Submitted to WGP '12

Sorting with Bialgebras and Distributive Laws

Ralf Hinze Daniel W. H. James Thomas Harper Nicolas Wu José Pedro Magalhães

Department of Computer Science, University of Oxford

{ralf.hinze;daniel.james;tom.harper;nicolas.wu;jose.pedro.magalhaes}@cs.ox.ac.uk

Abstract

Sorting algorithms are an intrinsic part of functional programming folklore as they exemplify algorithm design using folds and unfolds. This has given rise to an informal notion of duality among sorting algorithms: insertion sorts are dual to selection sorts. Using bialgebras and distributive laws, we formalise this notion within a categorical setting. We use types as a guiding force in exposing the recursive structure of bubble, insertion, selection, quick, tree, and heap sorts. Moreover, we show how to distill the computational essence of these algorithms down to one-step operations that are expressed as natural transformations. From this vantage point, the duality is clear, and one side of the algorithmic coin will neatly lead us to the other "for free". As an optimisation, the approach is also extended to paramorphisms and apomorphisms, which allow for more efficient implementations of these algorithms than the corresponding folds and unfolds.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.2.2 [*Analysis of Algorithms and Problem Complexity*]: Nonnumerical Algorithms and Problems—Sorting and searching

General Terms Algorithms, Languages

1. Introduction

Sorting algorithms are often the first non-trivial programs that are introduced to fledgling programmers. The problem of sorting lends itself to a myriad of algorithmic strategies with varying asymptotic complexities to explore, making it an ideal pedagogical tool. Within the functional programming community, the insertion sort also serves to exemplify the use of folds, where the key is to define an appropriate function *insert* which inserts a value into a sorted list.

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

The insertion function partitions a given list into two using an ordering of its elements with respect to the value to be inserted. This value is then inserted in between the partitions:

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

This is an entirely routine and naïve definition, which makes use of the *partition* function from the list utilities section of the Haskell Report. When the input list *ys* is ordered, *insert y ys* adds *y* to the list *ys* and maintains the invariant that the ensuing list is ordered. Thus, we are able to fold an unordered list into an ordered one when we start with an empty list as the initial value of the fold.

Perhaps less well known is that an alternative sorting algorithm, selection sort, can be written in terms of an unfold. An unfold can be thought of as the dual of a fold: a fold consumes a list, whereas unfold produces a list, as evident in the type of *unfoldr*:

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

A selection sort constructs an ordered list by repeatedly extracting the least element from an unordered list. This effectively describes an unfold where the input seed is an unordered list that is used to produce an ordered list:

$$\begin{aligned}&\mathit{selectSort} :: [\mathit{Integer}] \to [\mathit{Integer}]\\&\mathit{selectSort} = \mathit{unfoldr}\ \mathit{select}\end{aligned}$$

The function *select* removes the least element from its input list, and returns that element along with the original list with the element removed. When the list is empty, the function signals that the unfolding must finish.

$$\begin{aligned}&\mathit{select} :: [\mathit{Integer}] \to \mathit{Maybe}\ (\mathit{Integer},[\mathit{Integer}])\\&\mathit{select}\ [\,] = \mathit{Nothing}\\&\mathit{select}\ \mathit{xs} = \mathit{Just}\ (x,\mathit{xs}')\\&\quad \textbf{where}\ x = \mathit{minimum}\ \mathit{xs}\\&\qquad\qquad \mathit{xs}' = \mathit{delete}\ x\ \mathit{xs}\end{aligned}$$

With a little intuition, one might see that these two sorting algorithms are closely related, since they fundamentally complement one another on two levels: folds dualise unfolds, and insertion dualises selection. However, the details of this relationship are somewhat shrouded by our language: the connection between the ingredients of *insert* and *select* is difficult to spot since *append* and *partition* seem to have little to do with *minimum* and *delete*. Furthermore, the rendition of *insert* and *select* in terms of folds and unfolds is not straightforward.

In order to illuminate the connection, we use a type-driven approach to synthesise these algorithms, where notions from category theory are used to guide the development. As we shall see, naïve variants of *insert* and *select* can be written as an unfold and fold, respectively, thus revealing that they are in fact dual. As a consequence, each one gives rise to the other in an entirely mechanical fashion: we effectively obtain algorithms for free. We will obtain the true *select* and *insert* with alternative recursion schemes.

Of course, both of these algorithms are inefficient, taking quadratic time in the length of the input list to compute, and in practice these toy examples are soon abandoned in favour of more practical sorting algorithms. As it turns out, our venture into understanding the structural similarities between *insertSort* and *selectSort* will not be in vain: the insights we shall gain will become useful when we investigate more efficient sorting algorithms.

The main contributions of the paper are as follows:

- A type-driven approach to the design of sorting algorithms using folds and unfolds, which we then extend to paramorphisms and apomorphisms in order to improve efficiency.
- An equivalence of sorting algorithms, which allows us to formalise folkloric relationships such as the one between insertion and selection sort.
- Algorithms for free; because the concepts we use to develop these algorithms dualise, each sorting algorithm comes with another for free.
- As a consequence of this formalisation, we relate bialgebras and distributive laws to folds of apomorphisms and unfolds of paramorphisms.

We continue this paper with a gentle introduction to folds, unfolds, and type-directed algorithm design in Section 2. Then, we delve into sorting by swapping in Section 3, defining two sorting algorithms at once using a distributive law with folds and unfolds. In Section 4, we introduce para- and apomorphisms and use them to define insertion and selection sort in Section 5. We move on to faster sorting algorithms in Section 6 (quicksort) and Section 7 (heapsort). Finally, we review related work in Section 8, and conclude in Section 9.

2. Preliminaries

The standard definitions of *foldr* and *unfoldr* in Haskell obscure the underlying theory that gives us these recursive schemes because they are specialised to lists; these schemes in fact generalise to a large class of recursive datatypes. Here, we give an alternate presentation of recursive datatypes, folds, and unfolds, that provides a more transparent connection to the theory presented in this paper. For this and subsequent sections, we assume a familiarity with folds, unfolds, and their characterisation as initial algebras and final coalgebras. We have otherwise attempted to keep the categorical jargon light, giving brief explanations where necessary.

Folds, also called catamorphisms, provide a recursion scheme for consuming a data structure by combining its elements to produce a value. The idea is that the recursion scheme follows the shape of the data structure, and the details of combining the elements are given by the functions that replace the constructors. Together, these functions constitute the *algebra* of the fold.

It is possible to define recursive datatypes in such a way that the definition of *fold* shows this connection more transparently than the usual Haskell definition. To do this, we introduce the view of recursive datatypes as fixpoints. First, the type

$$\textbf{newtype}\ \mu f = \mathit{In}\ \{\mathit{in}^{\circ} :: f\ (\mu f)\}$$

takes a functor to its least fixed point. When used in a point-free manner, *In* will be written as *in*, but *In a* will be written as $\lceil a \rceil$. As an example of building a recursive datatype, consider the functor

$$\textbf{data}\ \mathit{List}\ \mathit{list} = \mathit{Nil} \mid \mathit{Cons}\ K\ \mathit{list}$$

where we use K to represent some ordered key type. Note that we deliberately introduce lists that are not parametric because this simplifies the exposition, and parametricity with type class constraints can be reintroduced without affecting the underlying theory. As its name suggests, this datatype is similar to that of lists with elements of type K. In this case, however, the recursive argument to *Cons* has been abstracted into a type parameter. We call such a datatype the *base functor* of a recursive datatype, and the functorial action of *map* marks the point of recursion within the datatype:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ \mathit{List}\ \textbf{where}\\
&\quad \mathit{map}\ f\ \mathit{Nil} = \mathit{Nil}\\
&\quad \mathit{map}\ f\ (\mathit{Cons}\ k\ x) = \mathit{Cons}\ k\ (f\ x)
\end{aligned}$$

We then tie the recursive knot by taking the least fixed point $\mu \mathit{List}$, which represents the type of finite lists with elements of type K. In a category theoretic context, $\langle \mu\mathsf{F}, \mathit{in}\rangle$ is the initial algebra of the functor F.

Now that datatypes and algebras are to be defined in terms of base functors, it is possible to give a generic definition of *fold*:

$$\begin{aligned}
&\mathit{fold} :: (\mathit{Functor}\ f) \Rightarrow (f\ a \to a) \to \mu f \to a\\
&\mathit{fold}\ f = f \cdot \mathit{map}\ (\mathit{fold}\ f) \cdot \mathit{in}^{\circ}
\end{aligned}$$

This definition of *fold* only depends on the base functor; this determines the type of the algebra, the shape of the data structure, and the recursive pattern over it (via the definition of *map*). One of the impacts of such a relationship is that the control flow of any program written as a fold matches the data structure. Assuming that the running time of an algebra is constant, this means that the running time of a fold is always linear in the size of input. Such a property can be a powerful guarantee, but also an onerous requirement when the control flow of an algorithm does not precisely match the data structure, as we will show. Note that our cost model will assume that Haskell is strict in order to avoid the additional complexity of lazy evaluation. We will continue in this manner, as such issues are not relevant to any of our discussions.

As a short example of the type-directed approach that we will follow again and again, we point out that we can write $\mathit{in}^{\circ} :: \mu\mathsf{F} \to \mathsf{F}\ (\mu\mathsf{F})$ in terms of *in*. It is a function from $\mu\mathsf{F}$, so we should try a fold: we simply need an algebra of type $\mathsf{F}\ (\mathsf{F}\ (\mu\mathsf{F})) \to \mathsf{F}\ (\mu\mathsf{F})$. An obvious candidate is *map in*, so $\mathit{in}^{\circ} = \mathit{fold}\ (\mathit{map}\ \mathit{in})$; we will see this again at the end of the section.

Dual to folds are unfolds, also known as anamorphisms, which provide a scheme for producing a data structure instead of consuming one. This requires the dual view of recursive datatypes as greatest fixed points of functors, which is defined as

$$\textbf{newtype}\ \nu f = \mathit{Out}^{\circ}\ \{\mathit{out} :: f\ (\nu f)\}$$

When used in a point-free manner, Out° will be written as out°, but $\mathit{Out}^{\circ}\ a$ will be written as $\lfloor a \rfloor$. Using the base functor *List*, $\nu \mathit{List}$ also ties the recursive knot, and represents the type of indefinite lists. However, instead of algebras and folds, we are now concerned with coalgebras and unfolds. A coalgebra is a function $b \to \mathit{List}\ b$, where b is the type of the seed value. As the categorical dual of an initial algebra, $\langle \nu\mathsf{F}, \mathit{out}\rangle$ is the final coalgebra of the functor F.

We can now define *unfold* in the same manner as *fold*, where the details of the recursive scheme depend only on the base functor of the datatype being produced:

$$\begin{aligned}
&\mathit{unfold} :: (\mathit{Functor}\ f) \Rightarrow (a \to f\ a) \to (a \to \nu f)\\
&\mathit{unfold}\ f = \mathit{out}^{\circ} \cdot \mathit{map}\ (\mathit{unfold}\ f) \cdot f
\end{aligned}$$

Again, the placement of the recursive calls is determined by the definition of *map*. As with folds, the control flow of unfolds is determined by the base functor (and therefore the shape of the data structure). In this case, this means that, given a coalgebra with a constant running time, the running time of an unfold is linear in the size of the output. As with folds, this is an important fact to keep in mind in subsequent sections.

We can again use a type-directed approach to express $\mathit{out}^{\circ} :: \mathsf{F}\ (\nu\mathsf{F}) \to \nu\mathsf{F}$ in terms of *out*. It is a function to $\nu\mathsf{F}$, so this time we should try an unfold. As one would expect from duality, $\mathit{out}^{\circ} = \mathit{unfold}\ (\mathit{map}\ \mathit{out})$.

