All sorts of permutations

Andrew Lelechenko
1@dxdy.ru

Papers We Love, Kyiv, 22.10.2016

All Sorts of Permutations (Functional Pearl)

Jan Christiansen
Flensburg University of Applied Sciences, Germany
jan.christiansen@hs-flensburg.de

Nikita Danilenko
University of Kiel, Germany
nda@informatik.uni-kiel.de

Sandra Dylus
University of Kiel, Germany
sad@informatik.uni-kiel.de

Abstract

The combination of non-determinism and sorting is mostly associated with permutation sort, a sorting algorithm that is not very useful for sorting and has an awful running time.

In this paper we look at the combination of non-determinism and sorting in a different light: given a sorting function, we apply it to a non-deterministic predicate to gain a function that enumerates permutations of the input list. We get to the bottom of necessary properties of the sorting algorithms and predicates in play as well as discuss variations of the modelled non-determinism.

On top of that, we formulate and prove a theorem stating that no matter which sorting function we use, the corresponding permutation function enumerates all permutations of the input list. We use free theorems, which are derived from the type of a function alone, to prove the statement.

Categories and Subject Descriptors D.1.4 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs

General Terms Languages, Algorithms

Keywords Haskell, monads, non-determinism, permutation, sorting, free theorems

Let us consider the following Haskell function[1]

$$\mathit{filterND} :: (\alpha \to \mathit{ND}\ \mathit{Bool}) \to [\alpha] \to \mathit{ND}\ [\alpha],$$

which is a non-deterministic extension of the well-known higher-order function *filter*. It is folklore knowledge that some non-deterministic extensions of predicate-based higher-order functions can be used to derive new non-deterministic functions by using a predicate that yields *True* and *False*. For example, when we apply *filterND* to the non-deterministic predicate

$$\begin{aligned}&\mathit{coinPredND} :: \alpha \to \mathit{ND}\ \mathit{Bool}\\&\mathit{coinPredND}\ \_ = [\mathit{True}, \mathit{False}]\end{aligned}$$

we get a function that non-deterministically enumerates all sublists of a given list.

Intuitively, when we apply *filterND* to *coinPredND* and a list xs, the resulting function non-deterministically chooses to keep or remove it from the result list for every element of xs. This decision is made for every element in the argument list independently, hence, we get all sublists of the argument list.

Similarly, this "trick" can be used to implement a function enumerating all permutations by sorting with a non-deterministic binary predicate. That is, for some non-deterministic version of a sorting algorithm

Notations

- $f :: \alpha \to \beta$ stands for a function, which consumes one argument of type α and returns a value of type β.
- $f :: \alpha_1 \to \alpha_2 \to \cdots \to \alpha_n \to \beta$ stands for a function, which consumes n arguments of types $\alpha_1, \alpha_2, \ldots, \alpha_n$ and returns a value of type β.
- $[\alpha]$ is a single-linked list of values of type α.
- $[]$ is an empty list.
- $(a : as)$ is a list with *head a* and *tail as*.
- a, b and c denote single values.
- as, bs and cs denote lists.

Deterministic sort

Typical sort function looks like

$$\mathsf{sortBy} :: (\alpha \to \alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

E. g.,

$$\begin{aligned}\mathsf{sortBy}\ (\leq)\ [3,4,1,2] &= [1,2,3,4],\\ \mathsf{sortBy}\ (\geq)\ [3,4,1,2] &= [4,3,2,1].\end{aligned}$$

Function $(\alpha \to \alpha \to \mathsf{Bool})$ is called a *comparator*.

Well-behaved comparators

- Consistency: the value of $a \preceq b$ is always the same.
- Reflexivity: $a \preceq a$.
- Antisymmetricity: $a \preceq b$ and $b \preceq a$ iff $a = b$.
- Transitivity: if $a \preceq b$ and $b \preceq c$, then $a \preceq c$.

Otherwise the output of sorting routine:

- may appear not to be linearly ordered.
- may appear not to be a permutation of the input.

What exactly goes wrong, when comparator is ill-behaved?

