

What is a Sorting Function? [★]

Fritz Henglein

*Department of Computer Science, University of Copenhagen (DIKU)
Universitetsparken 1, DK-2100 Copenhagen, Denmark. Email: henglein@diku.dk*

Abstract

What is a sorting function—not a sorting function for a given ordering relation, but a sorting function with *nothing* given?

Formulating four basic properties of sorting algorithms as defining requirements, we arrive at intrinsic notions of sorting and stable sorting: A function is a sorting function if and only if it is an intrinsically parametric permutation function. It is a stable sorting function if and only if it is an intrinsically stable permutation function.

We show that ordering relations can be represented isomorphically as inequality tests, comparators and stable sorting functions, each with their own intrinsic characterizations, which in turn provide a basis for run-time monitoring of their expected I/O behaviors. The isomorphisms are parametrically polymorphically definable, which shows that it is sufficient to provide any one of the representations since the others are derivable without compromising data abstraction.

Finally we point out that stable sorting functions as default representations of ordering relations have the advantage of permitting linear-time sorting algorithms; inequality tests forfeit this possibility.

Key words: sorting, sort function, sorting function, sorting algorithm, stable, permutation, comparator, inequality test, isomorphism, parametric, parametricity.

[★] This work has been partially supported by the Danish Natural Science Research Council (Forskningråd for Natur og Univers) under Project *Applications and Principles of Programming Languages (APPL)*. This is a preprint of: Fritz Henglein, What is a sorting function?, J. Log. Algebr. Program.(2009), doi:10.1016/j.jlap.2008.12.003

1 Introduction

Sorting is one of the most-studied subjects of computer science. So it seems silly to ask “what a sorting function is”. What makes the question interesting is that it asks without reference to a given order and seeks an answer in terms of I/O-behavior only. The use of *function* is intended to convey that we consider the mathematical input-output transformation expressed by sorting algorithms, whether executed in-place, out-of-place and independent of data structure representation, and without regard to their computational complexity.

1.1 Overview

In Section 2 we identify four central properties sorting algorithms share.

Sections 3 to 7 formulate the properties as functional requirements and develop the notions of parametric and stable permutation functions as the models of sorting function and stable sorting functions, respectively. We also give a local characterization of stable permutation functions, which can be used in monitoring the calls to a function at run time for checking that it is permutative and stable.

Section 9 shows that there are isomorphic ways of providing an operation that gives access to the ordering relation (and no more) of an ordered data type: inequality tests, comparators and stable sorting functions, where each has a first-order characterization.

We conclude in Section 10 with a summary, extensions, related work and a discussion of why sorting functions should not be considered subsidiary to inequality tests for providing access to ordered data types.

1.2 Prerequisites

In view of the fundamental and basic nature of the subject matter of the paper the presentation is self-contained and elementary. Knowledge of some sorting algorithms, basic set theory, fundamental mathematical notions and acquaintance with computability theory are sufficient. We shall draw on Reynolds’ Parametricity Theory [Rey83,Wad89], but introduce what is relevant here in a self-contained fashion.

Since much of the insights lie in which assumptions play a role where, full

proofs for most statements are included.

1.3 Notational conventions

This subsection contains a brief review of basic notions and notations that will be used later.

1.3.1 Orders

An *total preorder* (S, O) is a set S together with a binary relation $O \subseteq S \times S$ that is

- *transitive*: $\forall x, y, z \in S : (x, y) \in O \wedge (y, z) \in O \implies (x, z) \in O$; and
- *total*: $\forall x, y \in S : (x, y) \in O \vee (y, x) \in O$

We say O is an *ordering relation* on S . A total preorder canonically induces an equivalence relation $(S, \equiv_O): x \equiv_O y \iff O(x, y) \wedge O(y, x)$.

(S, O) is a *total order* if it is a total preorder and O is also

- *antisymmetric*: $\forall x, y \in S : (x, y) \in O \wedge (y, x) \in O \implies x = y$.

We shall be careful to write *total order*, without the “pre”, whenever the underlying ordering relation is antisymmetric. Otherwise, whenever we write “order” we mean a *total preorder*.

Ordering relations are usually denoted by O in prefix notation, $O(x, y)$, and by \leq in infix notation, $x \leq y$, with subscripted and primed variants. We write $x < y$ if $x \leq y$, but not $y \leq x$.

