Generic Sorting Multiset Discriminators

How to sort complex data in linear time

Fritz Henglein

Department of Computer Science
University of Copenhagen
Email: henglein@diku.dk

Outline

Outline

Outline

Standard recipe

1. For each (first-order) type T, define a *standard order* by induction on type denotation T. Denote standard order by term r or think of T itself as a denotation of the order.
2. Define a *generic comparison function/inequality test* (characteristic function of order) *compositionally* on standard order/type denotation.
3. Choose a good comparison-based sorting algorithm, say randomized Quicksort.
4. Define generic sorting function by *applying* sorting algorithm to generically defined comparison function.
5. Result: a function that takes a standard order denotation as (possibly implicit) input and returns a sorting function for that standard order.

Standard recipe: Observations

- It is the *comparison function* that is generically defined.
- The sorting algorithm is *not* generically *defined*: it is *parametric*in the comparison functions.
- Since definition of comparison function is compositional, standard order denotations need not be explicit. They can be given providing record of *combinators* instead.

Generic sorting with type classes

1. Define type class (Ord t). Designate name of function to be defined generically (compare).
2. Provide instance declarations, which are individual clauses of the compositional definition.
3. Ask compiler to extend to recursively defined functions over recursively defined types by employing “deriving” construct.
4. Then define sorting function parametrically from generically defined comparison function:

```
sort ::  (Ord t) => [t] -> [t]
```

Questions

1. Do we only ever want *at most one* order per type? What about sorting pairs in ascending order on first component and descending order on second components? On first components only (and with higher-order values in second component)? On the first four letters of the elements only?
2. Do we need or want explicit denotations instead of providing a record of the composition functions only?
3. How to deal with recursively defined types?
4. Why define the comparison function generically and then use comparison-based sorting, which *only* provides access to the comparison function of a type instead of defining sorting generically directly?
5. Can you sort generically in linear time?

Orders

Definition (Total preorder)

A *total preorder (order)* $(T, \leq)$ is a type T together with a binary relation $\leq \subseteq T \times T$ that is reflexive, transitive and total.

Order denotations

See order.hs

Generic definition of comparison function

See inequality.hs

Generic definition of sorting function

We can try to define sorting functions directly generically:

```
dsort :: Order k -> [k] -> [k]
```

Imagine now we want to define the case for Pair r1 r2:

```
sort (Pair r1 r2) xs = ... sort r1 ... sort
                        r2 ...
```

How to do this?
Equivalently, how can we define a combinator for sorting pairs given only sorting functions for the first and second components, respectively?

```
sortPair :: ([t1]->[t1) -> ([t2]->[t2]) ->
            [(t1, t2)] -> [(t1, t2)]
sortPair s1 s2 xs = ... s1 ... s2 ...
```

Generic definition sorting

We can sort the individual components by themselves using s1 or s2, but this does not help us much since we will then need to reassociate the sorted component values with their associated other component values.

Conclusion: We should generalize the type of sort to sort elements according to a *part* of the elements. Call this part the *key* of the element and the remaining part its associated *value* and the whole element the *record* to be sorted. (Indeed this is the original formulation of the sorting problem.)

Dicriminative sorting

See sort.hs

Discriminative sorting: Observation 1

- Each part of a key, once used for sorting is returned as part of the output, but never used (inspected/destructed) again as part of the sorting algorithm.
- Keys that are sorted on often need to be discarded from the output in the recursive calls.

Idea 1: Return only values, not keys, as part of output. Amounts to “sorting the value according to the keys”.

Discriminative sorting: Observation 2

- Sorting of pairs is right-to-left: Sort records according to right component first. Then sort result according to left component.
  - Requires a stable sorting function to be correct.
  - Consider when used to sort list-elements: Inspects all parts of (almost) all keys, not just minimal distinguishing prefix.
- Left-to-right sorting requires knowing which elements are equivalent according to left component.

Idea 2: Return equivalence classes, not just individual elements, in sorted order.

Order-preserving discrimination

See disc.hs

Discriminator combinators

See disccomb.hs

Explicit denotations versus combinators

Same for both:

- There may be any number (0, 1 or more) of denotable orders at a given type.
- Any which order may be denoted by *multiple* denotations (combinator expressions); e.g. `Inv (Sum r1 r2)` and `sum2 (Inv r1) (Inv r2)`.
- Since algorithms are defined by induction on denotations, different denotations (combinator expressions) give different algorithms.
- Denotations (combinator expressions) can be used to "control" which algorithm is generated.

