Sorting with Bialgebras and Distributive Laws

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

University of Oxford

September 9, 2012

This talk I

"Everyone knows" that insertion sort and selection sort are just two sides of the same coin:

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

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

This talk I

"Everyone knows" that insertion sort and selection sort are just two sides of the same coin:

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

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

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

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

This talk I

"Everyone knows" that insertion sort and selection sort are just two sides of the same coin:

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

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

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

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

. . . but that's certainly not immediately obvious from this definition.

This talk II

- "Everyone knows" that insertion sort and selection sort are just two sides of the same coin
- Bialgebras and distributive laws let us define them in one go, and give us a proof of their equivalence
- Sorting algorithms are type-driven: the types mostly dictate the implementation
- Duality gives us "algorithms for free"

Preliminaries I

We view lists as base functors...

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

Preliminaries I

We view lists as base functors...

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

...together with an associated functorial map:

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

Preliminaries I

We view lists as base functors...

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

...together with an associated functorial map:

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

This allows us to give a generic definition of *fold* using least fixed points:

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

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

Note that $\mu\ \mathit{List} \cong [K]$.

Preliminaries II

Unfolds are duals to folds, and require greatest fixed points:

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

$$unfold :: (Functor\ f) \Rightarrow (a \to f\ a) \to (a \to \nu\ f)$$
$$unfold\ f = Out^{\circ} \cdot fmap\ (unfold\ f) \cdot f$$

Even though μ and ν coincide in Haskell, we keep them distinct in our presentation.

Preliminaries II

Unfolds are duals to folds, and require greatest fixed points:

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

$$unfold :: (Functor\ f) \Rightarrow (a \to f\ a) \to (a \to \nu\ f)$$
$$unfold\ f = Out^{\circ} \cdot fmap\ (unfold\ f) \cdot f$$

Even though μ and ν coincide in Haskell, we keep them distinct in our presentation. As such, we will need conversions between ν and μ:

$$downcast :: (Functor\ f) \Rightarrow \nu\ f \to \mu\ f$$
$$downcast = In \cdot fmap\ downcast \cdot out$$

downcast is, in general, a lossy operation.

Preliminaries III

The other direction is always possible:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \rightarrow \nu\ f$$

Preliminaries III

The other direction is always possible:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \rightarrow \nu\ f$$

How can we write a function of this type? It could be a fold...

$$fold\ \ldots\ :\ \mu\ \mathsf{F} \rightarrow \nu\ \mathsf{G}$$

$$\ldots$$

Preliminaries III

The other direction is always possible:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \to \nu\ f$$

How can we write a function of this type? It could be a fold...

$$\begin{aligned} fold\ \ldots &: \mu\ \mathsf{F} \to \nu\ \mathsf{G} \\ \ldots &: \mathsf{F}\ (\nu\ \mathsf{G}) \to \nu\ \mathsf{G} \end{aligned}$$

Preliminaries III

The other direction is always possible:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \to \nu\ f$$

How can we write a function of this type? It could be a fold...

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

Preliminaries III

The other direction is always possible:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \to \nu\ f$$

How can we write a function of this type? It could be a fold...

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

Preliminaries III

The other direction is always possible:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \to \nu\ f$$

How can we write a function of this type? It could be a fold...

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

...or an unfold:

$$unfold\ (fold\ a) : \mu\ \mathsf{F} \to \nu\ \mathsf{G}$$

Preliminaries III

The other direction is always possible:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \to \nu\ f$$

How can we write a function of this type? It could be a fold...

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

...or an unfold:

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

Preliminaries III

The other direction is always possible:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \to \nu\ f$$

How can we write a function of this type? It could be a fold...

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

...or an unfold:

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

Preliminaries III

The other direction is always possible:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \to \nu\ f$$

How can we write a function of this type? It could be a fold...

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

...or an unfold:

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

Preliminaries IV

So we have two ways of defining *upcast*:

$$upcast :: (Functor\ f) \Rightarrow \mu\ f \to \nu\ f$$
$$upcast = fold \quad (unfold\ (fmap\ out)) = fold \quad Out^{\circ}$$

Preliminaries IV

So we have two ways of defining *upcast*:

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

Sorted lists

Let us create another datatype for representing sorted lists:

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

$$\begin{aligned}&\textbf{instance}\ Functor\ \underline{List}\ \textbf{where}\\ &\quad fmap\ f\ \underline{Nil} &&= \underline{Nil}\\ &\quad fmap\ f\ (\underline{Cons}\ k\ list) &&= \underline{Cons}\ k\ (f\ list)\end{aligned}$$

Note that $\underline{List} \cong List$.