The natural numbers (including 0) \mathbb{N}_0 with the standard order on numbers, henceforth denoted \leq_ω here, is a total order. The natural numbers with \leq_k where $k > 0$ and $x \leq_k y \iff x \bmod k \leq_\omega y \bmod k$ defines a total preorder; it is not a total order.

The product relation O on $\mathbb{N}_0 \times \mathbb{N}_0$ defined by $((x, y), (x', y')) \in O \iff x \leq_\omega x' \wedge y \leq_\omega y'$ does not define an ordering relation since it is not total. However,

$$(x, y) \leq_{fst} (x', y') \iff x \leq_\omega x'$$

and the *lexicographic* orders

$$(x, y) \leq_{x_1} (x', y') \iff x \leq_\omega x' \wedge (x' \leq_\omega x \implies y \leq_\omega y')$$

and

$$(x, y) \leq_{\times_2} (x', y') \iff y \leq_k \omega y' \wedge (y' \leq_k y \Rightarrow x \leq_\omega x')$$

are examples of three different ordering relations on $\mathbb{N}_0 \times \mathbb{N}_0$, none of which are antisymmetric.

1.3.2 Sequences and permutations

For any set S we write S^* for the set of all finite sequences with elements from S . We write a sequence by juxtaposition of elements: $x_1x_2 \dots x_n$; or using brackets around a comma-separated enumeration of the elements: $[x_1, x_2, \dots, x_n]$. We write $|\vec{x}|$ for n , the length of $\vec{x} = x_1 \dots x_n$.

Given a subset $P \subseteq S$, we write $\vec{x}|_P$ for the subsequence of all elements $x_i \in \vec{x}$ such that $x_i \in P$; e.g., with $Even = \{x \mid (x = 0) \bmod 2\}$, the even numbers, we have $[1, 5, 6, 7, 0, 4, 8, 5]|_{Even} = [6, 0, 4, 8]$.

The *Symmetric Group* \mathcal{S}_n is the set of bijective functions on $[1 \dots n]$, where the group operation is functional composition. We use \mathcal{S}_n to refer to both the group and the underlying set of bijective functions. An element of \mathcal{S}_n is called a *permutation (on n)*. It is denoted by a rearrangement of the integer segment $[1 \dots n]$; e.g. $\pi_0 = (3\ 1\ 2)$ is an element of \mathcal{S}_3 that maps 1 to 3, 2 to 1 and 3 to 2. A permutation on n can be *applied* to or *acts* on any sequence of length n ; e.g. $\pi_0 [44, \text{“blob”}, \text{true}] = [\text{true}, 44, \text{“blob”}]$. We say a sequence \vec{y} is a *permutation of* sequence \vec{x} if \vec{y} is the result of a permutation acting on \vec{x} .

2 Properties of sorting algorithms

Cormen, Leiserson, Rivest and Stein define the *sorting problem* in their algorithms textbook as follows:

Input: A sequence of n numbers $\langle a_1, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq \dots \leq a'_n$.

The input sequence is usually an n -element array, although it may be represented in some other fashion, such as a linked list.

[...]” [CLRS01, p. 123]

Some questions immediately pose themselves: Can you only sort numbers? What about strings? Sets? Trees? Graphs?

Knuth defines sorting as “the arrangement of items into ascending or descending order” [Knu98, p. 1] and then expounds:

“The goal of sorting is to determine a permutation $p(1)p(2)\dots p(N)$ of the indices $\{1, 2, \dots, N\}$ that will put the keys into nondecreasing order [.]” [Knu98, p. 4]

He allows arbitrary data items, not just numbers, to be sorted, but implies, just as in Cormen, Leiserson, Rivest and Stein’s formulation, that the data come equipped with a fixed order.¹ But does the order always have to be the same? How about sorting strings in reverse lexicographic order or some user-defined order invented by a programmer for whatever purpose? Is that *not* sorting? And what actually are *orders*? Do they have to be total orders?

Note also the subtle difference in the formulations of what the *functionality* of sorting is: Knuth expects the result of sorting n inputs to be a *permutation* $\pi \in S_n$, not the *result* of applying π to the input, as Cormen, Leiserson, Rivest and Stein do.

In this section we identify basic properties of the I/O behavior of sorting algorithms without unduly restricting the notion of sorting itself to particular record types, data structures for representing sequences, or ways of specifying sorting criteria.

