor of two elements a and b as $a \risingdotseq b$, to remind us that the first element is the precomputed result of the second.

A typical example of a paramorphism on lists is the function that calculates all *proper* suffixes of a list:

$$
\begin{aligned}
&\mathit{suffixes} :: \mathit{Fix\ List} \to [\mathit{Fix\ List}] \\
&\mathit{suffixes} = \mathit{para\ suf} \\
&\mathit{suf} :: \mathit{List}\ (\mathit{Fix\ List} \times [\mathit{Fix\ List}]) \to [\mathit{Fix\ List}] \\
&\mathit{suf\ Nil} \qquad\qquad\qquad = [\,] \\
&\mathit{suf}\ (\mathit{Cons}\ _n\ (l \mathbin{\triangle} ls)) = l : ls
\end{aligned}
$$

Although it may seem like paramorphisms are more powerful than folds, this is not the case. They simply make certain algorithms more convenient to express by providing direct access to the original structure. This behaviour can also be expressed in a fold; in fact, we can define *para* as a fold:

$$
\begin{aligned}
&\mathit{para}' :: (\mathit{Functor}\, f) \Rightarrow (f\ (\mathit{Fix}\, f \times a) \to a) \to (\mathit{Fix}\, f \to a) \\
&\mathit{para}'\, f = \mathit{snd} \cdot \mathit{fold}\ ((\mathit{In} \cdot \mathit{fmap\, fst}) \mathbin{\triangle} f)
\end{aligned}
$$

Dually, an apomorphism is a variation of an unfold which makes use of a sum in the target of its first argument. As before, we abuse terminology, and name this argument a coalgebra. This sum is used to encode the choice between stopping the apomorphism with a concrete value of type $\mathit{Fix}\, f$, or going on with a new seed of type a:

$$
\begin{aligned}
&\mathit{apo} :: (\mathit{Functor}\, f) \Rightarrow (a \to f\ (\mathit{Fix}\, f + a)) \to (a \to \mathit{Fix}\, f) \\
&\mathit{apo}\, f = \mathit{In} \cdot \mathit{fmap}\ (\mathit{id} \mathbin{\triangledown} \mathit{apo}\, f) \cdot f
\end{aligned}
$$

Here the coalgebra uses its source value to produce either a final result, or an intermediate step. If a final result is given then the recursion no longer continues; otherwise, values are produced just as in an unfold. For mnemonic reasons, we will render the *Left* constructor as *Stop*, as it encodes stopping the recursion, and *Right* as *Go*, as it encodes continuing the traversal.

Note that the power to improve the running time of our sorting algorithms relies on the use of apomorphisms. Paramorphisms are mostly a cosmetic improvement; the resulting traversal still consumes the entire input linearly. Apomorphisms, on the other hand, allow for early termination of the computation, so their running time is no longer necessarily linear on the size of the resulting structure.

4.1 Folds of Apomorphisms, Unfolds of Paramorphisms

As before, we use a type-directed approach to guide the development of a sorting algorithm, except this time we replace the inner recursions with *apo* and *para*, since we are aiming for a more efficient algorithm. Deriving the appropriate algebra and coalgebra yields the following:

$$
\begin{aligned}
\mathit{fold}\ (\mathit{apo}\ c) &:: \mathit{Fix\ List} \to \mathit{Fix}\ \underline{\mathit{List}} \\
\mathit{apo}\ c &:: \mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}) \to \mathit{Fix}\ \underline{\mathit{List}} \\
c &:: \mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\mathit{Fix}\ \underline{\mathit{List}} + \mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}))
\end{aligned}
$$

$$\begin{aligned} \mathit{unfold}\ (\mathit{para}\ a) &:: \mathit{Fix\ List} \to \mathit{Fix}\ \underline{\mathit{List}} \\ \mathit{para}\ a &:: \mathit{Fix\ List} \to \underline{\mathit{List}}\ (\mathit{Fix\ List}) \\ a &:: \mathit{List}\ (\mathit{Fix\ List} \times \underline{\mathit{List}}\ (\mathit{Fix\ List})) \to \underline{\mathit{List}}\ (\mathit{Fix\ List}) \end{aligned}$$