Sorted lists

Let us create another datatype for representing sorted lists:

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

$$\begin{aligned}&\textbf{instance}\ Functor\ \underline{List}\ \textbf{where}\\ &\quad fmap\ f\ \underline{Nil} &&= \underline{Nil}\\ &\quad fmap\ f\ (\underline{Cons}\ k\ list) &&= \underline{Cons}\ k\ (f\ list)\end{aligned}$$

Note that $\underline{List} \cong List$.

A sorting function will now have the following type:

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

Sorting by swapping I

Again, we follow a type-directed approach:

$$f :: \mu\ List \to \nu\ \underline{List}$$

Sorting by swapping I

Again, we follow a type-directed approach:

$$
\begin{aligned}
&f :: \mu\ List \to \nu\ \underline{List} \\
&f = unfold\ g \\
&\quad \textbf{where}\ g = fold\ alg
\end{aligned}
$$

$$alg :: List\ (\underline{List}\ (\mu\ List)) \to \underline{List}\ (\mu\ List)$$

Sorting by swapping I

Again, we follow a type-directed approach:

$$
\begin{aligned}
&f :: \mu\ List \to \nu\ \underline{List} \\
&f = unfold\ g \\
&\quad \textbf{where}\ g = fold\ alg
\end{aligned}
$$

$$
\begin{aligned}
&alg :: List\ (\underline{List}\ (\mu\ List)) \to \underline{List}\ (\mu\ List) \\
&alg\ Nil \qquad\qquad\quad = \underline{Nil} \\
&alg\ (Cons\ x\ \underline{Nil}) = \underline{Cons}\ x\ (In\ Nil)
\end{aligned}
$$

Sorting by swapping I

Again, we follow a type-directed approach:

$$
\begin{aligned}
&f :: \mu\ List \to \nu\ \underline{List} \\
&f = unfold\ g \\
&\quad \textbf{where}\ g = fold\ alg
\end{aligned}
$$

$$
\begin{aligned}
&alg :: List\ (\underline{List}\ (\mu\ List)) \to \underline{List}\ (\mu\ List) \\
&alg\ Nil \qquad\qquad = \underline{Nil} \\
&alg\ (Cons\ x\ \underline{Nil}) = \underline{Cons}\ x\ (In\ Nil) \\
&alg\ (Cons\ x\ (\underline{Cons}\ y\ l)) \\
&\quad \mid x \leqslant y \qquad = \underline{Cons}\ x\ (In\ (Cons\ y\ l)) \\
&\quad \mid otherwise = \underline{Cons}\ y\ (In\ (Cons\ x\ l))
\end{aligned}
$$

Sorting by swapping I

Again, we follow a type-directed approach:

$$
\begin{aligned}
&f :: \mu\ List \to \nu\ \underline{List} \\
&f = unfold\ g \\
&\quad \textbf{where}\ g = fold\ alg
\end{aligned}
$$

$$
\begin{aligned}
&alg :: List\ (\underline{List}\ (\mu\ List)) \to \underline{List}\ (\mu\ List) \\
&alg\ Nil \qquad\qquad\quad = \underline{Nil} \\
&alg\ (Cons\ x\ \underline{Nil}) = \underline{Cons}\ x\ (In\ Nil) \\
&alg\ (Cons\ x\ (\underline{Cons}\ y\ l)) \\
&\quad \mid x \leqslant y \qquad = \underline{Cons}\ x\ (In\ (Cons\ y\ l)) \\
&\quad \mid otherwise = \underline{Cons}\ y\ (In\ (Cons\ x\ l))
\end{aligned}
$$

We've just defined bubble sort!

