The Computational Essence of Sorting Algorithms

Ralf Hinze

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze@cs.ox.ac.uk
http://www.cs.ox.ac.uk/ralf.hinze/

November 2012

Joint Work with Daniel W.H. James, Thomas Harper,
Nicolas Wu and José Pedro Magalhães.

Section 1

Prologue

algorithmics

$\updownarrow$

programming languages

Any customer can have a car painted any colour that he wants so long as it is black.

My Life and Work (1922) by Henry Ford

The limits of my language mean the limits of my world.

Tractatus Logico-Philosophicus (1922)
by Ludwig Wittgenstein
translated by C. K. Ogden and D. F. Pears

design of sorting algorithms

$\updownarrow$

semantics of programming languages

categorical algorithmics

Section 2

Insertion and selection sort I

⟨. . .⟩ the area of sorting and searching provides an ideal framework for discussing a wide variety of important general issues:

- How are good algorithms discovered?
- ⟨. . .⟩

Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of sorting or searching!

The Art of Computer Programming
Volume 3: Sorting and Searching
by Donald E. Knuth

. . . .

The time you spent working on the challenge problem will pay dividends as you continue to read this chapter. Chances are your solution is one of the following types:

A. *An insertion sort.* The items are considered one at a time, and each new item is inserted into the appropriate position relative to the previously-sorted items. (This is the way many bridge players sort their hands, picking up one card at a time.)

B. *An exchange sort.* If two items are found to be out of order, they are interchanged. This process is repeated until no more exchanges are necessary.

C. *A selection sort.* First the smallest (or perhaps the largest) item is located, and it is somehow separated from the rest; then the next smallest (or next largest) is selected, and so on.

D. *An enumeration sort.* Each item is compared with each of the others; an item's final position is determined by the number of keys that it exceeds.

E. *A special-purpose sort*, which works nicely for sorting five elements as stated in the problem, but does not readily generalize to larger numbers of items.

F. *A lazy attitude*, with which you ignored the suggestion above and decided not to solve the problem at all. Sorry, by now you have read too far and you have lost your chance.

G. *A new, super sorting technique* that is a definite improvement over known methods. (Please communicate this to the author at once.)

2.1 Insertion sort

- also known as "bridge player sort"
- *idea:* maintain an ordered sequence (invariant)
- consume the input sequence one by one
- insert each element into the ordered sequence
- the focus is on the *input*

2.1 Knuth's straight insertion sort

Program S (*Straight insertion sort*). The records to be sorted are in locations INPUT+1 through INPUT+N; they are sorted in place in the same area, on a full-word key. $\text{rI1} \equiv j - N$; $\text{rI2} \equiv i$; $\text{rA} \equiv R \equiv K$; assume that $N \geq 2$.

START	ENT1	2-N	1	<u>*S1. Loop on j.*</u> $j \leftarrow 2$.
2H	LDA	INPUT+N,1	$N-1$	<u>*S2. Set up i, K, R.*</u>
	ENT2	N-1,1	$N-1$	$i \leftarrow j-1$.
3H	CMPA	INPUT,2	$B+N-1-A$	<u>*S3. Compare* $K : K_i$.</u>
	JGE	5F	$B+N-1-A$	To S5 if $K \geq K_i$.
4H	LDX	INPUT,2	B	<u>*S4. Move* R_i*, decrease i.*</u>
	STX	INPUT+1,2	B	$R_{i+1} \leftarrow R_i$.
	DEC2	1	B	$i \leftarrow i-1$.
	J2P	3B	B	To S3 if $i > 0$.
5H	STA	INPUT+1,2	$N-1$	<u>*S5. R into* R_{i+1}.</u>
	INC1	1	$N-1$	
	J1NP	2B	$N-1$	$2 \leq j \leq N$. ▮

2.1 A functional insertion sort

$$\begin{aligned}
&\mathit{insertSort} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \to [\,a\,]\\
&\mathit{insertSort}\ [\,] \qquad = [\,]\\
&\mathit{insertSort}\ (x : xs) = x\ \text{`}\mathit{insert}\text{`}\ \mathit{insertSort}\ xs
\end{aligned}$$

$$\begin{aligned}
&\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow a \to [\,a\,] \to [\,a\,]\\
&\mathit{insert}\ x\ [\,] \qquad = [\,x\,]\\
&\mathit{insert}\ x\ (y : xs)\\
&\quad \mid x \leqslant y \qquad = x : y : xs\\
&\quad \mid \mathit{otherwise} = y : \mathit{insert}\ x\ xs
\end{aligned}$$

2.1 *insertSort* is an instance of *foldr*

$$x_1 : (x_2 : \cdots : (x_{n-1} : (x_n : [\,])))$$

$$\Big\downarrow \quad \mathit{foldr}\ \mathit{insert}\ [\,]$$

$$x_1 \ \text{`}\mathit{insert}\text{`}\ (x_2 \ \text{`}\mathit{insert}\text{`} \cdots \text{`}\mathit{insert}\text{`}\ (x_{n-1} \ \text{`}\mathit{insert}\text{`}\ (x_n \ \text{`}\mathit{insert}\text{`}\ [\,])))$$

2.1 *foldr* captures a recursion pattern

$$x_1 : (x_2 : \cdots : (x_{n-1} : (x_n : [\,])))$$

$$\Big\downarrow \; \mathit{foldr}\ (\otimes)\ e$$

$$x_1 \otimes (x_2 \otimes \cdots \otimes (x_{n-1} \otimes (x_n \otimes e)))$$

Slogan: replacing constructors by functions.

2.1 Insertion sort, revisited

$$
\begin{aligned}
&\mathit{insertSort} :: (\mathit{Ord}\ a) \Rightarrow [a] \to [a] \\
&\mathit{insertSort} = \mathit{foldr}\ \mathit{insert}\ [\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow a \to [a] \to [a] \\
&\mathit{insert}\ x\ [\,] \qquad = [x] \\
&\mathit{insert}\ x\ (y : \mathit{xs}) \\
&\quad \mid x \leqslant y \qquad = x : y : \mathit{xs} \\
&\quad \mid \mathit{otherwise} = y : \mathit{insert}\ x\ \mathit{xs}
\end{aligned}
$$

But what about *insert*?