The duality is somewhat hidden by some noise in the types, but can be easily recovered by introducing some synonyms for what are sometimes called *pointed* and *copointed* types, and unrolling some fixpoints:

$$\begin{aligned} &\textbf{type}\ f_{+}\ a = a + f\ a \\ &\textbf{type}\ f_{\times}\ a = a \times f\ a \end{aligned}$$

After unrolling one layer of the fixed point, we obtain the following types for our (co)algebras:

$$\begin{aligned} c \cdot \mathit{fmap\ In} &:: \mathit{List}\ (\underline{\mathit{List}}\ \ \ (\mathit{Fix}\ \underline{\mathit{List}})) \to \underline{\mathit{List}}\ (\mathit{List}_{+}\ (\mathit{Fix}\ \underline{\mathit{List}})) \\ \mathit{fmap\ out} \cdot a &:: \mathit{List}\ (\underline{\mathit{List}}_{\times}\ (\mathit{Fix\ List})) \to \underline{\mathit{List}}\ (\mathit{List}\ \ \ (\mathit{Fix\ List})) \end{aligned}$$

From this it is clear that the (co)algebras are almost of the same form, except that the coalgebra should be modified to consume a copointed type in its source, and the algebra should be modified to produce a pointed type in its target.

This suggests that we can combine the (co)algebras into a single step function with a more general type:

$$b :: \mathit{List}\ (\underline{\mathit{List}}_{\times}\ x) \to \underline{\mathit{List}}\ (\mathit{List}_{+}\ x)$$

For convenience, we shall abuse terminology, and occasionally call such step functions distributive laws, since they serve almost the same purpose as the distributive laws introduced in Section 3. With some gentle massaging, we can use such a step function in the context of either an apomorphic coalgebra or a paramorphic algebra:

$$\begin{aligned} c &= b \cdot \mathit{fmap}\ (\mathit{id} \vartriangle \mathit{out}) :: \mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\mathit{List}_{+}\ (\mathit{Fix}\ \underline{\mathit{List}})) \\ a &= \mathit{fmap}\ (\mathit{id} \triangledown \mathit{In}) \cdot b :: \mathit{List}\ (\underline{\mathit{List}}_{\times}\ (\mathit{Fix\ List})) \to \underline{\mathit{List}}\ (\mathit{Fix\ List}) \end{aligned}$$

Once again, the step function crucially depends on parametricity for unifying algebras and coalgebras.

5 Insertion and Selection Sort

Now that we have apomorphisms, which allow us to stop recursion, we can write a non-naive version of insertion sort that adequately stops traversing the result list once the element has been inserted. Insertion sort is the *fold* of an *apo*:

$$\begin{aligned} &\mathit{insertSort} :: \mathit{Fix\ List} \to \mathit{Fix}\ \underline{\mathit{List}} \\ &\mathit{insertSort} = \mathit{fold}\ (\mathit{apo\ insert}) \end{aligned}$$

The coalgebra *insert* is similar to *naiveInsert*, but with the essential difference that it stops creating the list (with *Stop*) if no swapping is required. Otherwise, it continues traversing (with *Go*):

$$
\begin{array}{l}
\mathit{insert} :: \mathit{List}\ (\mathit{Fix}\ \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\mathit{List}_{+}\ (\mathit{Fix}\ \underline{\mathit{List}})) \\
\mathit{insert}\ \mathit{Nil} = \underline{\mathit{Nil}} \\
\mathit{insert}\ (\mathit{Cons}\ a\ (\mathit{In}\ \underline{\mathit{Nil}})) = \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ (\mathit{In}\ \underline{\mathit{Nil}})) \\
\mathit{insert}\ (\mathit{Cons}\ a\ (\mathit{In}\ (\underline{\mathit{Cons}}\ b\ x'))) \\
\quad \mid a \leqslant b \qquad = \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ (\mathit{In}\ (\underline{\mathit{Cons}}\ b\ x'))) \\
\quad \mid \mathit{otherwise} = \underline{\mathit{Cons}}\ b\ (\mathit{Go}\quad (\mathit{Cons}\ a\ x'))
\end{array}
$$

Because we are using apomorphisms, *insertSort* will run in linear time on a list that is already sorted, as the inner traversal is immediately terminated each time it is started. This behaviour is crucial for the best case behaviour of *insertSort*.