Sorting by swapping II

Dually, we could have defined a fold:

$$g :: \mu\ List \to \nu\ \underline{List}$$

Sorting by swapping II

Dually, we could have defined a fold:

$$
\begin{aligned}
&g :: \mu\ List \to \nu\ \underline{List} \\
&g = fold\ f \\
&\quad \textbf{where}\ f = unfold\ coalg \\
&coalg :: List\ (\nu\ \underline{List}) \to \underline{List}\ (List\ (\nu\ \underline{List}))
\end{aligned}
$$

Sorting by swapping II

Dually, we could have defined a fold:

$$
\begin{aligned}
&g :: \mu\ List \to \nu\ \underline{List} \\
&g = fold\ f \\
&\quad \textbf{where}\ f = unfold\ coalg \\
&coalg :: List\ (\nu\ \underline{List}) \to \underline{List}\ (List\ (\nu\ \underline{List})) \\
&coalg\ Nil \qquad\qquad\qquad\qquad\qquad\quad = \underline{Nil} \\
&coalg\ (Cons\ x\ (Out^{\circ}\ \underline{Nil})) = \underline{Cons}\ a\ Nil \\
&coalg\ (Cons\ x\ (Out^{\circ}\ (\underline{Cons}\ y\ l))) \\
&\quad \mid x \leqslant y \qquad\qquad\qquad\qquad\qquad\quad = \underline{Cons}\ x\ (Cons\ y\ l) \\
&\quad \mid otherwise \qquad\qquad\qquad\qquad\qquad\quad = \underline{Cons}\ y\ (Cons\ x\ l)
\end{aligned}
$$

Sorting by swapping II

Dually, we could have defined a fold:

$$
\begin{aligned}
&g :: \mu\ List \to \nu\ \underline{List} \\
&g = fold\ f \\
&\quad \textbf{where}\ f = unfold\ coalg \\
&coalg :: List\ (\nu\ \underline{List}) \to \underline{List}\ (List\ (\nu\ \underline{List})) \\
&coalg\ Nil \qquad\qquad\qquad\qquad\quad = \underline{Nil} \\
&coalg\ (Cons\ x\ (Out^{\circ}\ \underline{Nil})) = \underline{Cons}\ a\ Nil \\
&coalg\ (Cons\ x\ (Out^{\circ}\ (\underline{Cons}\ y\ l))) \\
&\quad \mid x \leqslant y \qquad\qquad\qquad\qquad\quad = \underline{Cons}\ x\ (Cons\ y\ l) \\
&\quad \mid otherwise \qquad\qquad\qquad\qquad\quad = \underline{Cons}\ y\ (Cons\ x\ l)
\end{aligned}
$$

We've just defined (naive) insertion sort!

Sorting by swapping III

Look at the algorithms side by side:

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

Sorting by swapping III

Look at the algorithms side by side:

$$
\begin{aligned}
&a :: List\ (\underline{List}\ (\mu\ List)) \to \underline{List}\ (\mu\ List)\\
&a\ Nil = \underline{Nil}\\
&a\ (Cons\ x\ \underline{Nil}) = \underline{Cons}\ x\ (In\ Nil)\\
&a\ (Cons\ x\ (\underline{Cons}\ y\ l))\\
&\quad \mid x \leqslant y = \underline{Cons}\ x\ (In\ (Cons\ y\ l))\\
&\quad \mid otherwise = \underline{Cons}\ y\ (In\ (Cons\ x\ l))
\end{aligned}
$$

$$
\begin{aligned}
&c :: List\ (\nu\ \underline{List}) \to \underline{List}\ (List\ (\nu\ \underline{List}))\\
&c\ Nil = \underline{Nil}\\
&c\ (Cons\ x\ (Out^{\circ}\ \underline{Nil})) = \underline{Cons}\ x\ Nil\\
&c\ (Cons\ x\ (Out^{\circ}\ (\underline{Cons}\ y\ l)))\\
&\quad \mid x \leqslant y = \underline{Cons}\ x\ (Cons\ y\ l)\\
&\quad \mid otherwise = \underline{Cons}\ y\ (Cons\ x\ l)
\end{aligned}
$$