2.2 Selection sort

- *idea:* produce an ordered sequence by repeatedly selecting the minimum element
- initial seed or state: input sequence
- the focus is on the *output*
- selection sort is in some sense *dual* to insertion sort

2.2 Knuth's straight selection sort

Program S (*Straight selection sort*). As in previous programs of this chapter, the records in locations `INPUT+1` through `INPUT+N` are sorted in place, on a full-word key. rA ≡ current maximum, rI1 ≡ $j - 1$, rI2 ≡ k (the current search position), rI3 ≡ i. Assume that $N \geq 2$.

```
START ENT1 N-1          1      S1. Loop on j. j ← N.
2H    ENT2 0,1          N−1    S2. Find max(K_1,...,K_j). k ← j−1.
      ENT3 1,1          N−1    i ← j.
      LDA  INPUT,3      N−1    rA ← K_i.
8H    CMPA INPUT,2      A
      JGE  *+3          A      Jump if K_i ≥ K_k.
      ENT3 0,2          B      Otherwise set i ← k,
      LDA  INPUT,3      B        rA ← K_i.
      DEC2 1            A      k ← k−1.
      J2P  8B           A      Repeat if k > 0.
      LDX  INPUT+1,1    N−1    S3. Exchange with R_j.
      STX  INPUT,3      N−1    R_i ← R_j.
      STA  INPUT+1,1    N−1    R_j ← rA.
      DEC1 1            N−1
      J1P  2B           N−1    N ≥ j ≥ 2. ▮
```

2.2 A functional selection sort

$$
\begin{aligned}
&\mathit{selectSort} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \to [\,a\,]\\
&\mathit{selectSort} = \mathit{unfoldr}\ \mathit{select}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{select} :: (\mathit{Ord}\ a) \Rightarrow [\,a\,] \to \mathit{Maybe}\ (a, [\,a\,])\\
&\mathit{select}\ [\,] = \mathit{Nothing}\\
&\mathit{select}\ (x : \mathit{xs})\\
&\quad = \mathbf{case}\ \mathit{select}\ \mathit{xs}\ \mathbf{of}\\
&\qquad \mathit{Nothing} \qquad\qquad \to \mathit{Just}\ (x, [\,])\\
&\qquad \mathit{Just}\ (y, \mathit{ys})\\
&\qquad\quad \mid x \leqslant y \qquad\quad \to \mathit{Just}\ (x, \mathit{xs})\\
&\qquad\quad \mid \mathit{otherwise} \to \mathit{Just}\ (y, x : \mathit{ys})
\end{aligned}
$$

But what about *select*?

2.2 Intermediate summary

- *unfoldr* is the *categorical* dual of *foldr*
- insertion is *in some sense* dual to selection
- the details of this relationship are somewhat shrouded by our language
- to illuminate the connection, we use a *type-driven approach* to algorithm design

Section 3

Background

3.0 Background

- the list datatype is recursively defined

$$\mathbf{data}\ [\,a\,] = [\ ] \mid a : [\,a\,]$$

- *semantics:* fixed point of an associated higher-order type constructor

$$\mathbf{data}\ \mathit{List}\ \mathit{list}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{list}\ a)$$

- for simplicity, let's fix the type of elements,

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

  where K is some key type that admits ordering
- *categorical concept:* initial algebra of a functor
- NB. *List* is the non-recursive *base functor* of the list datatype; $\mu \mathit{List}$ is then the recursive list type

3.1 Functor in Haskell

- in Haskell, a functor is given by a datatype definition

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

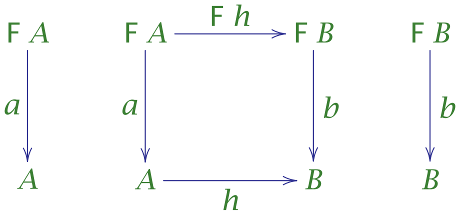
- and an associated *Functor* declaration

$$\begin{array}{l}\textbf{instance}\ \mathit{Functor}\ \mathit{List}\ \textbf{where}\\ \quad \mathit{map}\ f\ \mathit{Nil} = \mathit{Nil}\\ \quad \mathit{map}\ f\ (\mathit{Cons}\ k\ \mathit{ks}) = \mathit{Cons}\ k\ (f\ \mathit{ks})\end{array}$$

- the mapping function changes the elements of a container, but keeps its structure intact
- the type parameter *list* marks the ‘recursive’ component

3.2 Algebra and algebra homomorphism

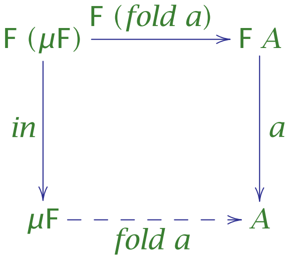
- an F-algebra a is an arrow of type $\mathsf{F}\,A \to A$



- an F-algebra homomorphism h preserves the structure
- F-algebras and homomorphisms form a category

3.2 Initial algebra

- the initial object in this category, the initial F-algebra, is the 'least' fixed point of F



- initiality entails that there is a *unique* homomorphism *from* the initial algebra to any algebra a, called $\mathit{fold}\ a$
- initial *List*-algebra: finite lists (in **Set**)

3.2 Initial algebra in Haskell

- in Haskell, μf can be defined

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

- as an aside, $In\ a$ will be written as $[\,a\,]$
- since in is an isomorphism, we can turn the commuting diagram into a *generic* definition of $fold$

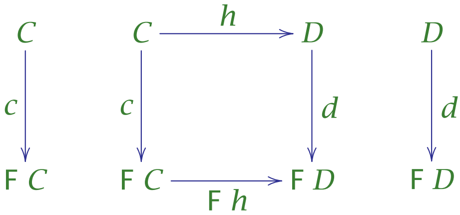
$$\begin{aligned} &fold :: (Functor\ f) \Rightarrow (f\ a \to a) \to (\mu f \to a) \\ &fold\ f = f \cdot map\ (fold\ f) \cdot in^{\circ} \end{aligned}$$