Because the type declarations for the fixed points of functors were given in Haskell, the difference between greatest and least fixed points is not obvious; the definitions are the same except for the names of the constructors and destructors, and these two datatypes are isomorphic, a property known as *algebraic compactness*. While this is the case for Haskell, it is not true in general and we do not depend on this. We will therefore be explicit about whether we are working with $\mu\mathsf{F}$ or $\nu\mathsf{F}$ by using different types. We will also be explicit when we move from $\nu\mathsf{F}$ to $\mu\mathsf{F}$ by using

$$\begin{aligned}&\mathit{downcast} :: (\mathit{Functor}\, f) \Rightarrow \nu f \to \mu f\\&\mathit{downcast} = \mathit{in} \cdot \mathit{map}\ \mathit{downcast} \cdot \mathit{out}\end{aligned}$$

We can always go the other way and embed the least into the greatest with a function $\mathit{upcast} :: (\mathit{Functor}\, f) \Rightarrow \mu f \to \nu f$. How can we define *upcast*? Let us first discuss a small generalisation: given the concrete base functors F and G, how can we write a function of type $\mu\mathsf{F} \to \nu\mathsf{G}$? We will follow a type directed approach; it is a function from $\mu\mathsf{F}$, so we can write it as a fold:

$$\begin{aligned}\mathit{fold}\ (\mathit{unfold}\ c) &: \mu\mathsf{F} \to \nu\mathsf{G}\\ \mathit{unfold}\ c &: \mathsf{F}\ (\nu\mathsf{G}) \to \nu\mathsf{G}\\ c &: \mathsf{F}\ (\nu\mathsf{G}) \to \mathsf{G}\ (\mathsf{F}\ (\nu\mathsf{G}))\\ &\cong \mathsf{F}\ (\mathsf{G}\ (\nu\mathsf{G})) \to \mathsf{G}\ (\mathsf{F}\ (\nu\mathsf{G}))\end{aligned}$$

In fact, it is a fold of an unfold. (The types on the right progress from top to bottom, the terms of the left are built from bottom to top.) Alternatively, *upcast* is a function to $\nu\mathsf{G}$, so we can write it as an unfold:

$$\begin{aligned}\mathit{unfold}\ (\mathit{fold}\ a) &: \mu\mathsf{F} \to \nu\mathsf{G}\\ \mathit{fold}\ a &: \mu\mathsf{F} \to \mathsf{G}\ (\mu\mathsf{F})\\ a &: \mathsf{F}\ (\mathsf{G}\ (\mu\mathsf{F})) \to \mathsf{G}\ (\mu\mathsf{F})\\ &\cong \mathsf{F}\ (\mathsf{G}\ (\mu\mathsf{F})) \to \mathsf{G}\ (\mathsf{F}\ (\mu\mathsf{F}))\end{aligned}$$

This time it is an unfold of a fold. In both cases, we have gone one step further and expanded the recursive types so that we could reveal that the type of the coalgebra c is almost the same as the type of the algebra a. This suggests that a and c are both instances of some function $s :: \mathsf{F}\ (\mathsf{G}\ x) \to \mathsf{G}\ (\mathsf{F}\ x)$ that is parametric in x. We will revisit this idea in the next section.

Now to define *upcast*: it is a specialisation of the above case, so we need either an algebra $\mathsf{F}\ (\mathsf{F}\ (\mu\mathsf{F})) \to \mathsf{F}\ (\mu\mathsf{F})$ or a coalgebra $\mathsf{F}\ (\nu\mathsf{F}) \to \mathsf{F}\ (\mathsf{F}\ (\nu\mathsf{F}))$. We have seen these before when defining in° and out°: the former is simply *map in*, and the latter *map out*.

$$\begin{aligned}&\mathit{upcast} :: (\mathit{Functor}\, f) \Rightarrow \mu f \to \nu f\\&\mathit{upcast} = \mathit{fold}\quad (\mathit{unfold}\ (\mathit{map}\ \mathit{out})) = \mathit{fold}\quad \mathit{out}^\circ\\&\qquad\ \ = \mathit{unfold}\ (\mathit{fold}\quad (\mathit{map}\ \mathit{in}))\ \ = \mathit{unfold}\ \mathit{in}^\circ\end{aligned}$$

Why are these equal? We will see why in the next section.

3. Sorting by Swapping

With the preliminaries of folds and unfolds in place, we now turn our attention back to sorting algorithms. First, we start by creating a new datatype to represent the base functor of sorted lists:

$$\begin{aligned}&\mathbf{data}\ \underline{\mathit{List}}\ \mathit{list} = \underline{\mathit{Nil}} \mid \underline{\mathit{Cons}}\ K\ \mathit{list}\\&\mathbf{instance}\ \mathit{Functor}\ \underline{\mathit{List}}\ \mathbf{where}\\&\quad \mathit{map}\, f\ \underline{\mathit{Nil}} \qquad\qquad = \underline{\mathit{Nil}}\\&\quad \mathit{map}\, f\ (\underline{\mathit{Cons}}\ k\ \mathit{list}) = \underline{\mathit{Cons}}\ k\ (f\ \mathit{list})\end{aligned}$$

Note that $\underline{\mathit{List}}$ is defined exactly like *List*, but we maintain the invariant that the elements in a $\underline{\mathit{List}}$ are sorted. Our goal is to express sorting algorithms as some function *sort*, with the following type:

$$\mathit{sort} :: \mu\mathit{List} \to \nu\underline{\mathit{List}}$$

This precisely captures the notion that we will be consuming, or folding over, an input list in order to produce, or unfold into, an ordered list. This choice of type is motivated by the fact that there is a unique arrow from an initial object, in this case $\mu\mathit{List}$, and there is a unique arrow to a final object, in this case $\nu\underline{\mathit{List}}$.

In Section 1, we wrote selection sort as an unfold. Let us replay this construction, but now with the definitions from Section 2 and following our type directed theme. What we obtain is not the true selection sort, but bubble sort:

$$\begin{aligned}&\mathit{bubbleSort} :: \mu\mathit{List} \to \nu\underline{\mathit{List}}\\&\mathit{bubbleSort} = \mathit{unfold}\ \mathit{bubble}\\&\quad \mathbf{where}\ \mathit{bubble} = \mathit{fold}\ \mathit{bub}\end{aligned}$$

$$\begin{aligned}&\mathit{bub} :: \mathit{List}\ (\underline{\mathit{List}}\ (\mu\mathit{List})) \to \underline{\mathit{List}}\ (\mu\mathit{List})\\&\mathit{bub}\ \mathit{Nil} \qquad\qquad = \underline{\mathit{Nil}}\\&\mathit{bub}\ (\mathit{Cons}\ a\ \underline{\mathit{Nil}}) = \underline{\mathit{Cons}}\ a\ \lceil \mathit{Nil} \rceil\\&\mathit{bub}\ (\mathit{Cons}\ a\ (\underline{\mathit{Cons}}\ b\ x))\\&\quad \mid a \leqslant b \qquad\ \ = \underline{\mathit{Cons}}\ a\ \lceil \mathit{Cons}\ b\ x \rceil\\&\quad \mid \mathit{otherwise} \quad = \underline{\mathit{Cons}}\ b\ \lceil \mathit{Cons}\ a\ x \rceil\end{aligned}$$

This is because the *select* operation should select the minimum element but leave the remaining list unchanged. Instead, *fold bub* produces the swapping behaviour seen in bubble sort. Since *bub* is a constant-time operation, bubble sort is a quadratic-time algorithm (the input and the output list have the same length).

We also wrote insertion sort as a fold. If we write it as a fold of an unfold, we obtain a *naïve* version of insertion sort.

$$\begin{aligned}&\mathit{naïveInsertSort} :: \mu\mathit{List} \to \nu\underline{\mathit{List}}\\&\mathit{naïveInsertSort} = \mathit{fold}\ \mathit{naïveInsert}\\&\quad \mathbf{where}\ \mathit{naïveInsert} = \mathit{unfold}\ \mathit{naïveIns}\end{aligned}$$

$$\begin{aligned}&\mathit{naïveIns} :: \mathit{List}\ (\nu\underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\mathit{List}\ (\nu\underline{\mathit{List}}))\\&\mathit{naïveIns}\ \mathit{Nil} \qquad\qquad\quad = \underline{\mathit{Nil}}\\&\mathit{naïveIns}\ (\mathit{Cons}\ a\ \lfloor \underline{\mathit{Nil}} \rfloor) = \underline{\mathit{Cons}}\ a\ \mathit{Nil}\\&\mathit{naïveIns}\ (\mathit{Cons}\ a\ \lfloor \underline{\mathit{Cons}}\ b\ x \rfloor)\\&\quad \mid a \leqslant b \qquad\qquad\quad = \underline{\mathit{Cons}}\ a\ (\mathit{Cons}\ b\ x)\\&\quad \mid \mathit{otherwise} \qquad\quad\ = \underline{\mathit{Cons}}\ b\ (\mathit{Cons}\ a\ x)\end{aligned}$$

Why have we labelled our insertion sort as naïve? This is because we are not making use of the fact that the incoming list is ordered—compare the types of *bub* and *naïveIns*. We will see how to capitalise on the type of *naïveIns* in Section 5.

Our *bub* and *naïveIns* are examples of the abstract algebra a and coalgebra c that we discussed at the end of the previous section. As pointed out then, the similarities in the types are plain to see, but another observation now is that the implementations of *bub* and *naïveIns* are almost identical. The only difference is that $\lceil - \rceil$ appears on the left in *bub*, and $\lfloor - \rfloor$ appears on the right in *naïveIns*. At the end of the previous section, we suggested that there must be some parametric function that generalises both the algebra and coalgebra. As bubble and naïve insertion sorts are swapping sorts, we will call this function *swap*.

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

This parametric function extracts the computational 'essence' of bubble and naïve insertion sorting. It expresses the core step: swapping *adjacent* elements. We have initially referred to it as parametric, but in a categorical setting we will consider it natural in x. Furthermore, we will read its type as a so-called *distributive law*—it distributes the head of a list over the head of an ordered list.

Given *swap*, how do we turn it back into *bub* and *naïveIns*? For the former, we match the return type of *swap*, $\underline{\mathit{List}}\ (\mathit{List}\ (\mu\mathit{List}))$, to the return type of *bub* $\underline{\mathit{List}}\ (\mu\mathit{List})$ using *map in*. Dually, we match the input type of *naïveIns* with the input type of *swap* using *map out*. So our final sorting functions become:

$$
\begin{aligned}
&\mathit{bubbleSort}' :: \mu \mathit{List} \to \nu \underline{\mathit{List}}\\
&\mathit{bubbleSort}' = \mathit{unfold}\ (\mathit{fold}\ (\mathit{map}\ \mathit{in} \cdot \mathit{swap}))\\
&\mathit{naïveInsertSort}' :: \mu \mathit{List} \to \nu \underline{\mathit{List}}\\
&\mathit{naïveInsertSort}' = \mathit{fold}\ (\mathit{unfold}\ (\mathit{swap} \cdot \mathit{map}\ \mathit{out}))
\end{aligned}
$$

Now that we can express *bub* as $\mathit{map}\ \mathit{in} \cdot \mathit{swap}$, and *naïveIns* as $\mathit{swap} \cdot \mathit{map}\ \mathit{out}$, we would like to dig deeper into the apparent relationship between the two.

3.1 Algebra and Coalgebra Homomorphisms

Let us proceed towards this goal by first looking at a property of *bubble*. Recall that an F-algebra homomorphism between F-algebras $a :: \mathsf{F}\, A \to A$ and $b :: \mathsf{F}\, B \to B$ is a function with the property $f \cdot a = b \cdot \mathit{map}\, f$—$\mathsf{F}$-coalgebra homomorphisms have the dual property. We originally wrote *bubble* as a fold of the algebra *bub*. The following law states that *bubble* is a *List*-algebra homomorphism.

$$\mathit{bubble} \cdot \mathit{in} = \mathit{bub} \cdot \mathit{map}\ \mathit{bubble}$$

It says that *bubble* is a homomorphism from the initial algebra, *in*, to the algebra *bub*. We will render this law as a diagram, as what follows is more easily motivated in diagrammatic form.

$$
\begin{tikzcd}
\mathit{List}\ (\mu\mathit{List}) \arrow[r, "\mathit{map}\ \mathit{bubble}"] \arrow[d, "\mathit{in}"'] & \mathit{List}\ (\underline{\mathit{List}}\ (\mu\mathit{List})) \arrow[d, "\mathit{bub}"] \\
\mu\mathit{List} \arrow[r, "\mathit{bubble} = \mathit{fold}\ \mathit{bub}"'] & \underline{\mathit{List}}\ (\mu\mathit{List})
\end{tikzcd}
$$

We claimed that we can replace *bub* with $\mathit{map}\ \mathit{in} \cdot \mathit{swap}$, so let us rewrite the homomorphism law, to express the relationship between *bubble* and *swap*:

$$
\begin{aligned}
&\mathit{bubble} \cdot \mathit{in} = \mathit{bub} \cdot \mathit{map}\ \mathit{bubble}\\
\iff &\{\ \mathit{bub} \text{ is replaceable by } \mathit{map}\ \mathit{in} \cdot \mathit{swap}\ \}\\
&\mathit{bubble} \cdot \mathit{in} = \mathit{map}\ \mathit{in} \cdot \mathit{swap} \cdot \mathit{map}\ \mathit{bubble}
\end{aligned}
$$

Let us also redraw the diagram with this replacement,

$$
\begin{tikzcd}
\mathit{List}\ (\mu\mathit{List}) \arrow[r, "\mathit{map}\ \mathit{bubble}"] \arrow[ddd, "\mathit{in}"'] & \mathit{List}\ (\underline{\mathit{List}}\ (\mu\mathit{List})) \arrow[d, "\mathit{swap}"'] \\
& \underline{\mathit{List}}\ (\mathit{List}\ (\mu\mathit{List})) \arrow[d, "\mathit{map}\ \mathit{in}"'] \\
& \\
\mu\mathit{List} \arrow[r, "\mathit{bubble}"'] & \underline{\mathit{List}}\ (\mu\mathit{List})
\end{tikzcd}
$$

and then re-arrange it to better see the symmetry by moving $\underline{\mathit{List}}\ (\mu\mathit{List})$ to the left.