Non-deterministic sort

Let us model a non-deterministic comparator as a function, returning a (maybe empty) list of Bool, representing possible results of comparison. The ultimate example is an *uncertain* comparator, which never dares to compare anything:

$$\text{uncertainCmp} :: \alpha \to \alpha \to [\text{Bool}]$$
$$\text{uncertainCmp}\ a\ b = [\text{True}, \text{False}]$$

The usual *certain* comparator looks like

$$\text{certainCmp} :: \alpha \to \alpha \to [\text{Bool}]$$
$$\text{certainCmp}\ a\ b = [a \leq b]$$

Sorts and permutations

Each time, when a non-deterministic comparator returns multiple results, fork execution and combine outputs. Since sorting involves many comparisons, it results in a huge list of possible outcomes.

- Is each permutation of the input listed?
  Yes, every sorting algorithm that actually sorts can describe every possible permutation. If there is a permutation that cannot be realised by the sorting algorithm, then there is an input list that cannot be sorted.
- Is each permutation listed exactly once?
  It depends.
- Is any non-permutation listed?
  It depends.

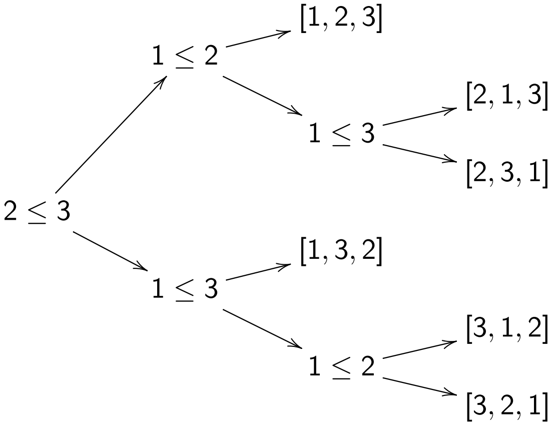
Any sorting algorithm may potentially result in a new algorithm for enumerating permutations!

Insertion sort — 1

```
insertSortBy :: forall μ α. Monad μ
             ⇒ (α → α → μ Bool) → [α] → μ [α]
insertSortBy cmp = insertSort
  where
    insertSort :: [α] → μ [α]
    insertSort []       = return []
    insertSort (a : as) = do
                          bs ← insertSort as
                          insert a bs

    insert :: α → [α] → μ [α]
    insert a []       = return [a]
    insert a (b : bs) = do
                        t ← a `cmp` b
                        if t then return (a : b : bs)
                             else (b :) <$> insert a bs
```

Insertion sort — 2

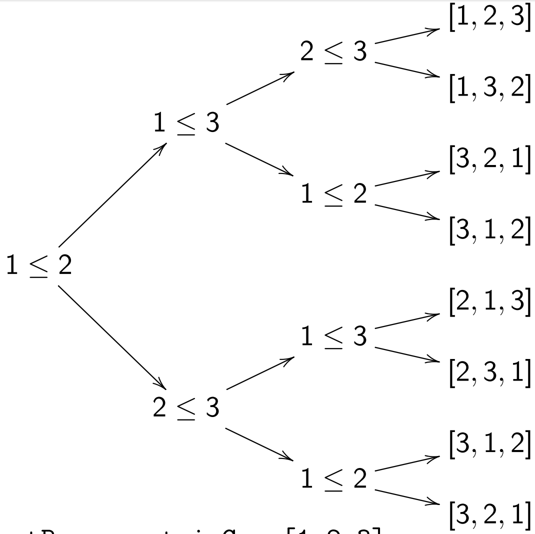



```
> insertSortBy uncertainCmp [1,2,3]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
```

Selection sort — 1

```
selectSortBy :: forall μ α. Monad μ
             ⇒ (α → α → μ Bool) → [α] → μ [α]
selectSortBy cmp = selectSort
  where
    selectSort :: [α] → μ [α]
    selectSort []       = return []
    selectSort (a : as) = do
                          (b, bs) ← selectMin a as
                          (b :) <$> selectSort bs
    selectMin :: α → [α] → μ (α, [α])
    selectMin a []       = return (a, [])
    selectMin a (b : bs) = do
                           t ← a `cmp` b
                           let (a', b') = if t then (a, b)
                                                else (b, a)
                           (c, cs) ← selectMin a' bs
                           return (c, b' : cs)
```