3.3 Duality

- we obtain the categorical *dual* by reversing the arrows
- algebras dualise to coalgebras
- initial algebras dualise to final coalgebras

3.3 Coalgebra

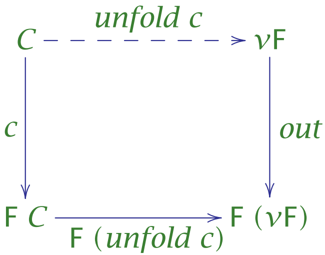
- an F-coalgebra c is an arrow of type $C \to \mathsf{F}\ C$



- an F-coalgebra homomorphism h preserves the structure
- F-coalgebras and homomorphisms form a category

3.3 Final coalgebra

- the final object in this category, the final F-coalgebra, is the 'greatest' fixed point of F



- finality entails that there is a *unique* homomorphism *to* the final coalgebra from any coalgebra *c*, called *unfold c*
- final *List*-coalgebra: finite and infinite lists (in **Set**)

3.3 Final coalgebra in Haskell

- in Haskell, νf can be defined

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

- as an aside, $Out^{\circ}\ a$ will be written as $\lfloor a \rfloor$
- since *out* is an isomorphism, we can turn the commuting diagram into a *generic* definition of *unfold*

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

- Haskell: initial algebras and final coalgebras coincide!

3.4 Embedding initial into final

Least fixed points can be embedded into greatest fixed points.

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

3.4 Embedding initial into final

Least fixed points can be embedded into greatest fixed points.

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

How to define *upcast*? We can write it as a fold . . .

$$fold\ \ldots : \mu\mathsf{F} \to \nu\mathsf{F}$$

$$\ldots$$

3.4 Embedding initial into final

Least fixed points can be embedded into greatest fixed points.

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

How to define *upcast*? We can write it as a fold …

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

3.4 Embedding initial into final

Least fixed points can be embedded into greatest fixed points.

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

How to define *upcast*? We can write it as a fold ...

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

3.4 Embedding initial into final

Least fixed points can be embedded into greatest fixed points.

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

How to define *upcast*? We can write it as a fold ...

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

3.4 Embedding initial into final

Least fixed points can be embedded into greatest fixed points.

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

How to define *upcast*? We can write it as a fold . . .

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

. . . or as an unfold:

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

3.4 Embedding initial into final

Least fixed points can be embedded into greatest fixed points.

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

How to define *upcast*? We can write it as a fold ...

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

... or as an unfold:

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

3.4 Embedding initial into final

Least fixed points can be embedded into greatest fixed points.

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

How to define *upcast*? We can write it as a fold ...

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

... or as an unfold:

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

3.4 Embedding initial into final

Least fixed points can be embedded into greatest fixed points.

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

How to define *upcast*? We can write it as a fold ...

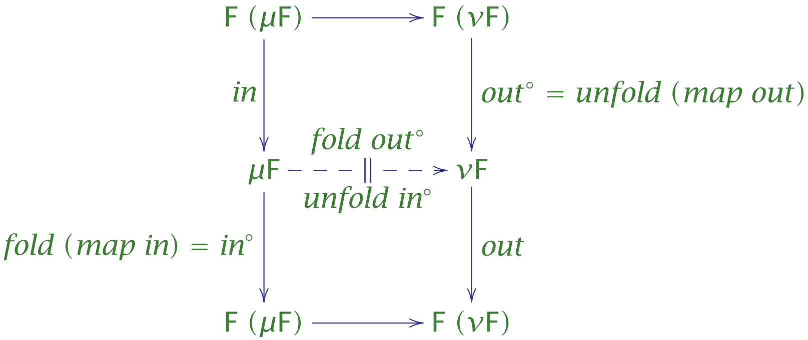
$$\begin{aligned} fold\ (unfold\ c) &: \mu \mathsf{F} \rightarrow \nu \mathsf{F} \\ unfold\ c &: \mathsf{F}\ (\nu \mathsf{F}) \rightarrow \nu \mathsf{F} \\ c &: \mathsf{F}\ (\nu \mathsf{F}) \rightarrow \mathsf{F}\ (\mathsf{F}\ (\nu \mathsf{F})) \end{aligned}$$

... or as an unfold:

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

Obvious candidates: $c = map\ out$ and $a = map\ in$.

The coalgebra *fold* (*map in*) is the inverse of *in*; the algebra *unfold* (*map out*) is the inverse of *out*. Moreover,



(The triples $\langle \mu F, in, in^\circ \rangle$ and $\langle \nu F, out^\circ, out \rangle$ are examples of *bialgebras*, more later.)

3.4 Intermediate summary

- *initial algebra*: syntax (finite trees)
- folds: replacing constructors by functions
- (denotational semantics: compositional valuation function that maps syntax to semantics—folding over syntax trees)
- *final coalgebra*: behaviour (finite and infinite trees)
- unfolds: tracing a state space
- (operational semantics: unfolding to transition trees)
- we have seen a glimpse of type-driven program development
- running time (assuming a *strict* setting):
  - *fold*: proportional to the size of the input
  - *unfold*: proportional to the size of the output (output-sensitive algorithm)

Section 4

Exchange sort

4.0 Back to sorting

A sorting function takes a list to an ordered list,

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

where $\nu \underline{List}$ is the datatype of ordered lists:

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

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

(No guarantees, we use $\underline{List}$ for emphasis.)