Explicit denotations versus combinators

Differences:

- Transformations of denotations to semantically equivalent denotations may be used to optimize algorithms:

```
optimize :: Order(t) -> Order(t) optimize
Char = Char ...
fdisc r xs = disc (optimize r) xs
```

  This requires reasoning about terms of type `Order(k)` (explicit denotations) versus `[(k, v)] -> [[v]]`. Since `Order` has an elimination form (definition by cases), the former is programmable in the object language, the latter not.

Applications

See discapps.hs

Classical sorting algorithms

- Quicksort
- Mergesort
- Heapsort
- Insertion sort
- Bubble sort
- Bitonic sort
- Shell sort
- Zero-one mergesort
- AKS sorting network
- Bucket sort
- Radix/lexicographic sort
- ...

Myths and facts

Everybody knows: Sorting requires $O(n \log n)$. $O(n \log n)$ what? And does it really require that? Facts:

1. Given abstract total preorder (order) $(T, \leq)$, any sorting algorithm requires $\Omega(m \log m)$ applications of the comparison operator $\leq$ to sort an input of m elements of type T.
2. There exist algorithms that, given any $(T, \leq)$, sort m inputs using $O(m \log m)$ applications of the comparison operator.
3. Fact 1 does not imply that sorting requires $\Omega(n \log n)$ time where n is the size of the input. $O(n)$ sorting algorithms for a large number of concrete orders exist (remainder of talk).
4. Fact 2 does not imply that those algorithms necessarily execute in worst-case time $O(n \log n)$ for non-constant size input elements. None of them do.

Time complexities reconsidered

Assume $(T, \leq)$ such that time complexity of executing $x \leq y$ is $\Theta(|x| + |y|)$. Input: $[x_1, \ldots, x_m]$ of size $n = \Sigma_{i=1}^{n} |x_i|$.

- Quicksort: $\Theta(n^2)$ ($O(n \log n)$ randomized?!)
- Mergesort: $\Theta(n^2)$
- Heapsort: $\Theta(n^2)$
- Selection sort: $\Theta(n^3)$
- Insertion sort: $\Theta(n^2)$
- Bubble sort: $\Theta(n^2)$
- Bitonic sort: $\Theta(n \log^2 n)$
- Shell sort: $\Theta(n \log^2 n)$
- Zero-one mergesort: $\Theta(n \log^2 n)$
- AKS sorting network: $O(n \log n)$ (uniformly constructible?)
- Bucket/counting sort: not comparison-based
- Radix/lexicographic sort: not comparison-based

Time complexities reconsidered

Proof ideas:

1. Consider one element of size $\Theta(n)$, the rest of size $O(1)$. How many comparisons performed on that one element?
2. Algorithm as sorting network: Maximum depth is upper bound on number of comparisons on each element.

Complexity of discrimination

Theorem (Top-down MSD)

For each canonical `r: Order(t)` *the discriminator* `disc r` *executes in worst-case linear time on it (unboxed size) input.*

- Canonical r: Standard order denotation, canoncially.
- Theorem also holds under Bag and Set equivalences.
- Linearity for top-down MSD only holds for unshared (unboxed) data (sequences, not lists with shared tails; trees, not dags).
- Linear time performance can be achieved for shared, acyclic data using bottom-up MSD.
- $O(n \log n)$ performance can be achieved for shared, cyclic data (using different algorithmic strategy).

Performance

Performance

- No algorithm engineering in the code!
- Need to understand not only Haskell, but compiler to figure out practical performance.
- Quite competitive vis a vis Quicksort in terms of time; sometimes much better, e.g. small distinguishing prefix in input.
- Distributive sorting is known to be problematic in terms of space consumption vis a vis comparison-based sorting algorithms.

Conclusion and perspectives

- Generic discrimination: Solves paritioning and sorting in one go in linear time.
- With a linear-time discriminator as primitive function for observing equality at an abstract type partitioning can be solved in linear time as opposed to quadratic time, when only given an equality test.
- GADTs, System F (rank 2) types and list comprehensions have been pleasant for specifying discrimination.