As before, we can find a dual algorithm to *insertSort* that is defined as an *unfold* of a *para*. Instead of writing a specialised algebra, we will directly write the distributive law that can be used both as argument to *apo* and *para*. Its type, as explained in Section 4, is the following:

$$
\mathit{swop} :: \mathit{List}\ (\underline{\mathit{List}}_{\times}\ x) \to \underline{\mathit{List}}\ (\mathit{List}_{+}\ x)
$$

We nickname this function *swop* as it "swaps and stops"; its type indicates that it has access to the sorted list as its argument, and that it can decide to abort recursion when producing a result. Its definition is an unsurprising generalisation of *insert*:

$$
\begin{array}{l}
\mathit{swop}\ \mathit{Nil} = \underline{\mathit{Nil}} \\
\mathit{swop}\ (\mathit{Cons}\ a\ (x \risingdotseq \underline{\mathit{Nil}})) = \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ x) \\
\mathit{swop}\ (\mathit{Cons}\ a\ (x \risingdotseq \underline{\mathit{Cons}}\ b\ x')) \\
\quad \mid a \leqslant b \qquad = \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ x) \\
\quad \mid \mathit{otherwise} = \underline{\mathit{Cons}}\ b\ (\mathit{Go}\quad (\mathit{Cons}\ a\ x'))
\end{array}
$$

Having defined *swop*, we can use it to define an alternative version of *insertSort*, which does not use *insert*:

$$
\begin{array}{l}
\mathit{insertSort}' :: \mathit{Fix}\ \mathit{List} \to \mathit{Fix}\ \underline{\mathit{List}} \\
\mathit{insertSort}' = \mathit{fold}\ (\mathit{apo}\ (\mathit{swop} \cdot \mathit{fmap}\ (\mathit{id} \vartriangle \mathit{out})))
\end{array}
$$

However, being a distributive law, *swop* can also be used to construct the algebra of a paramorphism .The sorting algorithm that we then obtain is selection sort:

$$
\begin{array}{l}
\mathit{selectSort} :: \mathit{Fix}\ \mathit{List} \to \mathit{Fix}\ \underline{\mathit{List}} \\
\mathit{selectSort} = \mathit{unfold}\ (\mathit{para}\ (\mathit{fmap}\ (\mathit{id} \triangledown \mathit{In}) \cdot \mathit{swop}))
\end{array}
$$

Unlike bubble sort (the dual of "naive" insertion sort), selection sort uses the accumulated result x in the $a \leqslant b$ case, meaning the smallest element has been placed in the correct location. We again get two, entirely dual sorting algorithms for the price of one step function.

6 Mergesort

The work in the previous section brought us a slight boost in performance over the naive version of insertion sort. However, its time complexity is still on average $O(n^2)$,

which is bound by the fact that we use folds and unfolds over *lists*: only the lower bound was improved to $\Omega(n)$. To improve on the average bound we must move to a different algorithm where an intermediate data structure with sublinear depth is built from the input list, and then used to produce the output list. This two-phase approach was used in our previous work to synthesise versions of quicksort and heapsort (Hinze et al. 2012). In this section we show the development of mergesort, which improves the average case complexity to $\Theta(n \log n)$.

These two phases can be seen in a typical implementation of mergesort in Haskell, where the recursive nature of the algorithm is expressed directly, rather than through an intermediary datastructure:

```
mergeSort :: [Integer] -> [Integer]
mergeSort as = merge (mergeSort bs) (mergeSort cs)
  where (bs, cs) = split as

split :: [Integer] -> ([Integer], [Integer])
split []         = ([],     [])
split [a]        = ([a],    [])
split (a:b:cs) = (a:as, b:bs)
  where (as, bs) = split cs

merge :: [Integer] -> [Integer] -> [Integer]
merge as     [] = as
merge []     bs = bs
merge (a:as) (b:bs)
  | a <= b      = a : merge as     (b:bs)
  | otherwise = b : merge (a:as) bs
```

In the first phase, *split* is called at each recursive step of *mergeSort*, and recursively splits the input list in two by uninterleaving the elements. The *merge* function performs the second phase of the algorithm, and recursively merges the lists generated in the first phase. In this section we will see how to expose the recursive structure of *mergeSort* as an explicit intermediate data structure, and each phase as a recursive morphism with an associated distributive law. We stress that this structure serves only to turn the recursion into data, allowing for more explicit control of computation, an idea that is echoed in our description of two level types, where recursion in a data structure is decomposed.