To define a sorting function let us follow a type-directed approach:

$$f :: \mu List \to \nu \underline{List}$$
$$f = unfold\ c$$

To define a sorting function let us follow a type-directed approach:

$$f :: \mu List \to \nu \underline{List}$$
$$f = unfold\ c$$

$$c :: \mu List \to \underline{List}\ (\mu List)$$
$$c = fold\ a$$

To define a sorting function let us follow a type-directed approach:

$$
\begin{aligned}
&f :: \mu List \to \nu \underline{List} \\
&f = unfold\ c
\end{aligned}
$$

$$
\begin{aligned}
&c :: \mu List \to \underline{List}\ (\mu List) \\
&c = fold\ a
\end{aligned}
$$

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

4.1 Bubble sort

We have re-invented bubble sort!

$$
\begin{aligned}
&\mathit{bubbleSort} :: \mu\mathit{List} \to \nu\underline{\mathit{List}}\\
&\mathit{bubbleSort} = \mathit{unfold}\ \mathit{bubble}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{bubble} :: \mu\mathit{List} \to \underline{\mathit{List}}\ (\mu\mathit{List})\\
&\mathit{bubble} = \mathit{fold}\ \mathit{bub}
\end{aligned}
$$

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

Dually, we can start with a fold:

$$f :: \mu List \to \nu \underline{List}$$
$$f = fold\ a$$

Dually, we can start with a fold:

$$f :: \mu List \to \nu \underline{List}$$
$$f = fold\ a$$

$$a :: List\ (\nu \underline{List}) \to \nu \underline{List}$$
$$a = unfold\ c$$

Dually, we can start with a fold:

$$f :: \mu List \to \nu \underline{List}$$
$$f = fold\ a$$

$$a :: List\ (\nu \underline{List}) \to \nu \underline{List}$$
$$a = unfold\ c$$

$$
\begin{aligned}
& c :: List\ (\nu \underline{List}) \to \underline{List}\ (List\ (\nu \underline{List})) \\
& c\ Nil = \underline{Nil} \\
& c\ (Cons\ x\ \lfloor \underline{Nil} \rfloor) = \underline{Cons}\ x\ Nil \\
& c\ (Cons\ x\ \lfloor \underline{Cons}\ y\ xs \rfloor) \\
& \quad \mid x \leqslant y = \underline{Cons}\ x\ (Cons\ y\ xs) \\
& \quad \mid otherwise = \underline{Cons}\ y\ (Cons\ x\ xs)
\end{aligned}
$$

4.2 Naïve insertion sort

We obtain a naïve variant of insertion sort!

$$
\begin{aligned}
&\mathit{naiveInsertionSort} :: \mu\mathit{List} \to \nu\underline{\mathit{List}}\\
&\mathit{naiveInsertionSort} = \mathit{fold}\ \mathit{naiveInsert}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{naiveInsert} :: \mathit{List}\ (\nu\underline{\mathit{List}}) \to \nu\underline{\mathit{List}}\\
&\mathit{naiveInsert} = \mathit{unfold}\ \mathit{naiveIns}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{naiveIns} :: \mathit{List}\ (\nu\underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\mathit{List}\ (\nu\underline{\mathit{List}}))\\
&\mathit{naiveIns}\ \mathit{Nil} && = \underline{\mathit{Nil}}\\
&\mathit{naiveIns}\ (\mathit{Cons}\ x\ \lfloor\underline{\mathit{Nil}}\rfloor) && = \underline{\mathit{Cons}}\ x\ \mathit{Nil}\\
&\mathit{naiveIns}\ (\mathit{Cons}\ x\ \lfloor\underline{\mathit{Cons}}\ y\ \mathit{xs}\rfloor)\\
&\quad \mid x \leqslant y && = \underline{\mathit{Cons}}\ x\ (\mathit{Cons}\ y\ \mathit{xs})\\
&\quad \mid \mathit{otherwise} && = \underline{\mathit{Cons}}\ y\ (\mathit{Cons}\ x\ \mathit{xs})
\end{aligned}
$$

Why naïve?

The algebra and the coalgebra are almost identical:

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

The algebra and the coalgebra are almost identical:

$$
\begin{array}{lcl}
a :: \mathit{List}\ (\underline{\mathit{List}}\ (\mu \mathit{List})) \to \underline{\mathit{List}}\ (\mu \mathit{List}) \\
a\ \mathit{Nil} & = & \underline{\mathit{Nil}} \\
a\ (\mathit{Cons}\ x\ \underline{\mathit{Nil}}) & = & \underline{\mathit{Cons}}\ x\ \lceil \mathit{Nil} \rceil \\
a\ (\mathit{Cons}\ x\ (\underline{\mathit{Cons}}\ y\ \mathit{xs})) \\
\quad \mid x \leqslant y & = & \underline{\mathit{Cons}}\ x\ \lceil \mathit{Cons}\ y\ \mathit{xs} \rceil \\
\quad \mid \mathit{otherwise} & = & \underline{\mathit{Cons}}\ y\ \lceil \mathit{Cons}\ x\ \mathit{xs} \rceil
\end{array}
$$

$$
\begin{array}{lcl}
c :: \mathit{List}\ (\nu \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\mathit{List}\ (\nu \underline{\mathit{List}})) \\
c\ \mathit{Nil} & = & \underline{\mathit{Nil}} \\
c\ (\mathit{Cons}\ x\ \lfloor \underline{\mathit{Nil}} \rfloor) & = & \underline{\mathit{Cons}}\ a\ \mathit{Nil} \\
c\ (\mathit{Cons}\ x\ \lfloor \underline{\mathit{Cons}}\ y\ \mathit{xs} \rfloor) \\
\quad \mid x \leqslant y & = & \underline{\mathit{Cons}}\ x\ (\mathit{Cons}\ y\ \mathit{xs}) \\
\quad \mid \mathit{otherwise} & = & \underline{\mathit{Cons}}\ y\ (\mathit{Cons}\ x\ \mathit{xs})
\end{array}
$$