$$
\begin{tikzcd}
\mathit{List}\ (\mu\mathit{List}) \arrow[d, "\mathit{in}"] \arrow[dr, "\mathit{map}\ \mathit{bubble}"] & \\
\mu\mathit{List} \arrow[dd, "\mathit{bubble}"] & \mathit{List}\ (\underline{\mathit{List}}\ (\mu\mathit{List})) \arrow[d, "\mathit{swap}"] \\
& \underline{\mathit{List}}\ (\mathit{List}\ (\mu\mathit{List})) \arrow[dl, "\mathit{map}\ \mathit{in}"] \\
\underline{\mathit{List}}\ (\mu\mathit{List}) &
\end{tikzcd}
$$

Similarly, we can express the relationship between *naïveInsert* and *swap*,

$$
\begin{aligned}
&\mathit{out} \cdot \mathit{naïveInsert} = \mathit{map}\ \mathit{naïveInsert} \cdot \mathit{naïveIns}\\
\iff &\{\ \mathit{naïveIns} \text{ is replaceable by } \mathit{swap} \cdot \mathit{map}\ \mathit{out}\ \}\\
&\mathit{out} \cdot \mathit{naïveInsert} = \mathit{map}\ \mathit{naïveInsert} \cdot \mathit{swap} \cdot \mathit{map}\ \mathit{out}
\end{aligned}
$$

along with the corresponding diagram, this time jumping directly to the re-arranged variant.

$$
\begin{tikzcd}
\mathit{List}\ (\nu\underline{\mathit{List}}) \arrow[d, "\mathit{naïveInsert}"] \arrow[dr, "\mathit{map}\ \mathit{out}"] & \\
\nu\underline{\mathit{List}} \arrow[dd, "\mathit{out}"] & \mathit{List}\ (\underline{\mathit{List}}\ (\nu\underline{\mathit{List}})) \arrow[d, "\mathit{swap}"] \\
& \underline{\mathit{List}}\ (\mathit{List}\ (\nu\underline{\mathit{List}})) \arrow[dl, "\mathit{map}\ \mathit{naïveInsert}"] \\
\underline{\mathit{List}}\ (\nu\underline{\mathit{List}}) &
\end{tikzcd}
$$

Now, not only do we have a new expression of the relationships between *bubble* and *swap*, and *naïveInsert* and *swap*, but we can also begin to see the relationship between *bubble* and *naïveInsert*.

3.2 Bialgebras

We have drawn the dashed boxes to highlight the fact that these are so-called *bialgebras*: that is, an algebra *a* and a coalgebra *c*, such that we can compose them, $c \cdot a$. In the first diagram, *bubble* forms a bialgebra $\langle \mu\mathit{List}, \mathit{in}, \mathit{bubble}\rangle$, and in the second, *naïveInsert* forms $\langle \nu\mathit{List}, \mathit{naïveInsert}, \mathit{out}\rangle$. To be precise, these are *swap*-bialgebras, where the algebra and coalgebra parts are related by a distributive law, in this case, *swap*. For an algebra $a : \mathit{List}\ X \to X$ and coalgebra $c : X \to \underline{\mathit{List}}\ X$ to be a *swap*-bialgebra, we must have that

$$c \cdot a = \mathit{map}\ a \cdot \mathit{swap} \cdot \mathit{map}\ c\ \ . \tag{1}$$

This condition is exactly what we have already seen in the previous diagrams for *bubble* and *naïveInsert*.

$$
\begin{tikzcd}
\mathit{List}\ X \arrow[d, "a"] \arrow[dr, "\mathit{map}\ c"] & \\
X \arrow[dd, "c"] & \mathit{List}\ (\underline{\mathit{List}}\ X) \arrow[d, "\mathit{swap}"] \\
& \underline{\mathit{List}}\ (\mathit{List}\ X) \arrow[dl, "\mathit{map}\ a"] \\
\underline{\mathit{List}}\ X &
\end{tikzcd}
$$

We now will use the theoretical framework of bialgebras to show that bubble sort and naïve insertion sort are, categorically speaking, two sides of the same coin.

We already have an understanding of initial algebras and final coalgebras, and we will proceed by identifying the initial and final *swap*-bialgebras. Our initial *swap*-bialgebra will be $\langle \mu\mathit{List}, \mathit{in}, \mathit{fold}\ (\mathit{map}\ \mathit{in} \cdot \mathit{swap})\rangle$ and $\mathit{fold}\ a$ will be the unique *swap*-bialgebra homomorphism to any bialgebra $\langle X, a, c\rangle$. Expressed diagrammatically,

$$
\begin{tikzcd}
\mathit{List}\ (\mu\mathit{List}) \arrow[r, dashed, "\mathit{map}\ (\mathit{fold}\ a)"] \arrow[d, "\mathit{in}"'] \arrow[dr, phantom, "\dagger"] & \mathit{List}\ X \arrow[d, "a"] \\
\mu\mathit{List} \arrow[r, dashed, "\mathit{fold}\ a"] \arrow[d, "\mathit{fold}\ (\mathit{map}\ \mathit{in} \cdot \mathit{swap})"'] \arrow[dr, phantom, "\ddagger"] & X \arrow[d, "c"] \\
\underline{\mathit{List}}\ (\mu\mathit{List}) \arrow[r, dashed, "\mathit{map}\ (\mathit{fold}\ a)"'] & \underline{\mathit{List}}\ X
\end{tikzcd}
$$

There are three proof obligations that arise from this diagram. First, that $\langle \mu List, in, fold\ (map\ in \cdot swap)\rangle$ is a valid *swap*-bialgebra, but this is true by definition. Second, that the top half of the diagram (†) commutes, but this is true by construction. Third, that the bottom half of the diagram (‡) commutes:

$$map\ (fold\ a) \cdot fold\ (map\ in \cdot swap) = c \cdot fold\ a \quad .$$

Proof. We proceed by showing that both sides of the equation can be expressed as a single fold.

$$\begin{aligned}
& map\ (fold\ a) \cdot fold\ (map\ in \cdot swap) = c \cdot fold\ a \\
\Longleftrightarrow \quad & \{\ c \text{ is an algebra homomorphism, see below }\} \\
& map\ (fold\ a) \cdot fold\ (map\ in \cdot swap) = fold\ (map\ a \cdot swap)
\end{aligned}$$

This first step is justified by the law for *swap*-bialgebras (1), which states that c is an algebra homomorphism from a to $map\ a \cdot swap$:

$$c \cdot a = (map\ a \cdot swap) \cdot map\ c \quad .$$

To conclude the proof, we need to show that $map\ (fold\ a)$ is also an algebra homomorphism from $map\ in \cdot swap$ to $map\ a \cdot swap$.

$$\begin{aligned}
& map\ (fold\ a) \cdot map\ in \cdot swap \\
= \quad & \{\ map \text{ preserves composition }\} \\
& map\ (fold\ a \cdot in) \cdot swap \\
= \quad & \{\ fold\ a \text{ is a homomorphism }\} \\
& map\ (a \cdot map\ (fold\ a)) \cdot swap \\
= \quad & \{\ map \text{ preserves composition }\} \\
& map\ a \cdot map\ (map\ (fold\ a)) \cdot swap \\
= \quad & \{\ swap \text{ is natural }\} \\
& map\ a \cdot swap \cdot map\ (map\ (fold\ a))
\end{aligned}$$

Thus, *fold a* is the coalgebra homomorphism that makes the bottom half of the diagram (‡) commute. □

We have now constructed the initial *swap*-bialgebra. We can dualise this construction to obtain the final *swap*-bialgebra. We take $\langle \nu \underline{List}, unfold\ (swap \cdot map\ out), out\rangle$ to be the final *swap*-bialgebra, and *unfold c* as the unique homomorphism from any bialgebra $\langle X, a, c\rangle$. Again, that this is a valid bialgebra is by definition, and that *unfold c* is a coalgebra homomorphism is by construction. The third proof obligation, that *unfold c* is an algebra homomorphism, follows from the dual of the proof: from the naturality of *swap*, and that a is a coalgebra homomorphism.

Now that we have the theoretical framework in place, we are in a position to say something about the relationship between bubble sort and naïve insertion sort. Let us remind ourselves of their definitions.

$$\begin{aligned}
bubbleSort' &= unfold\ (fold\ \ (map\ in \cdot swap)) \\
naïveInsertSort' &= fold\ \ (unfold\ (swap \cdot map\ out))
\end{aligned}$$

We have shown that *bubble* and *naïveInsert* are the initial and final *swap*-bialgebras, respectively. Because of initiality, *fold naïveInsert* is the unique arrow from *bubble*. Dually, because of finality, the unique arrow to *naïveInsert* is *unfold bubble*.

$$\begin{tikzcd}
List\ (\mu List) \arrow[r, dashed] \arrow[d, "in"'] & List\ (\nu \underline{List}) \arrow[d, "naïveInsert"] \\
\mu List \arrow[r, dashed, "fold\ naïveInsert", "unfold\ bubble"', "\|" description] \arrow[d, "bubble"'] & \nu \underline{List} \arrow[d, "out"] \\
\underline{List}\ (\mu List) \arrow[r, dashed] & \underline{List}\ (\nu \underline{List})
\end{tikzcd}$$

By uniqueness, *bubbleSort′* and *naïveInsertSort′* are equal, and with that we have arrived at our first result.

We need to be precise about what we mean by "equal". This equality is more than merely extensional: indeed, all the sorts in this paper are extensionally equal as they correctly sort a list. Our achievement is twofold.

First, we have distilled the computational essence of insertion into a list, and selection from a list, down to a function *swap*; we read it as a distributive law that describes a single step of both of these sorting algorithms.

Second, given a distributive law such as *swap*, there are two ways to turn it into a sorting function, $\mu List \to \nu \underline{List}$: as a fold of an unfold, and an unfold of a fold. While these mundanely construct the recursive computation, this is truly where the duality arises. In the case of *swap*, the former is naïve insertion sort and the latter is bubble sort. Moreover, using the framework of bialgebras we can set up the former as the initial construction, and the latter as the final construction. There is a unique arrow from initial to final, and our two sorting algorithms are simply dual ways of describing it.

4. Para- and Apomorphisms

In the previous section, we have seen how to write insertion sort naïvely as a fold of an unfold, and bubble sort as an unfold of a fold. While simple, these algorithms are inefficient: *fold* traverses the list linearly, applying the algebra to each element. Since our algebras are *unfold*s, the running time is quadratic in length of the list. Dually, the same holds for *unfold* with coalgebras that are *fold*s.

At this stage, the novice functional programmer is typically introduced to other sorting algorithms with better asymptotic complexity. However, to write the swapping sorts that we have seen thus far in terms of primitive recursive morphisms, we need variants of our cata- and anamorphisms, namely *para-* and *apomorphisms*. Apomorphisms allow us to write more efficient coalgebras that can signal when to stop corecursion, and paramorphisms provide a way to match on the intermediate state of the list during computation.

4.1 Paramorphisms

Paramorphisms are a variation of catamorphisms—folds—where the algebra is given more information about the intermediate state of the list during the traversal. By analogy with catamorphisms, we call the argument to a paramorphism an algebra, though this is not strictly accurate. In a catamorphism, the algebra gets the current element and the result computed so far; in a paramorphism, the algebra also gets the remainder of the list. This extra parameter can be seen as a form of an *as*-pattern and is typically used to match on more than one element at a time or to detect that we have reached the final element.

For the paramorphism algebra we will need products and a *split* combinator that builds a product from two functions:

$$\begin{aligned}
&\mathbf{data}\ a \times b = As\ \{outl :: a, outr :: b\} \\
&(\triangle) :: (x \to a) \to (x \to b) \to (x \to a \times b) \\
&(f \triangle g)\ x = As\ (f\ x)\ (g\ x)
\end{aligned}$$

We will write the constructor of products, *As a b*, as $a \mathbin{\dot{=}} b$ (which should be read as the Haskell *as*-pattern: *a@b*).

We are now ready to define the paramorphism:

$$\begin{aligned}
&para\ \ :: (Functor\ f) \Rightarrow (f\ (\mu f \times a) \to a) \to (\mu f \to a) \\
&para\ f = f \cdot map\ (id \triangle para\ f) \cdot in^{\circ}
\end{aligned}$$

Note the similarity with *fold* (Section 2); the important difference is in the type of the algebra, which is now $f\ (\mu f \times a) \to a$ instead of just $f\ a \to a$. In fact, *para* can be defined directly as a fold:

$$\begin{aligned}
&para'\ \ :: (Functor\ f) \Rightarrow (f\ (\mu f \times a) \to a) \to (\mu f \to a) \\
&para'\ f = outr \cdot fold\ ((in \cdot map\ outl) \triangle f)
\end{aligned}$$

Another name often given to *para* is *recurse* (Augusteijn 1999).