When considering which data structure with sublinear depth to use for sorting, one natural choice is the type of balanced binary trees, since these have logarithmic depth in the number of elements they contain. This tree must faithfully represent the structure of the sorting algorithm in question; for quicksort, which picks a pivot element and splits the list in two halves, we would use trees with elements in the branches, where the element would be the pivot, and each branch a fragment of the list. Mergesort, on the other hand, works by merging lists at each recursive step. A tree structure with elements at the leaves is appropriate to encode this behaviour:

```
data Tree tree = Tip | Leaf Integer | Fork tree tree
instance Functor Tree where
  fmap f Tip          = Tip
```

$$
\begin{aligned}
&\mathit{fmap}\ f\ (\mathit{Leaf}\ a) &&= \mathit{Leaf}\ a \\
&\mathit{fmap}\ f\ (\mathit{Fork}\ l\ r) &&= \mathit{Fork}\ (f\ l)\ (f\ r)
\end{aligned}
$$

We include a *Tip* constructor in our datatype that correspond to the empty lists that can be observed in the output of *split*. As before, this is a two-level type, and the recursive site is denoted by the *Functor* instance.

6.1 First Phase: Growing a Tree

Using the same type directed approach as before, we begin by defining a step function that relates our two structures. The type of the function we seek is therefore:

$$\mathit{grow} :: \mathit{List}\ (\mathit{Tree}_{\times}\ x) \rightarrow \mathit{Tree}\ (\mathit{List}_{+}\ x)$$

Defining such a function for the first few cases is unproblematic, but alas, this line of development turns out to be in vein, where our best efforts to deal with the following case are always thwarted:

$$\mathit{grow}\ (\mathit{Cons}\ a\ (t \asymp \mathit{Leaf}\ b)) = \mathit{Fork}\ (\mathit{Go}\ (\mathit{Cons}\ a\ ?))\ (\mathit{Stop}\ t)$$

The goal in this case is to create a *Fork* that contains the element a in one branch, and b in the other. The type of *Tree* demands that the branches in this fork are of type $\mathit{List}_{+}\ x$. Embedding the b in one branch is a simple matter of making use of *Stop* t, since t refers to the *Leaf* b construct. The problem arises in that we cannot provide a suitable second parameter to *Cons*, since the only value of appropriate type x in scope is the t that refers to the *Leaf* b constructor. While this is correctly typed, using this value as a parameter is incorrect for two reasons: first, this would result in a duplication of the value b, and second, it leads to infinite recursion, since in the next step the value *Case* $a\ t$ would considered again. Possible alternative definitions are either merely variations on this theme, or satisfy the types by dropping data, which is manifestly unsatisfactory for a sorting function.

The heart of the problem lies in the fact that the constructors of *List* offer no way of signaling the existence of a singleton that should terminate the recursion. In a sense, there is no constructor in a *List* that is the counterpart to a *Leaf*. To recover a step function that can build trees, the list representation needs a means of expressing singletons as a primitive constructor. One solution is to lift existing lists so that there is a a new constructor *Single Integer*:

$$\mathbf{data}\ \mathit{List}\ \mathit{list} = \mathit{Nil} \mid \mathit{Single}\ \mathit{Integer} \mid \mathit{Cons}\ \mathit{Integer}\ \mathit{list}$$

This type gives us everything we need to build the distributive law that relates lists and trees. The first few cases fall naturally from the types, and there is very little choice in how to proceed:

$$
\begin{aligned}
&\mathit{grow} :: \mathit{List}\ (\mathit{Tree}_{\times}\ x) \rightarrow \mathit{Tree}\ (\mathit{List}_{+}\ x) \\
&\mathit{grow}\ \mathit{Nil} && = \mathit{Tip} \\
&\mathit{grow}\ (\mathit{Single}\ a) && = \mathit{Leaf}\ a
\end{aligned}
$$

$$\begin{aligned}
&\mathit{grow}\ (\mathit{Cons}\ a\ (t \asymp \mathit{Tip})) &&= \mathit{Leaf}\ a\\
&\mathit{grow}\ (\mathit{Cons}\ a\ (t \asymp \mathit{Leaf}\ b)) &&= \mathit{Fork}\ (\mathit{Go}\ (\mathit{Single}\ a))\ (\mathit{Stop}\ t)
\end{aligned}$$

Note that here the problem encountered previously has been circumvented by embedding the value a in a *Single* constructor in the left hand branch of the *Fork*. In a later step, this *Single a* value will be turned into a *Leaf a* as desired.