Sorting by swapping III

Look at the algorithms side by side:

$$
\begin{aligned}
&a :: List\ (\underline{List}\ (\mu\ List)) \to \underline{List}\ (\mu\ List)\\
&a\ Nil = \underline{Nil}\\
&a\ (Cons\ x\ \underline{Nil}) = \underline{Cons}\ x\ (In\ Nil)\\
&a\ (Cons\ x\ (\underline{Cons}\ y\ l))\\
&\quad \mid x \leqslant y = \underline{Cons}\ x\ (In\ (Cons\ y\ l))\\
&\quad \mid otherwise = \underline{Cons}\ y\ (In\ (Cons\ x\ l))
\end{aligned}
$$

$$
\begin{aligned}
&c :: List\ (\nu\ \underline{List}) \to \underline{List}\ (List\ (\nu\ \underline{List}))\\
&c\ Nil = \underline{Nil}\\
&c\ (Cons\ x\ (Out^{\circ}\ \underline{Nil})) = \underline{Cons}\ x\ Nil\\
&c\ (Cons\ x\ (Out^{\circ}\ (\underline{Cons}\ y\ l)))\\
&\quad \mid x \leqslant y = \underline{Cons}\ x\ (Cons\ y\ l)\\
&\quad \mid otherwise = \underline{Cons}\ y\ (Cons\ x\ l)
\end{aligned}
$$

We can unify them in a single *natural transformation*:

$$
\begin{aligned}
&swap :: List\ (\underline{List}\ x) \to \underline{List}\ (List\ x)\\
&swap\ Nil = \underline{Nil}\\
&swap\ (Cons\ x\ \underline{Nil}) = \underline{Cons}\ x\ Nil\\
&swap\ (Cons\ x\ (\underline{Cons}\ y\ l))\\
&\quad \mid x \leqslant y = \underline{Cons}\ x\ (Cons\ y\ l)\\
&\quad \mid otherwise = \underline{Cons}\ y\ (Cons\ x\ l)
\end{aligned}
$$

Sorting by swapping IV

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

Sorting by swapping IV

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

And now we can write bubble sort and naive insertion sort using *swap*:

$$
\begin{aligned}
&\mathit{bubbleSort}', \mathit{naiveInsertionSort}' :: \mu\ \mathit{List} \to \nu\ \underline{\mathit{List}}\\
&\mathit{bubbleSort}' &&= \mathit{unfold}\ (\mathit{fold}\quad (\mathit{fmap}\ \mathit{In} \cdot \mathit{swap}))\\
&\mathit{naiveInsertionSort}' &&= \mathit{fold}\quad (\mathit{unfold}\ (\mathit{swap} \cdot \mathit{fmap}\ \mathit{out}))
\end{aligned}
$$

Sorting by swapping IV

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

And now we can write bubble sort and naive insertion sort using *swap*:

$$
\begin{aligned}
&\mathit{bubbleSort}', \mathit{naiveInsertionSort}' :: \mu\ \mathit{List} \to \nu\ \underline{\mathit{List}}\\
&\mathit{bubbleSort}' &&= \mathit{unfold}\ (\mathit{fold}\quad (\mathit{fmap}\ \mathit{In} \cdot \mathit{swap}))\\
&\mathit{naiveInsertionSort}' &&= \mathit{fold}\quad (\mathit{unfold}\ (\mathit{swap} \cdot \mathit{fmap}\ \mathit{out}))
\end{aligned}
$$

But how can we write true insertion sort?

Interlude—paramorphisms

To move on to more efficient sorting algorithms we need a variant on *fold*. We start by defining products and a "fan-out" operator:

$$\textbf{data}\ a \times b = Pair\ \{outl :: a, outr :: b\}$$

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

Interlude—paramorphisms

To move on to more efficient sorting algorithms we need a variant on *fold*. We start by defining products and a "fan-out" operator:

$$\textbf{data}\ a \times b = Pair\ \{outl :: a, outr :: b\}$$

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

And now we can define paramorphisms:

$$para :: (Functor\ f) \Rightarrow (f\ (\mu\ f \times a) \to a) \to (\mu\ f \to a)$$
$$para\ f = f \cdot fmap\ (id \triangle para\ f) \cdot in^{\circ}$$

The important difference is in the type of the algebra, which is now $f\ (\mu\ f \times a) \to a$ instead of just $f\ a \to a$.