4.2 Apomorphisms

Having seen how to construct the paramorphism, we now proceed to its dual: the apomorphism. Apomorphisms generalise anamorphisms—unfolds—and can be used to provide a stopping condition on production, which in turn improves the efficiency of the algorithm. Also by analogy, we will refer to argument to an apomorphism as a coalgebra.

For defining apomorphisms we will need a disjoint union type and a combinator that destructs a sum using two functions, implementing a case analysis:

$$\begin{aligned}
&\mathbf{data}\ a + b = \mathit{Stop}\ a \mid \mathit{Play}\ b \\
&(\triangledown) :: (a \to x) \to (b \to x) \to (a + b \to x) \\
&(f \triangledown g)\ (\mathit{Stop}\ a) = f\ a \\
&(f \triangledown g)\ (\mathit{Play}\ b) = g\ b
\end{aligned}$$

We name the constructors of $+$, *Stop* and *Play*, alluding to their behaviour in the context of a coalgebra. We write *Stop a* as $a\,\blacksquare$, and *Play b* as $\blacktriangleright b$.

We are now ready to define the apomorphism:

$$\begin{aligned}
&\mathit{apo} \ :: (\mathit{Functor}\ f) \Rightarrow (a \to f\ (\nu f + a)) \to (a \to \nu f) \\
&\mathit{apo}\ f = \mathit{out}^{\circ} \cdot \mathit{map}\ (\mathit{id} \triangledown \mathit{apo}\ f) \cdot f
\end{aligned}$$

As expected, *apo* is similar to *unfold*, but the corecursion is split into two branches, with no recursive call on the left. Another name often given to *apo* is *corecurse*.

Apomorphisms can also be defined in terms of unfolds. However, this is *not* as efficient: recursion continues in $\blacksquare$ mode, copying the data step-by-step:

$$\begin{aligned}
&\mathit{apo}' \ :: (\mathit{Functor}\ f) \Rightarrow (a \to f\ (\nu f + a)) \to (a \to \nu f) \\
&\mathit{apo}'\ f = \mathit{unfold}\ ((\mathit{map}\ (\blacksquare) \cdot \mathit{out}) \triangledown f) \cdot \blacktriangleright
\end{aligned}$$

At the end of Section 2, we followed a type-directed approach to derive the types of an algebra a and a coalgebra c in the terms $\mathit{fold}\ (\mathit{unfold}\ c)$ and $\mathit{unfold}\ (\mathit{fold}\ a)$. We will now repeat this exercise for a and c, but this time with the terms $\mathit{fold}\ (\mathit{apo}\ c)$ and $\mathit{unfold}\ (\mathit{para}\ a)$.

$$\begin{aligned}
\mathit{fold}\ (\mathit{apo}\ c) &: \mu\mathsf{F} \to \nu\mathsf{G} \\
\mathit{apo}\ c &: \mathsf{F}\ (\nu\mathsf{G}) \to \nu\mathsf{G} \\
c &: \mathsf{F}\ (\nu\mathsf{G}) \to \mathsf{G}\ (\nu\mathsf{G} + \mathsf{F}\ (\nu\mathsf{G})) \\
&\cong \mathsf{F}\ (\mathsf{G}\ (\nu\mathsf{G})) \to \mathsf{G}\ (\mathsf{F}_{+}\ (\nu\mathsf{G}))
\end{aligned}$$

$$\begin{aligned}
\mathit{unfold}\ (\mathit{para}\ a) &: \mu\mathsf{F} \to \nu\mathsf{G} \\
\mathit{para}\ a &: \mu\mathsf{F} \to \mathsf{G}\ (\mu\mathsf{F}) \\
a &: \mathsf{F}\ (\mu\mathsf{F} \times \mathsf{G}\ (\mu\mathsf{F})) \to \mathsf{G}\ (\mu\mathsf{F}) \\
&\cong \mathsf{F}\ (\mathsf{G}_{\times}\ (\mu\mathsf{F})) \to \mathsf{G}\ (\mathsf{F}\ (\mu\mathsf{F}))
\end{aligned}$$

By introducing the types,

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

we can see that the algebra a and a coalgebra c are still closely related. While the correspondence is no longer as obvious as in Section 2, we will see that we can describe both a and c in terms of a natural transformation of type $\mathsf{F} \circ \mathsf{G}_{\times} \mathrel{\dot{\to}} \mathsf{G} \circ \mathsf{F}_{+}$.

An obvious question is why we do not use a *para* of an *apo*, or an *apo* of a *para*. The answer is simply that we lose the relationship between a and c: we cannot construct a natural transformation that relates the two.

$$\begin{aligned}
\mathit{para}\ (\mathit{apo}\ c) &: \mu\mathsf{F} \to \nu\mathsf{G} \\
\mathit{apo}\ c &: \mathsf{F}\ (\mu\mathsf{F} \times \nu\mathsf{G}) \to \nu\mathsf{G} \\
c &: \mathsf{F}\ (\mu\mathsf{F} \times \nu\mathsf{G}) \to \mathsf{G}\ (\nu\mathsf{G} + \mathsf{F}\ (\mu\mathsf{F} \times \nu\mathsf{G}))
\end{aligned}$$

$$\begin{aligned}
\mathit{apo}\ (\mathit{para}\ a) &: \mu\mathsf{F} \to \nu\mathsf{G} \\
\mathit{para}\ a &: \mu\mathsf{F} \to \mathsf{G}\ (\nu\mathsf{G} + \mu\mathsf{F}) \\
a &: \mathsf{F}\ (\mu\mathsf{F} \times \mathsf{G}\ (\nu\mathsf{G} + \mu\mathsf{F})) \to \mathsf{G}\ (\nu\mathsf{G} + \mu\mathsf{F})
\end{aligned}$$

While expressive, these types are not useful to us.

5. Insertion and Selection Sort

The naïve insertion sort presented in Section 3 is unable to leverage the fact that the list being inserted into is already sorted, and so it continues to scan through the list, even after the element to insert has been placed appropriately. Now that we have apomorphisms, however, we can write the insertion function as one that avoids scanning needlessly:

$$\begin{aligned}
&\mathit{insertSort} :: \mu\mathit{List} \to \nu\underline{\mathit{List}} \\
&\mathit{insertSort} = \mathit{fold}\ \mathit{insert} \\
&\quad \mathbf{where}\ \mathit{insert} = \mathit{apo}\ \mathit{ins}
\end{aligned}$$

The coalgebra *ins* is now enriched with the ability to signal that the recursion should stop.

$$\begin{aligned}
&\mathit{ins} :: \mathit{List}\ (\nu\underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\nu\underline{\mathit{List}} + \mathit{List}\ (\nu\underline{\mathit{List}})) \\
&\mathit{ins}\ \mathit{Nil} && = \underline{\mathit{Nil}} \\
&\mathit{ins}\ (\mathit{Cons}\ a\ \lfloor\underline{\mathit{Nil}}\rfloor) && = \underline{\mathit{Cons}}\ a\ (\lfloor\underline{\mathit{Nil}}\rfloor\,\blacksquare) \\
&\mathit{ins}\ (\mathit{Cons}\ a\ \lfloor\underline{\mathit{Cons}}\ b\ x'\rfloor) \\
&\quad \mid a \leqslant b && = \underline{\mathit{Cons}}\ a\ (\lfloor\underline{\mathit{Cons}}\ b\ x'\rfloor\,\blacksquare) \\
&\quad \mid \mathit{otherwise} && = \underline{\mathit{Cons}}\ b\ (\blacktriangleright(\mathit{Cons}\ a\ x'))
\end{aligned}$$

This signal appears in the definition of the third case, where the element to insert, a, is ordered with respect to the head of the already ordered list, so there is no more work to be done. The stop signal is also used in the second case, where the list to insert into is empty. We could have given the following as an alternative definition for this case:

$$\mathit{ins}\ (\mathit{Cons}\ a\ \lfloor\underline{\mathit{Nil}}\rfloor) = \underline{\mathit{Cons}}\ a\ (\blacktriangleright \mathit{Nil})$$

While still correct, the apomorphism would take one more step, to turn $\blacktriangleright \mathit{Nil}$ into $\underline{\mathit{Nil}}$. With or without the superfluous step, *insertSort* will run in linear time on a list that is already sorted; this is in contrast to *naïveInsertSort*, *bubbleSort*, and selection sort, which we will define shortly. All of these will still run in quadratic time, as they cannot abort their traversals. Early termination in apomorphisms avoids *redundant* comparisons and is the key to *insertSort*'s best and average case behaviour.

We can extract a new natural transformation from *ins*. In Section 3 we called the natural transformation for swapping sorts, *swap*; we will call our enriched version *swop*, for *swap'n'stop*.

$$\begin{aligned}
&\mathit{swop} :: \mathit{List}\ (x \times \underline{\mathit{List}}\ x) \to \underline{\mathit{List}}\ (x + \mathit{List}\ x) \\
&\mathit{swop}\ \mathit{Nil} && = \underline{\mathit{Nil}} \\
&\mathit{swop}\ (\mathit{Cons}\ a\ (x \asymp \underline{\mathit{Nil}})) && = \underline{\mathit{Cons}}\ a\ (x\,\blacksquare) \\
&\mathit{swop}\ (\mathit{Cons}\ a\ (x \asymp (\underline{\mathit{Cons}}\ b\ x'))) \\
&\quad \mid a \leqslant b && = \underline{\mathit{Cons}}\ a\ (x\,\blacksquare) \\
&\quad \mid \mathit{otherwise} && = \underline{\mathit{Cons}}\ b\ (\blacktriangleright(\mathit{Cons}\ a\ x'))
\end{aligned}$$

The type makes it clear that $x \asymp \_$ and $x\,\blacksquare$ really go hand-in-hand.

Before, we had a natural transformation, $\mathit{List} \circ \underline{\mathit{List}} \mathrel{\dot{\to}} \underline{\mathit{List}} \circ \mathit{List}$; now we have one with type, $\mathit{List} \circ \underline{\mathit{List}}_{\times} \mathrel{\dot{\to}} \underline{\mathit{List}} \circ \mathit{List}_{+}$. In Section 3 we saw a diagram that described the relationship between *naïveInsert* and *swap*; contrast this with the relationship between *insert* and *swop* in the following diagram.

$$\begin{tikzcd}
List\ (\nu\underline{List}) \arrow[d, "insert"'] \arrow[r, "map\ (id \vartriangle out)"] & List\ (\underline{List}_{\times}\ (\nu\underline{List})) \arrow[d, "swop"] \\
\nu\underline{List} \arrow[d, "out"'] & \underline{List}\ (List_{+}\ (\nu\underline{List})) \arrow[dl, "map\ (id \triangledown insert)"] \\
\underline{List}\ (\nu\underline{List}) &
\end{tikzcd}$$

Note that this diagram is not symmetric in the way that the diagrams were in Section 3: for example, *out* is matched with *map* $(id \vartriangle out)$, rather than *map out*. This is because *swop* itself is not symmetric. In Appendix A we briefly sketch how *swop* can be turned into a distributive law of type $List_{+} \circ \underline{List}_{\times} \dot{\to} \underline{List}_{\times} \circ List_{+}$. This distributive law is unneeded here, as we will write *insert* directly in terms of *swop* using an apomorphism. (The proof of why this is the case is, again, in Appendix A.) As in Section 3, we can also dualise this development. Just as naïve insertion as an unfold was dual to bubble as a fold, insertion as an apomorphism can be dualised to selection as a paramorphism.

$$\begin{aligned}
&selectSort :: \mu List \to \nu\underline{List} \\
&selectSort = unfold\ select \\
&\quad \mathbf{where}\ select = para\ sel
\end{aligned}$$

$$\begin{aligned}
&sel :: List\ (\mu List \times \underline{List}\ (\mu List)) \to \underline{List}\ (\mu List) \\
&sel\ Nil && = \underline{Nil} \\
&sel\ (Cons\ a\ (x \mathrel{\text{⩦}} \underline{Nil})) && = \underline{Cons}\ a\ x \\
&sel\ (Cons\ a\ (x \mathrel{\text{⩦}} (\underline{Cons}\ b\ x'))) && \\
&\quad \mid a \leqslant b && = \underline{Cons}\ a\ x \\
&\quad \mid otherwise && = \underline{Cons}\ b\ \lceil Cons\ a\ x' \rceil
\end{aligned}$$

The sole difference between *sel* and *bub* (Section 3) is in the case where $a \leqslant b$: *sel* uses the remainder of the list, supplied by the paramorphism, rather than the result computed so far. This is why *para sel* is the true selection function, and *fold bub* is the naïve variant, if you will.