The remaining final case where the list contains a *Fork* can be implemented in a number of ways. We can arbitrarily insert the element a in either the left or the right side of the *Fork* that is produced. Once this choice is established we have another, more interesting, decision to make with regards to the subtrees given by l and r. One option is to preserve their order:

$$\mathit{grow}\ (\mathit{Cons}\ a\ (t \asymp \mathit{Fork}\ l\ r)) = \mathit{Fork}\ (\mathit{Go}\ (\mathit{Cons}\ a\ l))\ (\mathit{Stop}\ r)$$

However, this solution leads to trees that are unbalanced in that new elements are always inserted on the left side of the tree. The only other option is to reverse the two subtrees when inserting an element:

$$\mathit{grow}\ (\mathit{Cons}\ a\ (t \asymp \mathit{Fork}\ l\ r)) = \mathit{Fork}\ (\mathit{Go}\ (\mathit{Cons}\ a\ r))\ (\mathit{Stop}\ l)$$

In so doing, we have rediscovered Braun's method of producing perfectly balanced trees (Braun and Rem 1983). In fact, this trick can also be seen in an alternative defintion of *split* that considers only empty and non-empty lists, and rotates lists as it recurses.

Since we have described a distributive law, two methods for producing trees emerge:

$$\begin{aligned}
&\mathit{makeTree}, \mathit{makeTree}' :: \mathit{Fix\ List} \to \mathit{Fix\ Tree}\\
&\mathit{makeTree}\ \ = \mathit{fold}\ \ \ (\mathit{apo}\ \ (\mathit{grow} \cdot \mathit{fmap}\ (\mathit{id} \vartriangle \mathit{out})))\\
&\mathit{makeTree}' = \mathit{unfold}\ (\mathit{para}\ (\mathit{fmap}\ (\mathit{id} \triangledown \mathit{In}) \cdot \mathit{grow}))
\end{aligned}$$

The first method, *makeTree*, encodes the standard way of building a tree by repeated insertion, using Braun's method for keeping the tree balanced. The second method, *makeTree*$'$, encodes the slightly more unusual process of generating a tree by repeatedly uninterleaving a list. This uninterleaving of the list has the same swapping behaviour as Braun's method on trees.

6.2 Second Phase: Merging Trees

Once the tree is constructed, the second phase of the algorithm reduces the tree until a sorted list is produced. The distributive law falls out naturally from the types:

$$\begin{aligned}
&\mathit{merge} :: \mathit{Tree}\ (\underline{\mathit{List}}_{\times}\ x) \to \underline{\mathit{List}}\ (\mathit{Tree}_{+}\ x)\\
&\mathit{merge}\ \mathit{Tip} && &&= \underline{\mathit{Nil}}\\
&\mathit{merge}\ (\mathit{Leaf}\ a) && &&= \underline{\mathit{Cons}}\ a\ (\mathit{Go\ Tip})\\
&\mathit{merge}\ (\mathit{Fork}\ (l \asymp \underline{\mathit{Nil}}) &&(r \asymp \underline{\mathit{Nil}})) &&= \underline{\mathit{Nil}}\\
&\mathit{merge}\ (\mathit{Fork}\ (l \asymp \underline{\mathit{Nil}}) &&(r \asymp \underline{\mathit{Cons}}\ b\ r')) &&= \underline{\mathit{Cons}}\ b\ (\mathit{Stop}\ r')\\
&\mathit{merge}\ (\mathit{Fork}\ (l \asymp \underline{\mathit{Cons}}\ a\ l') &&(r \asymp \underline{\mathit{Nil}})) &&= \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ l')\\
&\mathit{merge}\ (\mathit{Fork}\ (l \asymp \underline{\mathit{Cons}}\ a\ l') &&(r \asymp \underline{\mathit{Cons}}\ b\ r'))
\end{aligned}$$

$$
\begin{aligned}
&\mid a \leqslant b &&= \underline{Cons}\ a\ (Go\ (Fork\ l'\ r)) \\
&\mid otherwise &&= \underline{Cons}\ b\ (Go\ (Fork\ l\ \ r'))
\end{aligned}
$$

