

What is a sort function?

Fritz Henglein

Department of Computer Science, University of Copenhagen (DIKU)
henglein@diku.dk

September 22nd, 2007

Total preorders Sorting is one of the most-studied subjects of computer science. So it seems silly to ask what a sort function is.¹ Obviously a sort function is a function that permutes sequences such that the elements are in order. This, however, begs the next question: What does it mean to be “in order”? An obvious answer is that the output must respect the ordering relation of a *given* total order.² So this suggests the following definition: Given total order (S, \leq_S) , a sort function is a permutation function $sort : S^* \rightarrow S^*$ such that if $y_1 \dots y_n = sort(x_1 \dots x_n)$ then $y_i \leq_S y_{i+1}$ for $1 \leq i \leq n$. We can quickly see, however, that expecting the sort function to output according to a *total order* is too much to expect of it; for example when sorting tuples “according to their i -th component” as a subroutine in radix-sort, this step permutes the whole tuples (records), but according to a total order on a particular component (key), not the whole tuple. Expecting keys to be totally ordered is still too much: If the keys are strings we may want to sort the records according to their keys, but *ignoring* their case. This means that the output may contain both $[("fReD", 15), ("FrEd", 18)]$ and $[("FrEd", 18), ("fReD", 15)]$, even though this is clearly incompatible with requiring the key domain to be totally ordered. It is the anti-symmetry requirement of total orders that is too strong. Since reflexivity, transitivity and totality seem reasonable requirements, we drop antisymmetry:

A *total preorder* or (here) just *order* (S, R) is a set S together with a binary relation $R \subseteq S \times S$ that is *reflexive*, *transitive*, and *total* ($\forall x, y \in S : (x, y) \in R \vee (y, x) \in R$). We say R is an *ordering relation* on S and (S, R) is an *order on S* .

Note that an order (S, R) canonically induces an equivalence relation $(S, \cong_R) : x \cong_R y \Leftrightarrow R(x, y) \wedge R(y, x)$.

We can now define a sort function to be a function on an order that permutes its input such that the output respects the ordering relation:

A function $sort : S^* \rightarrow S^*$ is a *sort function for order* (S, R) if for all $x_1 \dots x_n \in S^*$ and $y_1 \dots y_m = sort(x_1 \dots x_n)$ we have: $y_1 \dots y_m$ is a permutation of $x_1 \dots x_n$ (and so $n = m$); and $y_1 \dots y_m$ is R -ordered: $R(y_i, y_{i+1})$ for all $1 \leq i \leq m$.

Comparators Now we know what a sort function is for a *given* order. But that was not the question: there was no mention of a given order. So what if somebody hands you a function and claims it is a sort function—without giving you an order?

An obvious answer is: It has to be a sort function for *some* order. But then follow-up questions beg themselves: Is there any order at all? If so, what if there are several? Does it *define* an order in some sense? The answer to the first question is trivial: Each permutation function $f : S^* \rightarrow S^*$ is a sort function for the trivial order $(S, S \times S)$, which relates all elements to each other. It is least informative, however, since it is least discriminative

¹The use of *function* is intended to convey that we are considering the mathematical transformation on the data, whether executed in-place, out-of-place and independent of data structure representation.

²Since “sorting” in ordinary usage only implies collecting related—equivalent—elements, essentially *partitioning*, a more accurate term would have been “ordering” for “sorting”. See Knuth [Knu98] for a humorous discussion of this. Note also the use in this sense more accurate usage of *sort* in multi-sorted algebra.

amongst candidate orders for which f is a sort function: all elements of S are equivalent to each other under ordering relation $S \times S$.

Consider f applied to two arguments: If $f[x_1, x_2] = [x_1, x_2]$ we can conclude that $x_1 \leq x_2$ for any ordering relation \leq consistent with f as a sort function. If $f[x_1, x_2] = [x_2, x_1]$ we can conclude $x_2 \leq x_1$, but we do not know whether or not $x_1 \leq x_2$ holds. Clearly, however, the unique *smallest* and thus most informative *possible* ordering relation \leq^{min} consistent with f is the one that lets us conclude $x_1 \not\leq^{min} x_2$ whenever $f[x_1, x_2] = [x_2, x_1]$ or, equivalently, $x_1 \leq^{min} x_2 \Leftrightarrow f[x_1, x_2] = [x_1, x_2]$, but only if this defines an *ordering* relation. It turns out that it does if and only if f as a function of two arguments is *permutative*, *transitive* and *idempotent*.

A *comparator structure* $(S, comp)$ is a set S together with a function $comp : S \times S \rightarrow S \times S$ that is *permutative*: $comp(x, y) = (x, y) \vee comp(x, y) = (y, x)$; *transitive*: $comp(x, y) = (x, y) \wedge comp(y, z) = (y, z) \implies comp(x, z) = (x, z)$; and *idempotent*: $comp(x, y) = (y, x) \implies comp(y, x) = (y, x)$ for all $x, y, z \in S$. We call $comp$ a *comparator (function)* on S .

Theorem 1: Consider $f : S \times S \rightarrow S \times S$ and define binary relation R on S by $R(x_1, x_2) \Leftrightarrow f(x_1, x_2) = (x_1, x_2)$. Then R is an ordering relation on S if and only if f is a comparator on S .