To conclude our discussion, we have new definitions of insertion and selection sort in terms of our new natural transformation, *swop*.

$$\begin{aligned}
&insertSort' :: \mu List \to \nu\underline{List} \\
&insertSort' = fold\ insert \\
&\quad \mathbf{where}\ insert = apo\ (swop \cdot map\ (id \vartriangle out))
\end{aligned}$$

$$\begin{aligned}
&selectSort' :: \mu List \to \nu\underline{List} \\
&selectSort' = unfold\ select \\
&\quad \mathbf{where}\ select = para\ (map\ (id \triangledown in) \cdot swop)
\end{aligned}$$

We shall omit the proofs that *select* and *insert* form initial and final bialgebras, respectively; the details are lengthy and beyond the scope of this paper, see Hinze and James (2011). Instead we shall simply give the diagram that states them.

$$\begin{tikzcd}
List_{+}\ (\mu List) \arrow[r, dashed] \arrow[d, "id \triangledown in"'] & List_{+}\ (\nu\underline{List}) \arrow[d, "id \triangledown insert"] \\
\mu List \arrow[r, dashed, "fold\ insert", "unfold\ select"', "\|" description] \arrow[d, "id \vartriangle select"'] & \nu\underline{List} \arrow[d, "id \vartriangle out"] \\
\underline{List}_{\times}\ (\mu List) \arrow[r, dashed] & \underline{List}_{\times}\ (\nu\underline{List})
\end{tikzcd}$$

Thus, *insertSort′* and *selectSort′* are dual; moreover, by uniqueness, they are equal.

6. Quicksort and Treesort

While the reader should not have expected better, the results of Section 5 are still somewhat disappointing; apomorphisms have helped implement a small optimisation, but the worst case running time is still quadratic. This arises from the use of swaps in both selection and insertion sort—they are fundamentally bound by the linear nature of lists. If we are to do better than a quadratic bound, we need sublinear insertion and selection operations. To use such operations, we must introduce an intermediate data structure that supports them. We do this by moving to a two-phase algorithm, where the first phase builds such an intermediate data structure from a list, and the second phase consumes it to produce a sorted list. In this section, we seek a better sorting algorithm by using binary trees with elements of type K.

$$\begin{aligned}
&\mathbf{data}\ Tree\ tree = Empty \mid Node\ tree\ K\ tree \\
&\mathbf{instance}\ Functor\ Tree\ \mathbf{where} \\
&\quad map\ f\ Empty && = Empty \\
&\quad map\ f\ (Node\ l\ k\ r) && = Node\ (f\ l)\ k\ (f\ r)
\end{aligned}$$

Henceforth, we will write *Empty* as ε and *Node l k r* as $l \mathbin{/\!\!/} k \mathbin{\backslash\!\!\backslash} r$. In this section, we will be using the tree type as a search tree,

$$\mathbf{type}\ SearchTree = Tree$$

where all the values in the left subtree of a node are less than or equal to the value at the node, and all values in the right subtree are greater. Such a tree orders the elements horizontally, such that an in-order traversal of the tree yields a sorted list.

6.1 Phase One: Growing Search Trees

First, we start with the unfold of a fold approach. Therefore, we seek a fold that produces something of type *SearchTree* $(\mu List)$. The idea is that the fold will create one level of the tree, where $l \mathbin{/\!\!/} k \mathbin{\backslash\!\!\backslash} r$ contains a value k for which values in the list l are less than or equal to k, and values in the list r are greater than k. In other words, k acts as a pivot around which l and r are partitioned.

$$\begin{aligned}
&pivot :: List\ (SearchTree\ (\mu List)) \to SearchTree\ (\mu List) \\
&pivot\ Nil && = \varepsilon \\
&pivot\ (Cons\ a\ \varepsilon) && = \lceil Nil \rceil \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} \lceil Nil \rceil \\
&pivot\ (Cons\ a\ (l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r)) && \\
&\quad \mid a \leqslant b && = \lceil Cons\ a\ l \rceil \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r \\
&\quad \mid otherwise && = l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} \lceil Cons\ a\ r \rceil
\end{aligned}$$

In effect, *fold pivot* $:: \mu List \to SearchTree\ (\mu List)$ is a partitioning function that takes a list and returns its last element as a pivot, along with the two partitions of that list. At each step, the enclosing unfold will call this fold on each of the resulting partitions, which will ultimately yield the entire search tree.

The type of *pivot* gives us little choice in its implementation; *Nil* will be replaced with ε, *Cons a* ε will become a tree of one element, with empty lists as children. Therefore, the construction of l and r is determined by value of the pivot, the last element.

As we have done before, we shall extract a natural transformation from this algebra.

$$\begin{aligned}
&sprout :: List\ (x \times SearchTree\ x) \to SearchTree\ (x + List\ x) \\
&sprout\ Nil && = \varepsilon \\
&sprout\ (Cons\ a\ (t \mathrel{\text{⩦}} \varepsilon)) && = (t \blacksquare) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (t \blacksquare) \\
&sprout\ (Cons\ a\ (t \mathrel{\text{⩦}} (l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r))) && \\
&\quad \mid a \leqslant b && = (\blacktriangleright (Cons\ a\ l)) \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} (r \blacksquare) \\
&\quad \mid otherwise && = (l \blacksquare) \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} (\blacktriangleright (Cons\ a\ r))
\end{aligned}$$

In Sections 3 and 5, we were operating with lists and swapped the elements to maintain the ordering. With trees, we must maintain the search tree property when inserting elements.

Having extracted the natural transformation, we can synthesise the coalgebra that is dual to *pivot*,

$$\begin{aligned}
&\mathit{treeIns} :: \mathit{List}\ (\nu \mathit{SearchTree}) \\
&\qquad \to \mathit{SearchTree}\ (\nu \mathit{SearchTree} + \mathit{List}\ (\nu \mathit{SearchTree})) \\
&\mathit{treeIns}\ \mathit{Nil} && = \varepsilon \\
&\mathit{treeIns}\ (\mathit{Cons}\ a\ \lfloor \varepsilon \rfloor) && = (\lfloor \varepsilon \rfloor \blacksquare) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (\lfloor \varepsilon \rfloor \blacksquare) \\
&\mathit{treeIns}\ (\mathit{Cons}\ a\ \lfloor l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r \rfloor) \\
&\quad \mid a \leqslant b && = (\blacktriangleright (\mathit{Cons}\ a\ l)) \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} (r \blacksquare) \\
&\quad \mid \mathit{otherwise} && = (l \blacksquare) \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} (\blacktriangleright (\mathit{Cons}\ a\ r))
\end{aligned}$$

which takes an element of the input list and inserts it one level deep into a search tree. We shall call this *treeIns*, since, as we shall see, this algebra forms the first phase of a treesort. *Efficient insertion into a tree is necessarily an apomorphism*; because of the search tree property, the recursion need only go down one of the branches, which is not possible with an unfold.

The derivation of *treeIns* merits some review. We began this section by writing a function to partition a list around a pivot. Then, we turned this into a natural transformation. Now, out the other side, so to speak, we have another useful function, which inserts an element into a search tree: $\mathit{apo}\ \mathit{treeIns} :: \mathit{List}\ (\nu \mathit{SearchTree}) \to \nu \mathit{SearchTree}$. Moreover, we got this for free.

As before, the algebra and coalgebra can be written in terms of the natural transformation, so $\mathit{pivot} = \mathit{map}\ (\mathit{id} \triangledown \mathit{in}) \cdot \mathit{sprout}$ and $\mathit{treeIns} = \mathit{sprout} \cdot \mathit{map}\ (\mathit{id} \vartriangle \mathit{out})$. This yields two algorithms for generating search trees:

$$\begin{aligned}
&\mathit{grow}, \mathit{grow}' :: \mu \mathit{List} \to \nu \mathit{SearchTree} \\
&\mathit{grow}\ \ = \mathit{unfold}\ (\mathit{para}\ (\mathit{map}\ (\mathit{id} \triangledown \mathit{in}) \cdot \mathit{sprout})) \\
&\mathit{grow}' = \mathit{fold}\ (\mathit{apo}\ (\mathit{sprout} \cdot \mathit{map}\ (\mathit{id} \vartriangle \mathit{out})))
\end{aligned}$$

We can either recursively partition a list, building subtrees from the resulting sublists, or start with an empty tree and repeatedly insert the elements into it.

6.2 Phase Two: Withering Search Trees

The previous section was concerned with growing search trees. With these in place, we will now look at ways of flattening these trees into a sorted lists.

We will start with the complement to *pivot*, which partitioned a list around a pivot. Here, we need a $\underline{\mathit{List}}$-coalgebra to glue the partitions back together. More specifically, we need a coalgebra for an apomorphism, so that we can signal when to stop.

$$\begin{aligned}
&\mathit{glue} :: \mathit{SearchTree}\ (\nu \underline{\mathit{List}}) \\
&\qquad \to \underline{\mathit{List}}\ (\nu \underline{\mathit{List}} + \mathit{SearchTree}\ (\nu \underline{\mathit{List}})) \\
&\mathit{glue}\ \varepsilon && = \underline{\mathit{Nil}} \\
&\mathit{glue}\ (\lfloor \underline{\mathit{Nil}} \rfloor \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} r) && = \underline{\mathit{Cons}}\ a\ (r \blacksquare) \\
&\mathit{glue}\ (\lfloor \underline{\mathit{Cons}}\ b\ l \rfloor \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} r) && = \underline{\mathit{Cons}}\ b\ (\blacktriangleright (l \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} r))
\end{aligned}$$

Note that $\mathit{apo}\ \mathit{glue} :: \mathit{SearchTree}\ (\nu \underline{\mathit{List}}) \to \nu \underline{\mathit{List}}$ is essentially a ternary version of append: it takes a left and a right sorted list, an element in the middle, and glues it all together. Had we implemented this naïvely as a plain unfold, the right list would also have to be traversed and thus induce unnecessary copying.

Following our established course, we can extract the natural transformation from this coalgebra,

$$\begin{aligned}
&\mathit{wither} :: \mathit{SearchTree}\ (x \times \underline{\mathit{List}}\ x) \to \underline{\mathit{List}}\ (x + \mathit{SearchTree}\ x) \\
&\mathit{wither}\ \varepsilon && = \underline{\mathit{Nil}} \\
&\mathit{wither}\ ((l \precsim \underline{\mathit{Nil}}) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \precsim _)) && = \underline{\mathit{Cons}}\ a\ (r \blacksquare) \\
&\mathit{wither}\ ((l \precsim (\underline{\mathit{Cons}}\ b\ l')) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \precsim _)) && = \underline{\mathit{Cons}}\ b\ (\blacktriangleright (l' \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} r))
\end{aligned}$$

which captures the notion of flattening by traversing a tree and collecting the elements in a list. Specifically, this function returns the leftmost element, along with the combination of the remainder.

We can now synthesise the algebra that is dual to *glue*.

$$\begin{aligned}
&\mathit{shear} :: \mathit{SearchTree}\ (\mu \mathit{SearchTree} \times \underline{\mathit{List}}\ (\mu \mathit{SearchTree})) \\
&\qquad \to \underline{\mathit{List}}\ (\mu \mathit{SearchTree}) \\
&\mathit{shear}\ \varepsilon && = \underline{\mathit{Nil}} \\
&\mathit{shear}\ ((l \precsim \underline{\mathit{Nil}}) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \precsim _)) && = \underline{\mathit{Cons}}\ a\ r \\
&\mathit{shear}\ ((l \precsim (\underline{\mathit{Cons}}\ b\ l')) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \precsim _)) && = \underline{\mathit{Cons}}\ b\ \lceil l' \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} r \rceil
\end{aligned}$$

To understand what is in our hands, let us look at the third case: a is the root of the tree, with l and r as the left and right subtrees; b is the minimum of the left subtree and l' the remainder of that tree without b. In which case, $\mathit{para}\ \mathit{shear} :: \mu \mathit{SearchTree} \to \underline{\mathit{List}}\ (\mu \mathit{SearchTree})$ is the function that deletes the minimum element from a search tree. Thus, the *fold* of this flattens a tree by removing the elements in order. This should surprise no one: the second phase of treesort would surely be an in-order traversal.

We can again define both the algebra and the coalgebra in terms of the natural transformation, which yields two algorithms for flattening a tree to a list:

$$\begin{aligned}
&\mathit{flatten}, \mathit{flatten}' :: \mu \mathit{SearchTree} \to \nu \underline{\mathit{List}} \\
&\mathit{flatten} = \mathit{fold}\ (\mathit{apo}\ (\mathit{wither} \cdot \mathit{map}\ (\mathit{id} \vartriangle \mathit{out}))) \\
&\mathit{flatten}' = \mathit{unfold}\ (\mathit{para}\ (\mathit{map}\ (\mathit{id} \triangledown \mathit{in}) \cdot \mathit{wither}))
\end{aligned}$$

6.3 Putting Things Together

We have now constructed the constituent parts of the famous quicksort and the less prominent treesort algorithms. The components for quicksort dualised to give us those for treesort, and now all that remains is to assemble the respective phases together.

Quicksort works by partitioning a list based on comparison around a pivot, and then recursively descending into the resulting sublists until it only has singletons left. This is precisely the algorithm used to create the tree in *grow*, and we have simply stored the result of this recursive descent in an intermediate data structure. The *flatten* then reassembles the lists by appending the singletons together, now in sorted order, and continuing up the tree to append sorted sublists together to form the final sorted list.