Empty trees give rise to empty lists, and a single element tree produces a single element list. When we have two subtrees, we inspect the list contained in each subtree. If both lists are empty, we return the empty list. In case only one of the lists is non-empty, the element is added to the front of the output sorted list, and the recursion stops with the tail in hand. In the most general case, we have one element in each branch. We compare the two elements, picking the smallest one and recursing using the appropriate subtrees.

Note that for this phase, lists do not need to be extended with a *Single* constructor, since in the case of *Leaf a* the fact that the $\underline{Cons}\ a$ should terminate is signalled by embedding a *Tip* in the tail. This *Tip* is later turned into a $\underline{Nil}$ as desired.

As before, we obtain two methods for merging a tree into a list:

$$
\begin{aligned}
&mergeTree, mergeTree' :: Fix\ Tree \to Fix\ \underline{List} \\
&mergeTree\ \ = fold\ \ \ \ (apo\ \ (merge \cdot fmap\ (id \vartriangle out))) \\
&mergeTree' = unfold\ (para\ (fmap\ (id \triangledown In) \cdot merge))
\end{aligned}
$$

An operational understanding of how these algorithms work helps develop our understanding of the sense in which duality expresses itself here.

The first of these two variations uses a *fold* in the outer recursion, and this drives the deconstruction of the tree of values from the bottom until a single list remains. The controlling *fold* has an algebra of type:

$$
Tree\ (Fix\ \underline{List}) \to Fix\ \underline{List}
$$

Here, the algebra is in fact an apomorphism that starts working at the bottom of the tree. When a *Tip* is encountered, the *merge* will simply produce a $\underline{Nil}$, and control is passed back to the *fold*. Otherwise, each *Leaf a* is initially turned into a $\underline{Cons}\ a\ (Go\ Tip)$, and the apomorphism continues to unfold the remaining *Tip* resulting in a remaining structure that is $\underline{Cons}\ a\ \underline{Nil}$. When the *fold* encounters a *Fork*, the contents of each branch are analysed by the apomorphism. In the case where both lists are empty, they become a single $\underline{Nil}$. When only one contains data, that branch is turned into a *Cons* with this payload, and the apomorphism is signalled to continue using the appropriate tail with a *Stop*. The more interesting case is when both branches contain data. In this case, the *merge* function is used to collapse the fork so that the least value is placed at the beginning of the new $\underline{List}$. The apomorphism is then signalled with *Go* to continue merging the remaining tail of the list that contained the least element, and the whole of the other list that contained the greater element. This control is directed by constructing a new *Fork* that contains these two lists that must be merged, and the *Go* constructor signifies that the apomorphism must continue merging. Once the apomorphism has finished merging the lists, control is given back to the *fold*, which will continue bottom-up, collapsing the tree using the apomorphism, until a single list remains.

The second variation makes use of an *unfold* that focuses on producing the ensuing sorted list. The *unfold* has a coalgebra that has the type:

$$
Fix\ Tree \to \underline{List}\ (Fix\ Tree)
$$

To determine the next element of the ordered list that is to be produced, the *unfold* must apply a paramorphism to its seed tree. The work of this paramorphism can be thought of as collapsing the tree into either a $\underline{Nil}$ when the tree is empty and the work is finished, or a $\underline{Cons}\ a\ t$, where a is the least value in the tree, and t is the tree that is to be used as the next seed. The tree is collapsed from the bottom, where the paramorphism is applied recursively at each *In Fork* until either an *In Tip* or $In\ (Leaf\ a)$ is reached. *In Tip* values are turned into $\underline{Nil}$ values, and $In\ (Leaf\ a)$ values are turned into $\underline{Cons}\ a\ (In\ Tip)$ values. These intermediate results are then combined bottom-up at each *Fork* by the function *merge*, which maintains the invariant that $\underline{Cons}\ a\ t$ contains the least element a, and t is the remaining tree. When two such $\underline{Cons}$ constructors meet at a fork, the least value is kept, and the seed tree is built out of a new *Fork* and the appropriate subtrees.