The algebra and the coalgebra are almost identical:

$$\begin{array}{lll}
a :: \mathit{List}\ (\underline{\mathit{List}}\ (\mu \mathit{List})) \to \underline{\mathit{List}}\ (\mu \mathit{List}) & & \\
a\ \mathit{Nil} & = & \underline{\mathit{Nil}} \\
a\ (\mathit{Cons}\ x\ \underline{\mathit{Nil}}) & = & \underline{\mathit{Cons}}\ x\ \lceil \mathit{Nil} \rceil \\
a\ (\mathit{Cons}\ x\ (\underline{\mathit{Cons}}\ y\ xs)) & & \\
\quad \mid x \leqslant y & = & \underline{\mathit{Cons}}\ x\ \lceil \mathit{Cons}\ y\ xs \rceil \\
\quad \mid \mathit{otherwise} & = & \underline{\mathit{Cons}}\ y\ \lceil \mathit{Cons}\ x\ xs \rceil
\end{array}$$

$$\begin{array}{lll}
c :: \mathit{List}\ (\nu \underline{\mathit{List}}) \to \underline{\mathit{List}}\ (\mathit{List}\ (\nu \underline{\mathit{List}})) & & \\
c\ \mathit{Nil} & = & \underline{\mathit{Nil}} \\
c\ (\mathit{Cons}\ x\ \lfloor \underline{\mathit{Nil}} \rfloor) & = & \underline{\mathit{Cons}}\ a\ \mathit{Nil} \\
c\ (\mathit{Cons}\ x\ \lfloor \underline{\mathit{Cons}}\ y\ xs \rfloor) & & \\
\quad \mid x \leqslant y & = & \underline{\mathit{Cons}}\ x\ (\mathit{Cons}\ y\ xs) \\
\quad \mid \mathit{otherwise} & = & \underline{\mathit{Cons}}\ y\ (\mathit{Cons}\ x\ xs)
\end{array}$$

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

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

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

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

We can re-define bubble and naïve insertion sort using *swap*:

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

$$
\begin{aligned}
&naiveInsertionSort :: \mu List \to \nu \underline{List}\\
&naiveInsertionSort = fold\ (unfold\ (swap \cdot map\ out))
\end{aligned}
$$

In a sense, *swap* extracts the computational ‘essence’ of bubble and naïve insertion sorting.

bubble sort

initial input

2	4	1	3	$2 \leftrightarrow 1$	$4 \leftrightarrow 1$	$1 \leftrightarrow 3$
1	2	4	3		$2 \leftrightarrow 3$	$4 \leftrightarrow 3$
1	2	3	4			$3 \leftrightarrow 4$
1	2	3	4			
1	2	3	4			

output

---

naïve insertion sort

input

2	4	1	3			$1 \leftrightarrow 3$
2	4	1	3		$4 \leftrightarrow 1$	$4 \leftrightarrow 3$
2	4	1	3	$2 \leftrightarrow 1$	$2 \leftrightarrow 3$	$3 \leftrightarrow 4$
2	1	3	4			
1	2	3	4			

final output

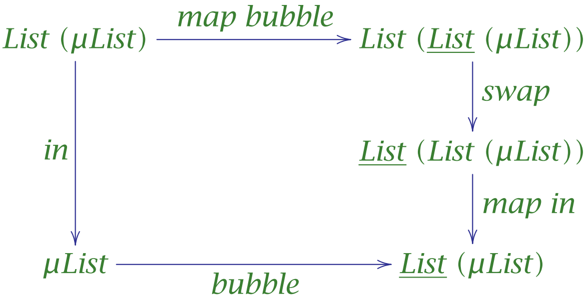
4.2 Intermediate summary

- *swap* exchanges adjacent elements
- *swap* is the computational essence of bubble sort and naïve insertion sort
- running time $\Theta(n^2)$
- how can we write true insertion sort?
- first: proof that *bubbleSort* and *naiveInsertionSort* are equal (in a strong sense)

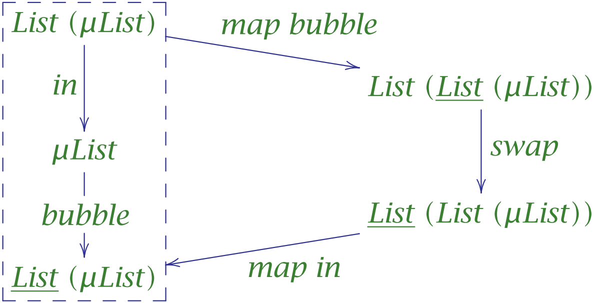
Section 5

Bialgebras and distributive laws

Recall that *bubble* is a *List*-algebra homomorphism.

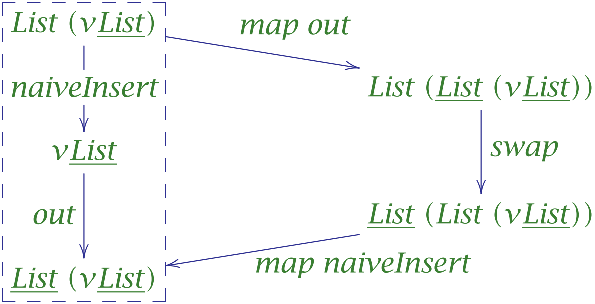


Let us rearrange the diagram.



The algebra *in* and the coalgebra *bubble* form a *swap*-bialgebra: $\langle \mu List, in, bubble \rangle$.

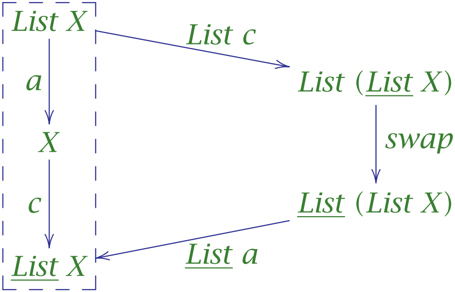
Recall that *naiveInsert* is a *List*-coalgebra homomorphism.



The algebra *naiveInsert* and the coalgebra *out* also form a *swap*-bialgebra: $\langle \nu List, naiveInsert, out \rangle$.