Interlude—apomorphisms

Dually, we need sums and a "fan-in" operator:

$$\textbf{data}\ a + b = \mathit{Stop}\ a \mid \mathit{Go}\ b$$

$$\begin{aligned}
&(\triangledown) :: (a \to c) \to (b \to c) \to (a + b \to c)\\
&(f \triangledown g)\ (\mathit{Stop}\ a) = f\ \ a\\
&(f \triangledown g)\ (\mathit{Go}\quad b) = g\ b
\end{aligned}$$

Interlude—apomorphisms

Dually, we need sums and a "fan-in" operator:

$$\textbf{data}\ a + b = \mathit{Stop}\ a \mid \mathit{Go}\ b$$

$$\begin{aligned}
&(\triangledown) :: (a \to c) \to (b \to c) \to (a + b \to c)\\
&(f \triangledown g)\ (\mathit{Stop}\ a) = f\ \ a\\
&(f \triangledown g)\ (\mathit{Go}\quad b) = g\ b
\end{aligned}$$

Then we can define apomorphisms:

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

The corecursion is now split into two branches, with no recursive call on the left.

Interlude—types of (co-)algebras

We can again use a type-directed approach to derive the types of our new (co-)algebras:

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

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

where

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

Insertion and selection sort

With apomorphisms, we can write an insertion sort that stops scanning after inserting an element:

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

$$
\mathit{ins} :: \mathit{List}\ (\nu\ \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\nu\ \underline{\mathit{List}} + \mathit{List}\ (\nu\ \underline{\mathit{List}}))
$$

Insertion and selection sort

With apomorphisms, we can write an insertion sort that stops scanning after inserting an element:

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

$$
\begin{aligned}
&\mathit{ins} :: \mathit{List}\ (\nu\ \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\nu\ \underline{\mathit{List}} + \mathit{List}\ (\nu\ \underline{\mathit{List}})) \\
&\mathit{ins}\ \mathit{Nil} &&= \underline{\mathit{Nil}} \\
&\mathit{ins}\ (\mathit{Cons}\ a\ (\mathit{Out}^{\circ}\ \underline{\mathit{Nil}})) &&= \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ (\mathit{Out}^{\circ}\ \underline{\mathit{Nil}})) \\
&\mathit{ins}\ (\mathit{Cons}\ a\ (\mathit{Out}^{\circ}\ (\underline{\mathit{Cons}}\ b\ x'))) \\
&\quad \mid a \leqslant b &&= \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ (\mathit{Out}^{\circ}\ (\underline{\mathit{Cons}}\ b\ x'))) \\
&\quad \mid \mathit{otherwise} &&= \underline{\mathit{Cons}}\ b\ (\mathit{Go}\ (\mathit{Cons}\ a\ x'))
\end{aligned}
$$

Swap and stop: *swop*

Let's skip selection sort, and go straight to the natural transformation:

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

Swap and stop: *swop*

From *swop* we get both insertion and selection sort:

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

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

Algorithms for free!

Quicksort and Treesort I

To move away from quadratic complexity algorithms, we need to use binary trees as an intermediate data structure:

$$\textbf{data}\ \mathit{Tree}\ \mathit{tree} = \mathit{Empty} \mid \mathit{Node}\ \mathit{tree}\ K\ \mathit{tree}$$

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ \mathit{Tree}\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ \mathit{Empty} = \mathit{Empty}\\ &\quad \mathit{fmap}\ f\ (\mathit{Node}\ l\ k\ r) = \mathit{Node}\ (f\ l)\ k\ (f\ r)\end{aligned}$$

Our trees will respect the search tree property: 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.

Quicksort and Treesort II

We first need to build a tree. Let's do that as the unfold of a fold:

$$
\begin{aligned}
&\mathit{pivot} :: \mathit{List}\ (\mathit{Tree}\ (\mu\ \mathit{List})) \to \mathit{Tree}\ (\mu\ \mathit{List}) \\
&\mathit{pivot}\ \mathit{Nil} &&= \mathit{Empty} \\
&\mathit{pivot}\ (\mathit{Cons}\ a\ \mathit{Empty}) &&= \mathit{Node}\ (\mathit{In}\ \mathit{Nil})\ a\ (\mathit{In}\ \mathit{Nil}) \\
&\mathit{pivot}\ (\mathit{Cons}\ a\ (\mathit{Node}\ l\ b\ r)) \\
&\quad \mid a \leqslant b &&= \mathit{Node}\ (\mathit{In}\ (\mathit{Cons}\ a\ l))\ b\ r \\
&\quad \mid \mathit{otherwise} &&= \mathit{Node}\ l\ b\ (\mathit{In}\ (\mathit{Cons}\ a\ r))
\end{aligned}
$$