While these two algorithms certainly share many characteristics, their operation differs significantly. The behaviour of *mergeTree* is closer to the traditional merge-sort, where lists are successively merged together until only one remains. In contrast, $mergeTree'$ behaves much more like a weak kind of heap sort, where the least element is floated out of the tree structure, but no heap property is maintained on the remaining tree.

6.3 Merging After Growing

Combining these two phases, we can write four variations of mergesort, where simple functional composition combines the various functions we have already discussed:

$$
\begin{aligned}
&mergeSort, mergeSort', mergeSort'', mergeSort''' :: Fix\ List \to Fix\ \underline{List} \\
&mergeSort \ \ \ = mergeTree \cdot makeTree \\
&mergeSort' \ \ = mergeTree \cdot makeTree' \\
&mergeSort'' \ = mergeTree' \cdot makeTree \\
&mergeSort''' = mergeTree' \cdot makeTree'
\end{aligned}
$$

In this section we have used the *Tree* structure as a concrete representation of the implicit way mergesort works. Generally, however, these intermediate representations are inefficient, and can be fused away in a process called deforestation (Wadler 1988). A deforested version of the above algorithms would look similar to *mergeSort*, as shown in the beginning of this section. In the fused version, the *Tree* data structure disappears, instead becoming implicit from the recursive structure of the function.

7 Conclusion

In this paper we have revisited our previous work on sorting with bialgebras and distributive laws (Hinze et al. 2012), recasting it in a more applied setting without use of category theory. Due to the structure of recursive morphisms, and through the use of a type-directed approach for program construction, we have not lost the intuition behind our development; the duality is obvious in each sorting method, giving us “algorithms for free”, and helping to understand the relations between different sorting methods.

Even though we have chosen Haskell as the presentation language for this paper, our developments readily generalise to other functional programming languages. In particular, all the code shown compiles with the “exchanging sources” version of the Clean

compiler (Van Groningen et al. 2010) (after some minor refactoring to remove symbolic operators). This reinforces the argument that sorting algorithms become more clean and elegant when expressed as distributive laws in recursive morphisms.

References

Braun, W., Rem, M.: A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology (1983)

Gibbons, J., Jones, G.: The Under-Appreciated Unfold. In: Proceedings of the International Conference on Functional Programming, ICFP 1998, pp. 273–279. ACM (1998), doi:10.1145/289423.289455

van Groningen, J., van Noort, T., Achten, P., Koopman, P., Plasmeijer, R.: Exchanging sources between Clean and Haskell: A double-edged front end for the Clean compiler. In: Proceedings of the Third ACM Haskell Symposium on Haskell, Haskell 2010, pp. 49–60. ACM (2010), doi:10.1145/1863523.1863530

Hinze, R.: Generic programming with adjunctions. In: Gibbons, J. (ed.) Generic and Indexed Programming. LNCS, vol. 7470, pp. 47–129. Springer, Heidelberg (2012)

Hinze, R., James, D.W.H., Harper, T., Wu, N., Magalhães, J.P.: Sorting with bialgebras and distributive laws. In: Proceedings of the 8th ACM SIGPLAN Workshop on Generic Programming, WGP 2012, pp. 69–80. ACM (2012), doi:10.1145/2364394.2364405

Knuth, D.E.: The Art of Computer Programming, 2nd edn. Sorting and Searching, vol. 3. Addison-Wesley (1998)

Meertens, L.: Paramorphisms. Formal Aspects of Computing 4(5), 413–424 (1992), doi:10.1007/BF01211391

Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 124–144. Springer, Heidelberg (1991)

Peyton Jones, S., et al.: Haskell 98, Language and Libraries. The Revised Report. Cambridge University Press (2003), A special issue of JFP

Sheard, T., Pasalic, T.: Two-level types and parameterized modules. Journal of Functional Programming 14(5), 547–587 (2004)

Vene, V., Uustalu, T.: Functional programming with apomorphisms (corecursion). Proceedings of the Estonian Academy of Sciences: Physics, Mathematics 47(3), 147–161 (1998)

Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: Ganzinger, H. (ed.) ESOP 1988. LNCS, vol. 300, pp. 344–358. Springer, Heidelberg (1988)