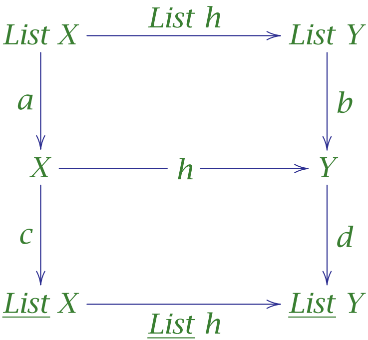
5.1 Bialgebra

For an algebra *a* and coalgebra *c* to be a *swap*-bialgebra, we must have that



5.2 Bialgebra homomorphism

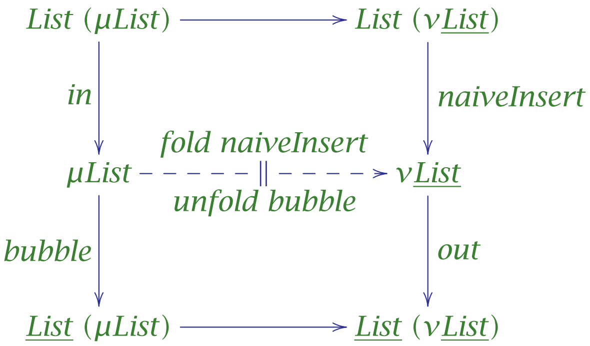
A *swap*-bialgebra homomorphism *h* is simultaneously an *List*-algebra and a *<u>List</u>*-coalgebra homomorphism.



swap-bialgebras and homomorphisms form a category.

5.2 Initial and final bialgebra

The *initial object* in this category is $\langle \mu List, in, bubble \rangle$; the *final object* is $\langle \nu \underline{List}, naiveInsert, out \rangle$.



By uniqueness, *naiveInsertionSort* and *bubbleSort* are equal.

5.2 Intermediate summary

- *swap* is a distributive law
- $\langle \mu List, in, bubble \rangle$ is the initial *swap*-bialgebra
- $\langle \nu \underline{List}, naiveInsert, out \rangle$ is the final *swap*-bialgebra
- bubble sort and naïve insertion sort are two (strongly related) variations of the same idea: repeatedly exchanging adjacent elements

Section 6

Insertion and selection sort II

- sorting algorithms as folds of unfolds or unfolds of folds necessarily have a running time of $\Theta(n^2)$
- to define insertion and selection sort, we need variants of folds and unfolds, so-called *para-* and *apomorphisms*

6.1 Paramorphism

- we start by defining products

$$\mathbf{data}\ a \times b = As\ \{ outl :: a, outr :: b \}$$

$$(\vartriangle) :: (c \to a) \to (c \to b) \to (c \to a \times b)$$
$$(f \vartriangle g)\ x = As\ (f\ x)\ (g\ x)$$

- we write $As\ a\ b$ as $a \mathbin{\text{≍}} b$ (we use it like Haskell's $a@b$).

6.1 Paramorphism

- we start by defining products

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

$$(\vartriangle) :: (c \to a) \to (c \to b) \to (c \to a \times b)$$
$$(f \vartriangle g)\ x = As\ (f\ x)\ (g\ x)$$

- we write $As\ a\ b$ as $a \asymp b$ (we use it like Haskell's $a@b$).
- we are now ready to define paramorphisms:

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

  a paramorphism also provides the intermediate input: the 'algebra' has type $f\ (\mu f \times a) \to a$ instead of $f\ a \to a$

- *slogan:* eats its argument and keeps it too

6.2 Apomorphism

- products dualise to sums

$$\mathbf{data}\ a + b = \mathit{Stop}\ a \mid \mathit{Play}\ b$$

$$(\triangledown) :: (a \to c) \to (b \to c) \to (a + b \to c)$$
$$(f \triangledown g)\ (\mathit{Stop}\ a) = f\ a$$
$$(f \triangledown g)\ (\mathit{Play}\ b) = g\ b$$

- we write $\mathit{Stop}\ a$ as $a\,\blacksquare$, and $\mathit{Play}\ b$ as $\blacktriangleright b$

6.2 Apomorphism

- products dualise to sums

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

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

- we write *Stop a* as $a\,\blacksquare$, and *Play b* as $\blacktriangleright b$
- paramorphisms dualise to apomorphisms:

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

  the corecursion is split into two branches, with *no recursive call* on the left
- *apomorphisms improve the running time*

With apomorphisms, we can write the insertion function as one that stops scanning after inserting an element:

$$\mathit{insertSort} :: \mu\mathit{List} \to \nu\underline{\mathit{List}}$$
$$\mathit{insertSort} = \mathit{fold}\ \mathit{insert}$$

With apomorphisms, we can write the insertion function as one that stops scanning after inserting an element:

$$insertSort :: \mu List \to \nu \underline{List}$$
$$insertSort = fold\ insert$$

$$insert :: List\ (\nu \underline{List}) \to \nu \underline{List}$$
$$insert = apo\ ins$$

With apomorphisms, we can write the insertion function as one that stops scanning after inserting an element:

$$
\begin{aligned}
&insertSort :: \mu List \to \nu \underline{List} \\
&insertSort = fold\ insert
\end{aligned}
$$

$$
\begin{aligned}
&insert :: List\ (\nu \underline{List}) \to \nu \underline{List} \\
&insert = apo\ ins
\end{aligned}
$$

$$
\begin{aligned}
&ins :: List\ (\nu \underline{List}) \to \underline{List}\ (\nu \underline{List} + List\ (\nu \underline{List})) \\
&ins\ Nil && = \underline{Nil} \\
&ins\ (Cons\ x\ \lfloor \underline{Nil} \rfloor) && = \underline{Cons}\ x\ (\lfloor \underline{Nil} \rfloor \blacktriangleleft) \\
&ins\ (Cons\ x\ \lfloor \underline{Cons}\ y\ xs \rfloor) \\
&\quad \mid x \leqslant y && = \underline{Cons}\ x\ (\lfloor \underline{Cons}\ y\ xs \rfloor \blacktriangleleft) \\
&\quad \mid otherwise && = \underline{Cons}\ y\ (\blacktriangleright (Cons\ x\ xs))
\end{aligned}
$$

From *ins* we can extract a natural transformation, which we call *swop* for *swap'n'stop*:

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

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

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

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

In general, a natural transformation such as *swop* gives rise to two algorithms. Algorithms for free!