Selection sort — 2



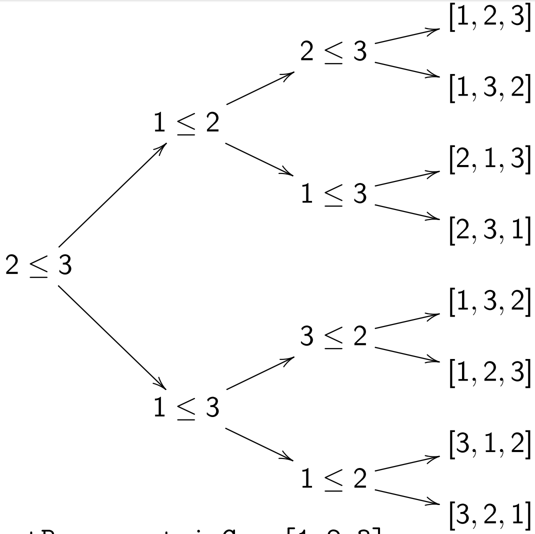

```
> selectSortBy uncertainCmp [1,2,3]
[[1,2,3],[1,3,2],[3,2,1],[3,1,2],
 [2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

Bubble sort — 1

```
bubbleSortBy :: forall μ α. Monad μ
             ⇒ (α → α → μ Bool) → [α] → μ [α]
bubbleSortBy cmp = bubbleSort
  where
    bubbleSort :: [α] → μ [α]
    bubbleSort []         = return []
    bubbleSort (a : as) = do
                            (b, bs) ← bubble a as
                            (b :) <$> bubbleSort bs

    bubble :: α → [α] → μ (α, [α])
    bubble a []         = return (a, [])
    bubble a (c : cs) = do
                          (b, bs) ← bubble c cs
                          t ← a `cmp` b
                          return $ if t then (a, b : bs)
                                        else (b, a : bs)
```

Bubble sort — 2




```
> bubbleSortBy uncertainCmp [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],
 [1,3,2],[1,2,3],[3,1,2],[3,2,1]]
```

Quicksort — 1

```
quickSortBy :: forall μ α. Monad μ
            ⇒ (α → α → μ Bool) → [α] → μ [α]
quickSortBy cmp = quickSort
  where
    quickSort :: [α] → μ [α]
    quickSort []       = return []
    quickSort (a : as) = do
                  (bs, cs) ← partitionBy (‘cmp‘ a) as
                  xs ← quickSort bs
                  ys ← quickSort cs
                  return $ xs ++ (a : ys)
```

```
partitionBy :: Monad μ
            ⇒ (α → μ Bool) → [α] → μ ([α], [α])
partitionBy predicate = partition
  where
    partition []       = return ([], [])
    partition (a : as) = do
                         (bs, cs) ← partition as
                         t ← predicate a
                         return $ if t then (a : bs, cs)
                                       else (bs, a : cs)
```

```
> quickSortBy uncertainCmp [1,2,3]
[[3,2,1],[2,3,1],[3,1,2],[2,1,3],[1,3,2],[1,2,3]]
```

Mergesort — 1

```
mergeSortBy :: forall μ α. Monad μ
            ⇒ (α → α → μ Bool) → [α] → μ [α]
mergeSortBy cmp = mergeSort
  where
    mergeSort :: [α] → μ [α]
    mergeSort []  = return []
    mergeSort [a] = return [a]
    mergeSort as  = do
                    let l = length as `div` 2
                    let (bs, cs) = splitAt l as
                    xs ← mergeSort bs
                    ys ← mergeSort cs
                    merge xs ys
```

Mergesort — 2

```
merge :: [α] → [α] → μ [α]
merge as       []       = return as
merge []       bs       = return bs
merge (a : as) (b : bs) = do
                 t ← a `cmp` b
                 if t then (a :) <$> merge as (b : bs)
                      else (b :) <$> merge (a : as) bs
```

```
> mergeSortBy uncertainCmp [1,2,3]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
```

Summary

- Insertion sort, quicksort and mergesort enumerates permutations precisely.
- Selection sort enumerates permutations precisely under assumption of consistency.
- Bubble sort enumerates permutations precisely under assumption of antisymmericity.
- Rare algorithms require transitivity for precise enumeration (namely, patience sort by Mallows).
- Rare algorithms produce not only permutations (namely, two-pass quicksort).
- Insertion sort, quicksort and mergesort can be transformed into algorithms for enumeration of permutations.
- But beware to use them as a random permutation generator!

Thank you!