Dually, treesort starts with an empty tree and builds a search tree by inserting the elements of the input list into it, which is the action of grow'. The sorted list is then obtained by pulling the elements out of the tree in order and collecting them in a list, which is how $\mathit{flatten}'$ produces a list. In each, tying the two phases together is *downcast*, which is necessary because *grow* and grow' *produce* trees, but *flatten* and $\mathit{flatten}'$ *consume* them.

$$\begin{aligned}
&\mathit{quickSort}, \mathit{treeSort} :: \mu \mathit{List} \to \nu \underline{\mathit{List}} \\
&\mathit{quickSort} = \mathit{flatten} \cdot \mathit{downcast} \cdot \mathit{grow} \\
&\mathit{treeSort}\ \ \ = \mathit{flatten}' \cdot \mathit{downcast} \cdot \mathit{grow}'
\end{aligned}$$

In the average case, quicksort and treesort run in linearithmic time. But, we have not succeeded in eliminating a quadratic running time in the worst case. We are not yet done.

7. Heapsort

Quicksort and treesort are *sensitive* to their input. Imposing a horizontal (total) ordering to the tree offers us no flexibility in how to arrange the elements, thus an unfavourably ordered input list leads to an unbalanced tree and linear, rather than logarithmic, operations. (Of course, we could use some balancing scheme.) For this section we will use *Heap* as our intermediate data structure,

$$\textbf{type}\ \mathit{Heap} = \mathit{Tree}$$

where the element of a tree node in a heap is less than or equal to all the elements of the subtrees. This heap property requires that trees are *vertically* ordered—a more 'flexible', partial order.

7.1 Phase One: Piling up a Heap

Now that we are accustomed to the natural transformations that describe the single steps of our sorting algorithms, in this section we will write them first; then we will derive the algebra and coalgebra that make up the final and initial bialgebras, respectively.

The type of our natural transformation, which we will call *pile*, will be the same as *sprout* in Section 6.1, modulo type synonyms. However, rather than its implementation being dictated by the search tree property, we have a choice to make for *pile*.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{pile} :: \mathit{List}\ (x \times \mathit{Heap}\ x) \to \mathit{Heap}\ (x + \mathit{List}\ x)} \\
\mathit{pile}\ \mathit{Nil} & = & \varepsilon \\
\mathit{pile}\ (\mathit{Cons}\ a\ (t \asymp \varepsilon)) & = & (t\,\blacksquare) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (t\,\blacksquare) \\
\multicolumn{3}{l}{\mathit{pile}\ (\mathit{Cons}\ a\ (t \asymp (l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r)))} \\
\quad \mid a \leqslant b & = & (\blacktriangleright (\mathit{Cons}\ b\ r)) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (l\,\blacksquare) \\
\quad \mid \mathit{otherwise} & = & (\blacktriangleright (\mathit{Cons}\ a\ r)) \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} (l\,\blacksquare)
\end{array}
$$

There is no choice in the first two cases, it is solely in the third case, which we will now examine. We can avoid the guards if we use *min* and *max*—this rewrite emphasises that the structure does not depend on the input data. We write *min* as $\sqcap$, and *max* as $\sqcup$, so the third case is now rendered as:

$$
\begin{array}{l}
\mathit{pile}\ (\mathit{Cons}\ a\ (t \asymp (l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r))) \\
\quad = (\blacktriangleright (\mathit{Cons}\ (a \sqcup b)\ r)) \mathbin{/\!\!/} (a \sqcap b) \mathbin{\backslash\!\!\backslash} (l\,\blacksquare)
\end{array}
$$

We actually have a choice between four different steps: adding the maximum to the left or to the right, and swapping or not swapping the results (the subtrees of a heap are, in a sense, unordered).

$$
\begin{array}{l}
\mathit{pile}\ (\mathit{Cons}\ a\ (t \asymp (l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r))) \\
\quad = (\blacktriangleright (\mathit{Cons}\ (a \sqcup b)\ l)) \mathbin{/\!\!/} (a \sqcap b) \mathbin{\backslash\!\!\backslash} (r\,\blacksquare) \\
\quad = (r\,\blacksquare) \mathbin{/\!\!/} (a \sqcap b) \mathbin{\backslash\!\!\backslash} (\blacktriangleright (\mathit{Cons}\ (a \sqcup b)\ l)) \\
\quad = (l\,\blacksquare) \mathbin{/\!\!/} (a \sqcap b) \mathbin{\backslash\!\!\backslash} (\blacktriangleright (\mathit{Cons}\ (a \sqcup b)\ r)) \\
\quad = (\blacktriangleright (\mathit{Cons}\ (a \sqcup b)\ r)) \mathbin{/\!\!/} (a \sqcap b) \mathbin{\backslash\!\!\backslash} (l\,\blacksquare)
\end{array}
$$

We chose the last option: we always add to the right and then swap left with right. By doing so, we will end up building a heap that is a *Braun tree* (Braun and Rem 1983), where a node's right subtree has, at most, one element less than its left. Thus we ensure that our heapsort is *insensitive* to the input, in contrast to quick (tree) sort.

Now that we have our natural transformation in place, it is routine to turn it into a *List*-algebra and *Heap*-coalgebra. We will start with the latter, as this will be the expected function for heapsort.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{heapIns} :: \mathit{List}\ (\nu\mathit{Heap})} \\
\multicolumn{3}{l}{\qquad\qquad \to \mathit{Heap}\ (\nu\mathit{Heap} + \mathit{List}\ (\nu\mathit{Heap}))} \\
\mathit{heapIns}\ \mathit{Nil} & = & \varepsilon \\
\mathit{heapIns}\ (\mathit{Cons}\ a\ \lfloor \varepsilon \rfloor) & = & (\lfloor \varepsilon \rfloor\,\blacksquare) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (\lfloor \varepsilon \rfloor\,\blacksquare) \\
\multicolumn{3}{l}{\mathit{heapIns}\ (\mathit{Cons}\ a\ \lfloor l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r \rfloor)} \\
\quad \mid a \leqslant b & = & (\blacktriangleright (\mathit{Cons}\ b\ r)) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (l\,\blacksquare) \\
\quad \mid \mathit{otherwise} & = & (\blacktriangleright (\mathit{Cons}\ a\ r)) \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} (l\,\blacksquare)
\end{array}
$$

We have called it *heapIns* as $\mathit{apo}\ \mathit{heapIns} :: \mathit{List}\ (\nu\mathit{Heap}) \to \nu\mathit{Heap}$ is the heap insertion function. Thus a *fold* of an *apo* will build a heap by repeated insertion. (It is instructive to compare *heapIns* to *treeIns* in Section 6.1.)

As an aside, we can actually do slightly better: sinking the element, b, into the right subheap, r, does not require any comparisons as the heap property ensures that b is smaller than the elements in r. One solution would be to introduce a variant of lists, List', with a third constructor $\mathit{Cons}^{\leqslant}$, to signal when no more comparisons are needed. We can then write $\mathit{fold}\ (\mathit{apo}\ \mathit{heapIns}') \cdot \mathit{toList}'$, where,

$$
\begin{array}{l}
\mathit{heapIns}'\ (\mathit{Cons}\ a\ \lfloor l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r \rfloor) \\
\quad \mid a \leqslant b = (\blacktriangleright (\mathit{Cons}^{\leqslant}\ b\ r)) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (l\,\blacksquare) \\
\quad \ldots \\
\mathit{heapIns}'\ (\mathit{Cons}^{\leqslant}\ a\ \lfloor l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r \rfloor) \\
\qquad = (\blacktriangleright (\mathit{Cons}^{\leqslant}\ b\ r)) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (l\,\blacksquare)
\end{array}
$$

All that is left is to examine the *List*-algebra that arises from *pile*. It is related to the *pivot* function in Section 6.1. There, we were building two lists partitioned around a pivot, but here we are selecting the least element and collecting the rest into two lists. We shall name the synthesised algebra *divvy*, meaning to divide up.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{divvy} :: \mathit{List}\ (\mathit{Heap}\ (\mu\mathit{List})) \to \mathit{Heap}\ (\mu\mathit{List})} \\
\mathit{divvy}\ \mathit{Nil} & = & \varepsilon \\
\mathit{divvy}\ (\mathit{Cons}\ a\ \varepsilon) & = & \lceil \mathit{Nil} \rceil \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} \lceil \mathit{Nil} \rceil \\
\multicolumn{3}{l}{\mathit{divvy}\ (\mathit{Cons}\ a\ (l \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r))} \\
\quad \mid a \leqslant b & = & \lceil \mathit{Cons}\ b\ r \rceil \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} l \\
\quad \mid \mathit{otherwise} & = & \lceil \mathit{Cons}\ a\ r \rceil \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} l
\end{array}
$$

The function $\mathit{fold}\ \mathit{divvy} :: \mu\mathit{List} \to \mathit{Heap}\ (\mu\mathit{List})$, selects the least element and divides the remaining list into two parts of balanced length (using Braun's trick). The unfold of *divvy* constructs a heap by repeated selection, rather than by repeated insertion. This is rather reminiscent of selection and insertion sort, and is an intriguing variant on building a heap.

7.2 Phase Two: Sifting through a Heap

Our natural transformation for describing one step of turning a heap into a list will take an interesting divergence from *wither* in Section 6.2. There, *wither* described one step of an in-order traversal. The search tree property provided the correct ordering for the output list, so no further comparisons were needed. The choice afforded to us by the heap property in Section 7.1 now means that further comparisons *are* needed, to obtain a sorted list.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{sift} :: \mathit{Heap}\ (x \times \underline{\mathit{List}}\ x) \to \underline{\mathit{List}}\ (x + \mathit{Heap}\ x)} \\
\mathit{sift}\ \varepsilon & = & \underline{\mathit{Nil}} \\
\mathit{sift}\ ((l \asymp \underline{\mathit{Nil}}) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \asymp \_)) & = & \underline{\mathit{Cons}}\ a\ (r\,\blacksquare) \\
\mathit{sift}\ ((l \asymp \_) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \asymp \underline{\mathit{Nil}})) & = & \underline{\mathit{Cons}}\ a\ (l\,\blacksquare) \\
\multicolumn{3}{l}{\mathit{sift}\ ((l \asymp (\underline{\mathit{Cons}}\ b\ l')) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \asymp (\underline{\mathit{Cons}}\ c\ r')))} \\
\quad \mid b \leqslant c & = & \underline{\mathit{Cons}}\ a\ (\blacktriangleright (l' \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r)) \\
\quad \mid \mathit{otherwise} & = & \underline{\mathit{Cons}}\ a\ (\blacktriangleright (l \mathbin{/\!\!/} c \mathbin{\backslash\!\!\backslash} r'))
\end{array}
$$

The fourth case is where these comparisons must be made: we need to pick the next minimum element from the left or the right. When constructing the heap node to continue with, we have the option to swap left with right, but this buys us nothing.

Once again, we can routinely turn our natural transformation into a *Heap*-algebra and $\underline{\mathit{List}}$-coalgebra. This time we will start with the former as this is the algebra that matches the *Heap*-coalgebra, *heapIns*, that performs heap insertion.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{meld} :: \mathit{Heap}\ (\mu\mathit{Heap} \times \underline{\mathit{List}}\ (\mu\mathit{Heap}))} \\
\multicolumn{3}{l}{\qquad \to \underline{\mathit{List}}\ (\mu\mathit{Heap})} \\
\mathit{meld}\ \varepsilon & = & \underline{\mathit{Nil}} \\
\mathit{meld}\ ((l \asymp \underline{\mathit{Nil}}) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \asymp \_)) & = & \underline{\mathit{Cons}}\ a\ r \\
\mathit{meld}\ ((l \asymp \_) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \asymp \underline{\mathit{Nil}})) & = & \underline{\mathit{Cons}}\ a\ l \\
\multicolumn{3}{l}{\mathit{meld}\ ((l \asymp (\underline{\mathit{Cons}}\ b\ l')) \mathbin{/\!\!/} a \mathbin{\backslash\!\!\backslash} (r \asymp (\underline{\mathit{Cons}}\ c\ r')))} \\
\quad \mid b \leqslant c & = & \underline{\mathit{Cons}}\ a\ \lceil l' \mathbin{/\!\!/} b \mathbin{\backslash\!\!\backslash} r \rceil \\
\quad \mid \mathit{otherwise} & = & \underline{\mathit{Cons}}\ a\ \lceil l \mathbin{/\!\!/} c \mathbin{\backslash\!\!\backslash} r' \rceil
\end{array}
$$

We have called it *meld* as $\mathit{para}\ \mathit{meld} :: \mu\mathit{Heap} \to \underline{\mathit{List}}\ (\mu\mathit{Heap})$ is a function one might find in a priority queue library, often called *deleteMin*. It returns the minimum element at the root and a new heap that is the melding of the left and right subheaps. This *Heap*-algebra is related to treesort's *SearchTree*-algebra, *shear*, but due to the contrasting ordering schemes the mechanics of extracting the next element are quite different.