Sort functions A sort function $sort : S^* \rightarrow S^*$ for order (S, R) is *stable* if R -equivalent elements occur in the same order in the output as in the input: $sort(\vec{x})|_{[z]_{\cong_R}} = \vec{x}|_{[z]_{\cong_R}}$ for all $z \in S$ and $\vec{x} \in S^*$, where $[z]_{\cong_R}$ denotes the set of R -equivalent S -elements of z .³ Each order has exactly one stable sort function since stability fixes the order in which equivalent elements must be output.

Note that stipulating that the ordering relation we are after satisfy $x_1 \leq x_2 \Leftrightarrow f[x_1, x_2] = [x_1, x_2]$, as we did for comparators, is tantamount to insisting that f be a *stable* sort function for that order, at least when applied to two elements. So insisting that one's proclaimed sort function is *stable* (for arbitrary length inputs) is a way of identifying the most informative order consistent with f . It is easy to see that, if f is a stable sort function for any ordering relation at all, then that ordering relation is unique. As it turns out, f is a stable sort function for some order if and only if it is *consistently permutative*.

Theorem 2: Let $f : S^* \rightarrow S^*$. There exists an ordering relation R on S such that f is a stable sort function for (S, R) if and only if f is *consistently permutative*: For each sequence $x_1 \dots x_n \in S^*$ there exists permutation $\pi \in S_{|\vec{x}|}$ such that

- $f(x_1 \dots x_n) = x_{\pi(1)} \dots x_{\pi(n)}$ (permutativity);
- $\forall i, j \in [1 \dots n] : f(x_i x_j) = x_i x_j \Leftrightarrow \pi^{-1}(i) \leq_\omega \pi^{-1}(j)$ (consistency).

Furthermore, if R exists, it is uniquely determined by f : $R(x_1, x_2) \Leftrightarrow f[x_1, x_2] = [x_1, x_2]$.

The permutation π in the definition of consistent permutativity can be thought of as mapping the *rank* of an (occurrence of an) element—where it occurs in the output of *sort*—to its *index*—where it occurs in the input. The inverse permutation π^{-1} thus maps the index of an element occurrence to its rank. Consistency expresses that the relative order of two elements in the output of *sort* must always be the same.

Now we know what a sort function is: any consistently permutative function! Finding such an *intrinsic* characterization of when a function is a (stable) sort function for some order is surprisingly tricky. Several plausible candidates turn out to be almost correct, but only almost [Hen07].

Isomorphisms Theorems 1 and 2 show that the categories of orders, comparator structures and sort structures are isomorphic where the morphisms are all the set-theoretic functions

³We write $\vec{x}|_P$ for the subsequence of \vec{x} made up of all elements from P . Defining stability is notoriously tricky to do correctly.

between the underlying sets; that is, the morphisms ignore the structure. The question then is: Are they isomorphic under some notion of structure-preserving morphisms; and if so, under which notion?

Loosely speaking, monotonic and order-mapping functions on orders correspond to *sort-preserving* functions on sort structures. See Henglein [Hen07].

Observing orders Imagine we want to implement an abstract data type and export a function that makes an ordering relation \leq on its element set S observable, but nothing else.⁴

The most obvious representation is by exporting a *comparison (function)*, the characteristic function $lte : S \times S \rightarrow Bool$ of (S, \leq) : $\forall x, y \in S : lte(x, y) = true \Leftrightarrow x \leq y$. There are alternatives, however. Theorems 1 and 2 show that an ordering relation can be made observable (and nothing else about the data type) by exporting a comparator or sort function. Each of the three possible exported functions *defines* an order once their respective intrinsic characteristics are verified, and, given any one, the other ones are parametrically polymorphically definable [Hen07].

So does it matter, whether comparison, comparator or sort function are implemented “natively” and subsequently exported? Comparison provides information on the ordering relation for only two elements at a time. As a well-known corollary, any comparison-based sorting algorithm requires $\Omega(n \log n)$ applications of comparison to sort n elements [Knu98, Section 5.3.1]. The same holds true for comparators. In contrast, *distributive* sorting algorithms [Knu98, Section 5.2.5] such as radix-sort run in linear time. Interestingly, such distributive algorithms can be defined and extended *generically* to arbitrary first-order types while preserving their linear-time performance [Hen06]. In other words: It is possible to implement a time-efficient sort function for a new data type if we have access to time-efficient *sort functions* for the (ordered) types used in its implementation. If instead only comparison functions (or comparators for that purpose) are available we are back to the comparison-based sorting bottleneck.

Conclusion Starting with a seemingly innocuous question—What is a sort function?—we have arrived at a maybe surprising answer: Any consistently permutative function. In doing so we have flipped the preeminence of orders over sort functions on the head by showing that it is not necessary to be given a (presentation) of an order to define (what it means to be) a sort function. Comparisons, comparators and sort functions are parametrically polymorphically interdefinable. So conceptually and behaviorally they are interchangeable. From a performance point of view, there is a difference, however, depending on which one we implement “natively”: Sort functions admit construction of generic distributive sorting algorithms that are not subject to the comparison-based sorting bottleneck.

References

- [Hen06] Fritz Henglein. Generic discrimination: Partitioning and sorting complex data in linear time. TOPPS Report D-559, Department of Computer Science, University of Copenhagen (DIKU), December 2006. <http://diku.dk/forskning/topps/bibliography>.
- [Hen07] Fritz Henglein. Intrinsic definition of sorting functions. TOPPS Report D-565, Department of Computer Science, University of Copenhagen (DIKU), April 2007. <http://diku.dk/forskning/topps/bibliography>.
- [Knu98] Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

⁴We shall implicitly assume that the data type also has an effective equality test. This is not necessary if we adopt a different notion of sort function where keys can carry associated values (“payload”).