6.3 Intermediate summary

- *apomorphisms improve the running time*
- running time of insertion sort: worst case still $\Theta(n^2)$, but best case $\Theta(n)$
- (paramorphisms don't improve the running time)
- the computational essence of insertion and selection sort is the natural transformation *swop*
- in general, we shall seek natural transformation of type

$$\mathsf{F}\ (A \times \mathsf{G}\ A) \to \mathsf{G}\ (A + \mathsf{F}\ A)$$

- (proof of equality involves (co-) pointed functors)

Section 7

Quicksort and treesort

- *so far:* one-phase sorting algorithms

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

- to improve performance we need to exchange non-adjacent elements
- *next:* two-phase sorting algorithms that make use of an intermediate data structure

$$\mu List \to \nu Tree \to \mu Tree \to \nu \underline{List}$$

- the intermediate data structure can sometimes be deforested (turning a data into a control structure)
- we can play our game for each phase

7.0 Search trees

- an obvious intermediate data structure is a *binary tree*

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

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

- we assume a ‘horizontal’ ordering

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

7.1 Phase one: growing a search tree

- the essence of growing a search tree

$$
\begin{aligned}
&\mathit{sprout} :: \mathit{List}\ (a \times \mathit{SearchTree}\ a) \rightarrow \mathit{SearchTree}\ (a + \mathit{List}\ a)\\
&\mathit{sprout}\ \mathit{Nil} && = \mathit{Empty}\\
&\mathit{sprout}\ (\mathit{Cons}\ x\ (t \asymp \mathit{Empty})) && = \mathit{Node}\ (t\,\blacksquare)\ x\ (t\,\blacksquare)\\
&\mathit{sprout}\ (\mathit{Cons}\ x\ (t \asymp (\mathit{Node}\ l\ y\ r)))\\
&\quad \mid x \leqslant y && = \mathit{Node}\ (\blacktriangleright (\mathit{Cons}\ x\ l))\ y\ (r\,\blacksquare)\\
&\quad \mid \mathit{otherwise} && = \mathit{Node}\ (l\,\blacksquare)\ y\ (\blacktriangleright (\mathit{Cons}\ x\ r))
\end{aligned}
$$

- this is the only sensible definition: *no choices*
- we compare elements across some distance

- 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

$$\begin{aligned} grow &:: \mu List \to \nu SearchTree \\ grow &= unfold\ (para\ (map\ (id \triangledown in) \cdot sprout)) \end{aligned}$$

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

- the algebra is a useful function on its own: insertion into a search tree
- *efficient insertion into a tree is necessarily an apomorphism*

7.2 Phase two: withering a search tree

- the essence of withering a search tree

$$
\begin{aligned}
&\mathit{wither} :: \mathit{SearchTree}\ (a \times \underline{\mathit{List}}\ a) \to \underline{\mathit{List}}\ (a + \mathit{SearchTree}\ a)\\
&\mathit{wither}\ \mathit{Empty}\\
&\quad = \underline{\mathit{Nil}}\\
&\mathit{wither}\ (\mathit{Node}\ (l \succeq \underline{\mathit{Nil}})\ x\ (r \succeq \_))\\
&\quad = \underline{\mathit{Cons}}\ x\ (r\ \blacksquare)\\
&\mathit{wither}\ (\mathit{Node}\ (l \succeq (\underline{\mathit{Cons}}\ x\ l'))\ y\ (r \succeq \_))\\
&\quad = \underline{\mathit{Cons}}\ x\ (\blacktriangleright (\mathit{Node}\ l'\ y\ r))
\end{aligned}
$$

- again, this is the only sensible definition

- this should surprise no one: the second phase would surely be an in-order traversal

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

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

- the algebra is essentially a ternary version of append
- the coalgebra deletes the leftmost element from a search tree

7.2 Putting things together

We obtain the famous *quicksort* and the less prominent *treesort* algorithms,

$$quickSort :: \mu List \to \nu \underline{List}$$
$$quickSort = flatten \cdot downcast \cdot grow$$

$$treeSort :: \mu List \to \nu \underline{List}$$
$$treeSort = flatten \cdot downcast \cdot grow'$$

where $downcast :: (Functor\ f) \Rightarrow \nu f \to \mu f$ projects the final coalgebra onto the initial algebra.

7.2 Intermediate summary

- once the intermediate data structure has been fixed, everything falls into place: *no choices*
- *observation:* only the first phase performs comparisons
- quicksort and treesort are are two (strongly related) variations of the same idea
- running time: worst case still $\Theta(n^2)$, but average case $\Theta(n \log n)$

Section 8

Heapsort and minglesort

8.0 Heaps

- a search tree imposes a horizontal ordering
- we can also assume a ‘vertical’ ordering

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

8.1 Phase one: piling up a heap

- the essence of piling up a heap

$$
\begin{aligned}
&\mathit{pile} :: \mathit{List}\ (a \times \mathit{Heap}\ a) \rightarrow \mathit{Heap}\ (a + \mathit{List}\ a)\\
&\mathit{pile}\ \mathit{Nil} && = \mathit{Empty}\\
&\mathit{pile}\ (\mathit{Cons}\ x\ (t \asymp \mathit{Empty})) && = \mathit{Node}\ (t\ \blacksquare)\ x\ (t\ \blacksquare)\\
&\mathit{pile}\ (\mathit{Cons}\ x\ (t \asymp (\mathit{Node}\ l\ y\ r)))\\
&\quad \mid x \leqslant y && = \mathit{Node}\ (\blacktriangleright (\mathit{Cons}\ y\ r))\ x\ (l\ \blacksquare)\\
&\quad \mid \mathit{otherwise} && = \mathit{Node}\ (\blacktriangleright (\mathit{Cons}\ x\ r))\ y\ (l\ \blacksquare)
\end{aligned}
$$

- *now we have a choice* (3rd equation)! Braun's trick!