The dual construction from *sift* is the $\underline{\mathit{List}}$-coalgebra that combines sorted lists; this time we will make a direct instantiation.

$$
\begin{aligned}
&\mathit{blend} :: \mathit{Heap}\,(\nu\underline{\mathit{List}} \times \underline{\mathit{List}}\quad(\nu\underline{\mathit{List}}))\\
&\qquad \to \underline{\mathit{List}}\quad(\nu\underline{\mathit{List}} + \mathit{Heap}\,(\nu\underline{\mathit{List}}))\\
&\mathit{blend}\ \varepsilon = \underline{\mathit{Nil}}\\
&\mathit{blend}\ ((l{\asymp}\underline{\mathit{Nil}}) /\!\!/ a \backslash\!\!\backslash (r{\asymp}\_)) = \underline{\mathit{Cons}}\ a\ (r\,\blacksquare)\\
&\mathit{blend}\ ((l{\asymp}\_) /\!\!/ a \backslash\!\!\backslash (r{\asymp}\underline{\mathit{Nil}})) = \underline{\mathit{Cons}}\ a\ (l\,\blacksquare)\\
&\mathit{blend}\ ((l{\asymp}(\underline{\mathit{Cons}}\ b\ l')) /\!\!/ a \backslash\!\!\backslash (r{\asymp}(\underline{\mathit{Cons}}\ c\ r')))\\
&\quad \mid b \leqslant c \qquad = \underline{\mathit{Cons}}\ a\ (\blacktriangleright (l' /\!\!/ b \backslash\!\!\backslash r))\\
&\quad \mid \mathit{otherwise} = \underline{\mathit{Cons}}\ a\ (\blacktriangleright (l /\!\!/ c \backslash\!\!\backslash r'))
\end{aligned}
$$

Note that $\mathit{apo}\ (\mathit{blend} \cdot \mathit{map}\ (\mathit{id} \vartriangle \mathit{out})) :: \mathit{Heap}\ (\nu\underline{\mathit{List}}) \to \nu\underline{\mathit{List}}$ is really a ternary version of merge, just as *apo glue* in Section 6.2 was a ternary append.

7.3 Putting Things Together

In the previous two sections, we took the approach of defining the natural transformations upfront. The algebras and coalgebras are the synthetic results, so we will express the final algorithms in terms of these. Fully assembled, our heapsort is defined as:

$$
\begin{aligned}
&\mathit{heapSort} :: \mu\mathit{List} \to \nu\underline{\mathit{List}}\\
&\mathit{heapSort} = \mathit{unfold}\ \mathit{deleteMin} \cdot \mathit{downcast} \cdot \mathit{fold}\ \mathit{heapInsert}\\
&\quad \textbf{where}\ \mathit{deleteMin}\ = \mathit{para}\ \mathit{meld}\\
&\qquad\qquad \mathit{heapInsert} = \mathit{apo}\ \mathit{heapIns}
\end{aligned}
$$

We use the names *deleteMin* and *heapInsert* to emphasise that this is exactly the expected algorithm for a function named *heapSort*.

The dual to heapsort is a strange creature, for which we will invent the name minglesort:

$$
\begin{aligned}
&\mathit{mingleSort} :: \mu\mathit{List} \to \nu\underline{\mathit{List}}\\
&\mathit{mingleSort} = \mathit{fold}\ (\mathit{apo}\ (\mathit{blend} \cdot \mathit{map}\ (\mathit{id} \vartriangle \mathit{out})))\\
&\qquad\qquad \cdot\ \mathit{downcast}\\
&\qquad\qquad \cdot\ \mathit{unfold}\ (\mathit{fold}\ \mathit{divvy})
\end{aligned}
$$

It uses the same intermediate data structure as heapsort, yet it behaves suspiciously like mergesort: the input list is recursively divided into two parts and then merged back together. This, of course is not quite true, as it actually divides into three parts: two lists of balanced length along with the minimum element. The merging phase is a similarly trimerous operation.

The true mergesort is really described by another intermediate data structure:

$$\textbf{data}\ \mathit{Bush}\ \mathit{bush} = \mathit{Leaf}\ K \mid \mathit{Fork}\ \mathit{bush}\ \mathit{bush}$$

A key facet of mergesort is that the first phase performs no comparisons: the input is recursively uninterleaved, which matches the dimerous nature of *Bush*. Remember that quick (tree) sort only performs comparisons in the first phase, and that heapsort, and thus minglesort, do so in *both* phases.

As minglesort impersonates mergesort, one would expect the dual of mergesort to be not unlike heapsort. It turns out that this is exactly the case; we have already continued this work and defined mergesort with *Bush* as the intermediate data structure and non-empty lists as the input and output data structure.

$$\textbf{data}\ \mathit{List1}\ \mathit{list1} = \mathit{Single}\ K \mid \mathit{Push}\ K\ \mathit{list1}$$

The non-empty list requirement comes from the fact that *Bush* is a non-empty container. For conciseness, we have not reported this work here. However, as a further justification that this is an intriguing avenue for future work, we should point out that $\mu\mathit{Bush}$ is isomorphic to $(K, \mu\mathit{Heap})$—heaps (trees) paired with an additional element (Hinze and James 2010). It is also isomorphic to $[\mu\mathit{Rose}]$, lists of rose trees, where *Rose* is defined as:

$$\textbf{data}\ \mathit{Rose}\ \mathit{rose} = \mathit{Rose}\ K\ [\mathit{rose}]$$

Rose trees can be used to implement binomial trees, and the type $[\mu\mathit{Rose}]$ is exactly the type of binomial heaps. Given the characteristics of these heaps, this also begs the question of how we apply our approach to a heapsort where the first phase runs in linear time. We leave the study of the relationship between mergesort, heapsort, and the various data intermediate structures to future investigations.

8. Related Work

Sorting algorithms, described in great detail by Knuth (1998), are often used in the functional programming community as prime examples of recursive morphisms. Recursive morphisms, known to be suitable for expressing many algorithms (Gibbons et al. 2001), have been widely studied (Gibbons 2003), especially ana- (Gibbons and Jones 1998), cata- (Hutton 1999), and paramorphisms (Meertens 1992). Apomorphisms are less frequently used, but Vene and Uustalu (1998) provide a detailed account.

Augusteijn (1999) presents the same sorting algorithms that we handle in this paper, but focuses on their implementation as hylomorphisms. A hylomorphism encapsulates a fold *after* an unfold, combining a coalgebra $A \to \mathsf{F}\,A$ and an algebra $\mathsf{F}\,B \to B$. The algebra and coalgebra have different carriers (A and B), but share the functor F. Their use has been explored in a wide variety of settings (Adámek et al. 2007; Capretta et al. 2006; Hinze et al. 2011; Meijer et al. 1991). However, we do not discuss hylomorphisms in this paper, instead using bialgebras, which combine an algebra $\mathsf{F}\,X \to X$ and a coalgebra $X \to \mathsf{G}\,X$: they share the same carrier, but operate on different functors. Moreover, we focus our attention on the (co-)algebras being recursive morphisms themselves. Dually, Gibbons (2007) has explored metamorphisms, i.e., an unfold after a fold, in which they gave quicksort as a hylomorphism and heapsort as a metamorphism as an example. We note that we obtain the same results with our approach but are also able to dualise each of these algorithms, yielding treesort as a metamorphism from quicksort and *mingleSort* as a hylomorphism from heapsort for free.

Others have looked at how to obtain “algorithms for free”, or develop programs calculationally. Bird (1996) give an account on formal derivation of sorting algorithms as folds and unfolds; Gibbons (1996) derives mergesort from insertion sort using the third homomorphism theorem; Oliveira (2002) analyses which sorting algorithms arise by combining independently useful algebras.

Our treatment of sorting algorithms as bialgebras and distributive laws is an application of the theoretical work that originates from Turi and Plotkin’s mathematical operational semantics (Turi and Plotkin 1997). Hinze and James (2011) also use this work to characterise the uniqueness of systems of equations that describe streams and codata in general. The types in this paper that we call F_+ and $\mathsf{G}_\times$ are really the free pointed functor for F and the cofree copointed functor for G (Lenisa et al. 2000); our Sections 3 and 5, and Appendix A have lightweight presentations of results from Turi and Plotkin (1997) and Lenisa et al. (2000).

9. Conclusion

Folds and unfolds already gave some insights into the structure of sorting algorithms, and we leveraged this fact by using a type-driven approach to guide our derivations. By taking the analysis into the world of bialgebras, we were able to isolate the computational essence of these sorting algorithms, which we read as distributive laws. This allowed us to talk of equivalence between two algorithms. Furthermore, we could *construct* one algorithm from another this way, giving us algorithms for free. Even in such a platonic domain, we were nevertheless able to address efficiency concerns, both algorithmically and by extending the theory to include para- and apomorphisms as more efficient recursion schemes.

Acknowledgments

This work has been partially funded by EPSRC grant number EP/J010995/1, and an EPSRC DTA Studentship. Thanks are due to Jeremy Gibbons for insightful discussions.

References

J. Adámek, D. Lücke, and S. Milius. Recursive coalgebras of finitary functors. *Theoretical Informatics and Applications*, 41(4):447–462, 2007. doi: 10.1051/ita:2007028.

L. Augusteijn. Sorting morphisms. In S. D. Swierstra, J. N. Oliveira, and P. R. Henriques, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer, 1999. doi: 10.1007/10704973_1.

R. S. Bird. Functional algorithm design. *Science of Computer Programming*, 26:15–31, 1996.

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

V. Capretta, T. Uustalu, and V. Vene. Recursive coalgebras from comonads. *Information and Computation*, 204(4):437–468, 2006. doi: 10.1016/j.ic.2005.08.005.

J. Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996. doi: 10.1017/S0956796800001908.

J. Gibbons. Origami programming. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 41–60. Palgrave, 2003. ISBN 1-4039-0772-2.

J. Gibbons. Metamorphisms: Streaming representation-changers. *Science of Computer Programming*, 65(2):108–139, 2007. doi: 10.1016/j.scico.2006.01.006.

J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *Proceedings of the International conference on Functional programming*, ICFP '98, pages 273–279. ACM, 1998. doi: 10.1145/289423.289455.

J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science*, 44(1):146–160, 2001. doi: 10.1016/S1571-0661(04)80906-X.

R. Hinze and D. W. H. James. Reason isomorphically! In B. C. Oliveira and M. Zalewski, editors, *Proceedings of the Workshop on Generic programming*, WGP '10, pages 85–96. ACM, 2010. doi: 10.1145/1863495.1863507.

R. Hinze and D. W. H. James. Proving the Unique Fixed-Point Principle Correct: An Adventure with Category Theory. In *Proceeding of the International conference on Functional programming*, ICFP '11, pages 359–371. ACM, 2011. doi: 10.1145/2034773.2034821.

R. Hinze, T. Harper, and D. W. H. James. Theory and Practice of Fusion. In J. Hage and M. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *LNCS*, pages 19–37. Springer, 2011. doi: 10.1007/978-3-642-24276-2_2.

G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999. doi: 10.1017/S0956796899003500.

D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

M. Lenisa, J. Power, and H. Watanabe. Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads. *Electronic Notes in Theoretical Computer Science*, 33:230–260, 2000. doi: 10.1016/S1571-0661(05)80350-0.

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

E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 124–144. Springer, 1991. doi: 10.1007/3540543961_7.

C. Okasaki. Three algorithms on braun trees. *Journal of Functional Programming*, 7(6):661–666, 1997. doi: 10.1017/S0956796897002876.

J. N. Oliveira. On the design of a periodic table of VDM specifications, 2002. Presented at the VDM'02 workshop.

D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proceedings of the Symposium on Logic in Computer Science*, LICS '97, pages 280–291. IEEE, 1997. doi: 10.1109/LICS.1997.614955.

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

A. Proofs

In this appendix we will take $\mathit{swop} :: \mathit{List} \circ \underline{\mathit{List}}_{\times} \dot{\to} \underline{\mathit{List}} \circ \mathit{List}_{+}$, from Section 5 and show how to make it symmetric. We do this so that we can apply the general theory of bialgebras and distributive laws to construct the initial and final bialgebras. This will be in a similar fashion to the conclusion of Section 3, albeit now in a more expressive setting. Having given the general construction, we will show how apo- and paramorphisms are 'shortcuts'. But first, we need to introduce a few definitions.