Quicksort and Treesort III

The following natural transformation arises out of *pivot*:

$$
\begin{aligned}
&\mathit{sprout} :: \mathit{List}\,(a \times \mathit{Tree}\,a) \to \mathit{Tree}\,(a + \mathit{List}\,a)\\
&\mathit{sprout}\ \mathit{Nil} &&= \mathit{Empty}\\
&\mathit{sprout}\ (\mathit{Cons}\ a\ (\mathit{Pair}\ t\ \mathit{Empty})) &&= \mathit{Node}\ (\mathit{Stop}\ t)\ a\ (\mathit{Stop}\ t)\\
&\mathit{sprout}\ (\mathit{Cons}\ a\ (\mathit{Pair}\ t\ (\mathit{Node}\ l\ b\ r)))\\
&\quad \mid a \leqslant b &&= \mathit{Node}\ (\mathit{Go}\ (\mathit{Cons}\ a\ l))\ b\ (\mathit{Stop}\ r)\\
&\quad \mid \mathit{otherwise} &&= \mathit{Node}\ (\mathit{Stop}\ l)\ b\ (\mathit{Go}\ (\mathit{Cons}\ a\ r))
\end{aligned}
$$

Quicksort and Treesort IV

The coalgebra that is dual to *pivot* comes for free:

$$
\begin{aligned}
&\mathit{treeIns} :: \mathit{List}\ (\nu\ \mathit{Tree}) \to \mathit{Tree}\ (\nu\ \mathit{Tree} + \mathit{List}\ (\nu\ \mathit{Tree})) \\
&\mathit{treeIns}\ \mathit{Nil} &&= \mathit{Empty} \\
&\mathit{treeIns}\ (\mathit{Cons}\ a\ (\mathit{Out}^\circ\ \mathit{Empty})) &&= \mathit{Node}\ (\mathit{Stop}\ (\mathit{Out}^\circ\ \mathit{Empty}))\ a \\
&&&\quad (\mathit{Stop}\ (\mathit{Out}^\circ\ \mathit{Empty})) \\
&\mathit{treeIns}\ (\mathit{Cons}\ a\ (\mathit{Out}^\circ\ (\mathit{Node}\ l\ b\ r))) \\
&\quad \mid a \leqslant b &&= \mathit{Node}\ (\mathit{Go}\ (\mathit{Cons}\ a\ l))\ b\ (\mathit{Stop}\ r) \\
&\quad \mid \mathit{otherwise} &&= \mathit{Node}\ (\mathit{Stop}\ l)\ b\ (\mathit{Go}\ (\mathit{Cons}\ a\ r))
\end{aligned}
$$

A useful, efficient tree insertion function, for free!

Quicksort and Treesort IV

The coalgebra that is dual to *pivot* comes for free:

$$
\begin{aligned}
&\mathit{treeIns} :: \mathit{List}\ (\nu\ \mathit{Tree}) \to \mathit{Tree}\ (\nu\ \mathit{Tree} + \mathit{List}\ (\nu\ \mathit{Tree})) \\
&\mathit{treeIns}\ \mathit{Nil} &&= \mathit{Empty} \\
&\mathit{treeIns}\ (\mathit{Cons}\ a\ (\mathit{Out}^{\circ}\ \mathit{Empty})) &&= \mathit{Node}\ (\mathit{Stop}\ (\mathit{Out}^{\circ}\ \mathit{Empty}))\ a \\
&&&\quad (\mathit{Stop}\ (\mathit{Out}^{\circ}\ \mathit{Empty})) \\
&\mathit{treeIns}\ (\mathit{Cons}\ a\ (\mathit{Out}^{\circ}\ (\mathit{Node}\ l\ b\ r))) \\
&\quad \mid a \leqslant b &&= \mathit{Node}\ (\mathit{Go}\ (\mathit{Cons}\ a\ l))\ b\ (\mathit{Stop}\ r) \\
&\quad \mid \mathit{otherwise} &&= \mathit{Node}\ (\mathit{Stop}\ l)\ b\ (\mathit{Go}\ (\mathit{Cons}\ a\ r))
\end{aligned}
$$

A useful, efficient tree insertion function, for free! The function for generating search trees follows:

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

Quicksort and Treesort V

After producing a search tree, we consume it:

$$
\begin{aligned}
&\mathit{wither} :: \mathit{Tree}\ (a \times \underline{\mathit{List}}\ a) \to \underline{\mathit{List}}\ (a + \mathit{Tree}\ a) \\
&\mathit{wither}\ \mathit{Empty} &&= \underline{\mathit{Nil}} \\
&\mathit{wither}\ (\mathit{Node}\ (\mathit{Pair}\ l\ \underline{\mathit{Nil}})\ a\ (\mathit{Pair}\ r\ \_)) &&= \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ r) \\
&\mathit{wither}\ (\mathit{Node}\ (\mathit{Pair}\ l\ (\underline{\mathit{Cons}}\ b\ l'))\ a\ (\mathit{Pair}\ r\ \_)) &&= \underline{\mathit{Cons}}\ b\ (\mathit{Go}\ (\mathit{Node}\ l'\ a\ r))
\end{aligned}
$$

Quicksort and Treesort VI

And these are the algorithms that can be extracted from *wither*:

$$
\begin{aligned}
&\mathit{glue} :: \mathit{Tree}\ (\nu\ \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\nu\ \underline{\mathit{List}} + \mathit{Tree}\ (\nu\ \underline{\mathit{List}})) \\
&\mathit{glue}\ \mathit{Empty} &&= \underline{\mathit{Nil}} \\
&\mathit{glue}\ (\mathit{Node}\ (\mathit{Out}^{\circ}\ \underline{\mathit{Nil}})\ a\ r) &&= \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ r) \\
&\mathit{glue}\ (\mathit{Node}\ (\mathit{Out}^{\circ}\ (\underline{\mathit{Cons}}\ b\ l))\ a\ r) &&= \underline{\mathit{Cons}}\ b\ (\mathit{Go}\ (\mathit{Node}\ l\ a\ r))
\end{aligned}
$$

Quicksort and Treesort VI

And these are the algorithms that can be extracted from *wither*:

$$
\begin{aligned}
&\mathit{glue} :: \mathit{Tree}\ (\nu\ \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\nu\ \underline{\mathit{List}} + \mathit{Tree}\ (\nu\ \underline{\mathit{List}})) \\
&\mathit{glue}\ \mathit{Empty} &&= \underline{\mathit{Nil}} \\
&\mathit{glue}\ (\mathit{Node}\ (\mathit{Out}^{\circ}\ \underline{\mathit{Nil}})\ a\ r) &&= \underline{\mathit{Cons}}\ a\ (\mathit{Stop}\ r) \\
&\mathit{glue}\ (\mathit{Node}\ (\mathit{Out}^{\circ}\ (\underline{\mathit{Cons}}\ b\ l))\ a\ r) &&= \underline{\mathit{Cons}}\ b\ (\mathit{Go}\ (\mathit{Node}\ l\ a\ r))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{shear} :: \mathit{Tree}\ (\mu\ \mathit{Tree} \times \underline{\mathit{List}}\ (\mu\ \mathit{Tree})) \to \underline{\mathit{List}}\ (\mu\ \mathit{Tree}) \\
&\mathit{shear}\ \mathit{Empty} &&= \underline{\mathit{Nil}} \\
&\mathit{shear}\ (\mathit{Node}\ (\mathit{Pair}\ l\ \underline{\mathit{Nil}})\ a\ (\mathit{Pair}\ r\ \_)) &&= \underline{\mathit{Cons}}\ a\ r \\
&\mathit{shear}\ (\mathit{Node}\ (\mathit{Pair}\ l\ (\underline{\mathit{Cons}}\ b\ l'))\ a\ (\mathit{Pair}\ r\ \_)) &&= \underline{\mathit{Cons}}\ b\ (\mathit{In}\ (\mathit{Node}\ l'\ a\ r))
\end{aligned}
$$

Quicksort and Treesort VII

Using *wither* we get two algorithms for flattening a tree to a list:

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

Quicksort and Treesort VII

Using *wither* we get two algorithms for flattening a tree to a list:

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

Now we can finally put everything together, defining quicksort and treesort:

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

Conclusion and future work

In the paper:

- Heapsort
- A hint at mergesort
- Beautiful proofs and intuition

Conclusion and future work

In the paper:

- Heapsort
- A hint at mergesort
- Beautiful proofs and intuition

Future work:

- Investigate how the intermediate structures dictate the sorting algorithm