8.1 Phase one: piling up a heap

- the essence of piling up a heap

$$
\begin{aligned}
&\mathit{pile} :: \mathit{List}\ (a \times \mathit{Heap}\ a) \to \mathit{Heap}\ (a + \mathit{List}\ a) \\
&\mathit{pile}\ \mathit{Nil} && = \mathit{Empty} \\
&\mathit{pile}\ (\mathit{Cons}\ x\ (t \asymp \mathit{Empty})) && = \mathit{Node}\ (t\, \blacksquare)\ x\ (t\, \blacksquare) \\
&\mathit{pile}\ (\mathit{Cons}\ x\ (t \asymp (\mathit{Node}\ l\ y\ r))) \\
&\quad \mid x \leqslant y && = \mathit{Node}\ (\blacktriangleright (\mathit{Cons}\ y\ r))\ x\ (l\, \blacksquare) \\
&\quad \mid \mathit{otherwise} && = \mathit{Node}\ (\blacktriangleright (\mathit{Cons}\ x\ r))\ y\ (l\, \blacksquare)
\end{aligned}
$$

- *now we have a choice* (3rd equation)! Braun's trick!
- let $a = x\ \text{`}\mathit{min}\text{`}\ y$ and $b = x\ \text{`}\mathit{max}\text{`}\ y$,

$$
\begin{aligned}
&= \mathit{Node}\ (\blacktriangleright (\mathit{Cons}\ b\ l))\ a\ (r\, \blacksquare) \\
&= \mathit{Node}\ (r\, \blacksquare)\ a\ (\blacktriangleright (\mathit{Cons}\ b\ l)) \\
&= \mathit{Node}\ (l\, \blacksquare)\ a\ (\blacktriangleright (\mathit{Cons}\ b\ r)) \\
&= \mathit{Node}\ (\blacktriangleright (\mathit{Cons}\ b\ r))\ a\ (l\, \blacksquare)
\end{aligned}
$$

- as usual we obtain two algorithms

$$
\begin{aligned}
&heapify :: \mu List \to \nu Heap \\
&heapify = unfold\ (para\ (map\ (id \triangledown in) \cdot pile))
\end{aligned}
$$

$$
\begin{aligned}
&heapify' :: \mu List \to \nu Heap \\
&heapify' = fold\ (apo\ (pile \cdot map\ (id \vartriangle out)))
\end{aligned}
$$

- the algebra is a useful function on its own: insertion into a heap

8.2 Phase two: sifting through a heap

- the essence of sifting through a heap

$$
\begin{aligned}
&\mathit{sift} :: \mathit{Heap}\ (a \times \underline{\mathit{List}}\ a) \to \underline{\mathit{List}}\ (a + \mathit{Heap}\ a)\\
&\mathit{sift}\ \mathit{Empty} && = \underline{\mathit{Nil}}\\
&\mathit{sift}\ (\mathit{Node}\ (l \asymp \underline{\mathit{Nil}})\ x\ (r \asymp \_)) && = \underline{\mathit{Cons}}\ x\ (r\ \blacksquare)\\
&\mathit{sift}\ (\mathit{Node}\ (l \asymp \_)\ x\ (r \asymp \underline{\mathit{Nil}})) && = \underline{\mathit{Cons}}\ x\ (l\ \blacksquare)\\
&\mathit{sift}\ (\mathit{Node}\ (l \asymp (\underline{\mathit{Cons}}\ y\ l'))\ x\ (r \asymp (\underline{\mathit{Cons}}\ z\ r')))\\
&\quad \mid y \leqslant z && = \underline{\mathit{Cons}}\ x\ (\blacktriangleright (\mathit{Node}\ l'\ y\ r))\\
&\quad \mid \mathit{otherwise} && = \underline{\mathit{Cons}}\ x\ (\blacktriangleright (\mathit{Node}\ l\ z\ r'))
\end{aligned}
$$

- when constructing the heap node to continue with, we have the option to swap left with right, but this buys us nothing

- again, we obtain two algorithms

$$unheapify :: \mu Heap \to \nu \underline{List}$$
$$unheapify = fold\ (apo\ (sift \cdot map\ (id \vartriangle out)))$$

$$unheapify' :: \mu Heap \to \nu \underline{List}$$
$$unheapify' = unfold\ (para\ (map\ (id \triangledown in) \cdot sift))$$

- the coalgebra deletes the mimimum element from a heap

8.2 Putting things together

- we obtain heapsort and a variant of heapsort that behaves suspiciously like mergesort

$$heapSort :: \mu List \to \nu \underline{List}$$
$$heapSort = unheapify \cdot downcast \cdot heapify$$

$$mingleSort :: \mu List \to \nu \underline{List}$$
$$mingleSort = unheapify' \cdot downcast \cdot heapify'$$

- trimerous mergesort, called minglesort, builds a heap by repeatedly dividing the input into three parts: two lists of balanced length along with the minimum element
- the merging phase is a similarly trimerous operation

8.2 Intermediate summary

- the intermediate data structure provides us with a choice (heaps are more flexible than search trees)
- *observation:* both phases use comparisons
- running time: worst case $\Theta(n \log n)$—insensitive to the input
- minglesort is a recent variant of heapsort (closely related to *Heap-Mergesort*, see Computer and Mathematics with Applications 39, 2000)
- an analogous approach using *binary leaf trees* works for true *mergesort*

Section 9

Epilogue

9.1 Summary

- *categorical algorithmics*
- type-driven algorithm design building on a few principled recursion operators
- few design decisions: intermediate data structures
- no rabbits!
- categorical duality gives us algorithms for free
- see WGP '12 paper: *Sorting with Bialgebras and Distributive Laws* by Ralf Hinze, Daniel W. H. James, Thomas Harper, Nicolas Wu and José Pedro Magalhães

9.2 Summary and future work

- computational essence
  - naïve insertion and bubble sort: *swap*
  - insertion and selection sort: *swop*
  - growing a search tree: *sprout*
  - withering a search tree: *wither*
  - piling up a heap: *pile*
  - sifting through a heap: *sift*
- *top-down algorithms:* regular datatypes
- *bottom-up algorithms:* nested datatypes