A.1 Casting

Folds that consume a list of type $\mu \mathit{List}$ require a *List*-algebra, but sometimes we will have a List_{+}-algebra in our hands. We can cast the latter into the former with the following function:

$$\begin{aligned}&\mathit{down}_{+} :: (\mathit{Functor}\,f) \Rightarrow (f_{+}\;a \to a) \to (f\;a \to a)\\&\mathit{down}_{+}\;b = b \cdot \mathit{inr}\end{aligned}$$

(In this appendix we will use *inl* and *inr* in place of ▪ and ▸, respectively, to better illustrate the duality with *outl* and *outr*.) We can also cast up:

$$\begin{aligned}&\mathit{up}_{+} :: (\mathit{Functor}\,f) \Rightarrow (f\;a \to a) \to (f_{+}\;a \to a)\\&\mathit{up}_{+}\;a = \mathit{id} \triangledown a\end{aligned}$$

Dually, unfolds that produce a list of type $\nu \underline{\mathit{List}}$ require a $\underline{\mathit{List}}$-coalgebra. Again, we can cast between the two:

$$\begin{aligned}&\mathit{down}_{\times} &&:: (\mathit{Functor}\,f) \Rightarrow (f_{\times}\;a \to a) \to (f\;\;a \to a)\\&\mathit{down}_{\times}\;d &&= \mathit{outr} \cdot d\\&\mathit{up}_{\times} &&:: (\mathit{Functor}\,f) \Rightarrow (f\;\;a \to a) \to (f_{\times}\;a \to a)\\&\mathit{up}_{\times}\;c &&= \mathit{id} \vartriangle c\end{aligned}$$

At a higher level of abstraction there is something deeper going on: there is an isomorphism between the category of *List*-algebras and the category of List_{+}-algebras—dually for $\underline{\mathit{List}}$-coalgebras and $\underline{\mathit{List}}_{\times}$-coalgebras. The functors that witness these isomorphisms are subject to various coherence conditions, but the details are beyond the scope of this paper, see Hinze and James (2011).

A.2 Symmetry

In Section 5, *swop* has the type $\mathsf{L} \circ \mathsf{O}_{\times} \dot{\to} \mathsf{O} \circ \mathsf{L}_{+}$, where *List* and $\underline{\mathit{List}}$ have been abbreviated to L and O, respectively. Given any natural transformation of type $\mathsf{L} \circ \mathsf{O}_{\times} \dot{\to} \mathsf{O} \circ \mathsf{L}_{+}$, we can construct a distributive law with the symmetric type $\mathsf{L}_{+} \circ \mathsf{O}_{\times} \dot{\to} \mathsf{O}_{\times} \circ \mathsf{L}_{+}$. We will use the name *swopsy* for the symmetric law constructed from *swop*; the two are related by the following equivalence.

$$\begin{aligned}\mathit{up}_{\times}\;c \cdot \mathit{up}_{+}\;a = \mathsf{O}_{\times}\;(\mathit{up}_{+}\;a) \cdot \mathit{swopsy} \cdot \mathsf{L}_{+}\;(\mathit{up}_{\times}\;c)\\\Longleftrightarrow\qquad\qquad\\c \cdot a = \mathsf{O}\;(\mathit{up}_{+}\;a) \cdot \mathit{swop} \cdot \mathsf{L}\;(\mathit{up}_{\times}\;c)\end{aligned}$$

(Note that here we have used the name of the functor in place of *map*, so that we can be clear as to which *map* is being used.) We can read this equivalence as a specification for *swopsy*; we shall also render it diagrammatically.

$$\begin{tikzcd}
\mathsf{L}_{+}\,X \arrow[d, "\mathit{up}_{+}\,a"'] \arrow[dr, "\mathsf{L}_{+}\,(\mathit{up}_{\times}\,c)"] & \\
X \arrow[dd, "\mathit{up}_{\times}\,c"'] & \mathsf{L}_{+}\,(\mathsf{O}_{\times}\,X) \arrow[d, "\mathit{swopsy}"] \\
 & \mathsf{O}_{\times}\,(\mathsf{L}_{+}\,X) \arrow[dl, "\mathsf{O}_{\times}\,(\mathit{up}_{+}\,a)"] \\
\mathsf{O}_{\times}\,X &
\end{tikzcd}
\quad\Longleftrightarrow\quad
\begin{tikzcd}
\mathsf{L}\,X \arrow[d, "a"'] \arrow[dr, "\mathsf{L}\,(\mathit{up}_{\times}\,c)"] & \\
X \arrow[dd, "c"'] & \mathsf{L}\,(\mathsf{O}_{\times}\,X) \arrow[d, "\mathit{swop}"] \\
 & \mathsf{O}\,(\mathsf{L}_{+}\,X) \arrow[dl, "\mathsf{O}\,(\mathit{up}_{+}\,a)"] \\
\mathsf{O}\,X &
\end{tikzcd}$$

From this specification, the definition of *swopsy* can be calculated. Again, this calculation is beyond the scope of this paper, see Hinze and James (2011). We will simply state the final construction. In fact, as the distributive law goes from a coproduct (initial) to a product (final), there are two constructions, and, following the theme of this paper, they are equal by uniqueness.

$$\begin{aligned} \mathit{swopsy} &= \mathsf{L}_{+}\ \mathit{outl}\ \vartriangle (\mathsf{O}\ \mathit{inl}\ \cdot\quad \mathit{outr}\ \triangledown\ \mathit{swop}) \\ &= \mathsf{O}_{\times}\ \mathit{inl}\ \ \triangledown\ (\quad \mathit{inr}\ \cdot \mathsf{L}\ \mathit{outl}\ \ \vartriangle \mathit{swop}) \end{aligned}$$

The following, regrettably detailed diagram, shows the initial and final *swopsy*-bialgebras. These are constructed in terms of folds and unfolds, which is why the terms are so complex: we need to mediate between L- and L_{+}-algebras, and O- and $\mathsf{O}_{\times}$-coalgebras.

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\mathsf{L}_{+}\,(\nu\mathsf{O}) \arrow[r, "\mathit{unfold}\,(\mathit{down}_{\times}\,(\mathit{swopsy}\cdot\mathsf{L}_{+}\,(\mathit{up}_{\times}\,\mathit{out})))"] & \nu\mathsf{O} \arrow[r, "\mathit{up}_{\times}\,\mathit{out}"] & \mathsf{O}_{\times}\,(\nu\mathsf{O}) \\
\mathsf{L}_{+}\,(\mu\mathsf{L}) \arrow[u, dashed] \arrow[r, "\mathit{up}_{+}\,\mathit{in}"] & \mu\mathsf{L} \arrow[u, dashed, "\begin{array}{c}\mathit{fold}\,(\mathit{down}_{+}\,(\mathit{unfold}\,(\mathit{down}_{\times}\,(\mathit{swopsy}\cdot\mathsf{L}_{+}\,(\mathit{up}_{\times}\,\mathit{out})))))\\=\\\mathit{unfold}\,(\mathit{down}_{\times}\,(\mathit{fold}\,(\mathit{down}_{+}\,(\mathsf{O}_{\times}\,(\mathit{up}_{+}\,\mathit{in})\cdot\mathit{swopsy}))))\end{array}"] \arrow[r, "\mathit{fold}\,(\mathit{down}_{+}\,(\mathsf{O}_{\times}\,(\mathit{up}_{+}\,\mathit{in})\cdot\mathit{swopsy}))"'] & \mathsf{O}_{\times}\,(\mu\mathsf{L}) \arrow[u, dashed]
\end{tikzcd}$$

It is worth comparing this diagram to the more simple diagram that concluded Section 3, which showed the initial and final *swap*-bialgebras. Where before we had $\mathit{in} :: \mathsf{L}\ (\mu\mathsf{L}) \to \mu\mathsf{L}$, we now have $\mathit{up}_{+}\ \mathit{in} :: \mathsf{L}_{+}\ (\mu\mathsf{L}) \to \mu\mathsf{L}$; and before we had $\mathit{unfold}\ (\mathit{swap} \cdot \mathsf{L}\ \mathit{out})$, but now we have $\mathit{unfold}\ (\mathit{down}_{\times}\ (\mathit{swopsy} \cdot \mathsf{L}_{+}\ (\mathit{up}_{\times}\ \mathit{out})))$, and in the centre of the diagram, where we apply *fold* to it, we must use a final down_{+} cast. Unfortunately, the selective but necessary use of casts makes the construction of the initial and final *swopsy*-bialgebras rather noisy.

A.3 Apomorphisms as a Shortcut

When we gave our final definition of insertion sort in Section 5, we wrote it as a fold of an apomorphism, rather than as a fold of an unfold. The reason for doing so was to utilise the computational efficiency of *swop* and apomorphisms—our insertion sort has linear complexity in the best case. From a theory perspective, apomorphisms also present a shortcut: we can use *swop* directly, rather than having to take the more general approach of constructing a distributive law that is symmetric. This leads to more concise terms, compared to what we see in the diagram above.

Paramorphisms and apomorphisms are useful in the case where we are building natural transformations involving F_{+} and $\mathsf{F}_{\times}$ functors; the following is a proof that $\mathit{apo}\ (\mathit{swop} \cdot \mathsf{L}\ (\mathit{id} \vartriangle \mathit{out}))$ is indeed a 'shortcut' for our general construction of the final *swopsy*-bialgebra.

$$\begin{aligned}
&\mathit{down}_{+}\ (\mathit{unfold}\ (\mathit{down}_{\times}\ (\mathit{swopsy} \cdot \mathsf{L}_{+}\ (\mathit{up}_{\times}\ \mathit{out}))))\\
=\ &\ \{\ \text{definition of } \mathit{down}_{+}\ \}\\
&\mathit{unfold}\ (\mathit{down}_{\times}\ (\mathit{swopsy} \cdot \mathsf{L}_{+}\ (\mathit{up}_{\times}\ \mathit{out}))) \cdot \mathit{inr}\\
=\ &\ \{\ \text{definition of } \mathit{down}_{\times}\ \}\\
&\mathit{unfold}\ (\mathit{outr} \cdot \mathit{swopsy} \cdot \mathsf{L}_{+}\ (\mathit{up}_{\times}\ \mathit{out})) \cdot \mathit{inr}\\
=\ &\ \{\ \text{definition of } \mathit{swopsy}\ \}\\
&\mathit{unfold}\ (\mathit{outr} \cdot (\mathsf{L}_{+}\ \mathit{outl} \vartriangle (\mathsf{O}\ \mathit{inl} \cdot \mathit{outr} \triangledown \mathit{swop})) \cdot \mathsf{L}_{+}\ (\mathit{up}_{\times}\ \mathit{out})) \cdot \mathit{inr}\\
=\ &\ \{\ \text{computation: } f_2 = \mathit{outr} \cdot (f_1 \vartriangle f_2)\ \}\\
&\mathit{unfold}\ ((\mathsf{O}\ \mathit{inl} \cdot \mathit{outr} \triangledown \mathit{swop}) \cdot \mathsf{L}_{+}\ (\mathit{up}_{\times}\ \mathit{out})) \cdot \mathit{inr}\\
=\ &\ \{\ \text{definition: } \mathsf{L}_{+}\ f = f + \mathsf{L}\ f\ \}\\
&\mathit{unfold}\ ((\mathsf{O}\ \mathit{inl} \cdot \mathit{outr} \triangledown \mathit{swop}) \cdot (\mathit{up}_{\times}\ \mathit{out} + \mathsf{L}\ (\mathit{up}_{\times}\ \mathit{out}))) \cdot \mathit{inr}\\
=\ &\ \{\ \text{functor fusion: } (f_1 \triangledown f_2) \cdot (g_1 + g_2) = f_1 \cdot g_1 \triangledown g_2 \cdot g_2\ \}\\
&\mathit{unfold}\ (\mathsf{O}\ \mathit{inl} \cdot \mathit{outr} \cdot \mathit{up}_{\times}\ \mathit{out} \triangledown \mathit{swop} \cdot \mathsf{L}\ (\mathit{up}_{\times}\ \mathit{out})) \cdot \mathit{inr}\\
=\ &\ \{\ \text{definition of } \mathit{up}_{\times}\ \}\\
&\mathit{unfold}\ (\mathsf{O}\ \mathit{inl} \cdot \mathit{outr} \cdot (\mathit{id} \vartriangle \mathit{out}) \triangledown \mathit{swop} \cdot \mathsf{L}\ (\mathit{id} \vartriangle \mathit{out})) \cdot \mathit{inr}\\
=\ &\ \{\ \text{computation: } f_2 = \mathit{outr} \cdot (f_1 \vartriangle f_2)\ \}\\
&\mathit{unfold}\ (\mathsf{O}\ \mathit{inl} \cdot \mathit{out} \triangledown \mathit{swop} \cdot \mathsf{L}\ (\mathit{id} \vartriangle \mathit{out})) \cdot \mathit{inr}\\
=\ &\ \{\ \text{definition of } \mathit{apo} \text{ as } \mathit{unfold}\ \}\\
&\mathit{apo}\ (\mathit{swop} \cdot \mathsf{L}\ (\mathit{id} \vartriangle \mathit{out}))
\end{aligned}$$

A dual proof would show that $\mathit{para}\ (\mathsf{O}\ (\mathit{id} \triangledown \mathit{out}) \cdot \mathit{swop})$ is equal to $\mathit{down}_{\times}\ (\mathit{fold}\ (\mathit{down}_{+}\ (\mathsf{O}_{\times}\ (\mathit{up}_{+}\ \mathit{in}) \cdot \mathit{swopsy})))$. However, where apomorphisms offer a shortcut in both efficiency and brevity, paramorphisms only offer the latter.