2.1 *Functionality: Permuted elements vs. permuted indexes*

What is the basic functionality of a sorting algorithm? In Knuth’s formulation it is a permutation of the *indexes* of the input sequence; in Cormen, Leiserson, Rivest and Stein’s it is a permutation of the *elements* themselves. That is not the same: Given input [“foo”, “bar”, “foo”] there are two distinct *permutations*² that put the input elements into ascending dictionary order: $\pi_1 = (213)$ and $\pi_2 = (231)$. The result of *applying* either one of them is the same, however: $\pi_1[\text{“foo”}, \text{“bar”}, \text{“foo”}] = \pi_2[\text{“foo”}, \text{“bar”}, \text{“foo”}] = [\text{“bar”}, \text{“foo”}, \text{“foo”}]$. The permuted indexes determine the permuted elements, but not conversely.

Returning the permuted indexes instead of the permuted elements is a not a severe functional requirement, but it is not trivial, either: The coding of the actual permutation requires extra space, and space is a premium resource in sorting. Indeed, most sorting algorithms do not make the permutation they construct *observable*, only the permuted elements. We can thus observe the following property, relating to the basic functionality implemented by a sorting

¹ Note that “ascending or descending” in the first quote has turned into “nondecreasing” in the second quote, however.

² Recall that a permutation in itself is an element of S_n for some n ; that is, a rearrangement of the integer segment $[1 \dots n]$.

algorithm.

Property 1: A sorting algorithm permutes its input: it transforms an input sequence into a rearranged sequence containing the same elements.

An algorithm that sorts according to a total order, say strings in dictionary order, may be stable or unstable. That is not an *observable* property, however: If its output is [“bar”, “foo”, “foo”] we cannot tell the difference between the two occurrences of “foo”. If the algorithm sorts the strings paired with their index in the input instead, we can observe the difference and recover the permutation computed by the algorithm from the result, however: Given [(“foo”, 1), (“bar”, 2), (“foo”, 3)] the result is either [(“bar”, 2), (“foo”, 1), (“foo”, 3)] or [(“bar”, 2), (“foo”, 3), (“foo”, 1)]. The former is a *stable* sort of the input, whereas the latter is unstable. Note that the ordering relation on input elements is not antisymmetric here: two observably different input elements, in our case (“foo”, 1) and (“foo”, 3) are *equivalent* for sorting purposes, but clearly not *equal*.

2.2 Sorting criteria: Key orders, total preorders

A sorting algorithm permutes input sequences such that the output satisfies a certain criterion: its elements have to be in some specified order. This, however, begs the question: What does it mean to be “in order” and how do we specify that? A deceptively obvious answer is that the output must respect the ordering relation of a *total order*.

Property 2, first attempt: The output of a sorting algorithm is totally ordered.

We can quickly see, however, that expecting a sorting algorithm to output its input elements according to a *total order* is too much to expect: most sorting algorithms do not do so. For example a “distribution sort based on the *least significant digit of the keys*” [Knu98, p. 170] in radix-sort permutes whole records, but according to a total order on the keys, not the whole records. Antisymmetry does not hold then: from $r_1 \leq r_2$ and $r_2 \leq r_1$ for input records r_1, r_2 we cannot conclude that the records themselves are actually *equal*, $r_1 = r_2$.

Indeed, in classical formulations of sorting the input is a sequence of *records* that are to be rearranged such that the records obey a given total order on their respective *keys*.

“We are given N items to be sorted; we shall call them *records* [...]. Each record R_j has a key, K_j , which governs the sorting process.” [Knu98, p. 4]

Allowing keys to be computed from records by arbitrary functions, not just projections, a sorting criterion can be specified by a *key order*.

Definition 2.1 [Key order] A *key order* for set S is a pair consisting of a total order (K, \leq_K) , and a function $key : S \rightarrow K$. \square

We call the elements of S *records* and those of K *keys*.

Property 2, second attempt: The output of a sorting algorithm obeys a key order $((K, \leq_K), key : S \rightarrow K)$, which serves as its *sorting criterion*: If $y_1 y_2 \dots y_n \in S^*$ is output then $key(y_i) \leq_K key(y_{i+1})$ for all $1 \leq i <_\omega n$.

Specifying sorting criteria as key orders is arguably too *concrete* in the sense that there are many different key orders that all are equivalent as sorting criteria: a sorting algorithm that sorts according to one also sorts according to the other and *vice versa*. Consider for example the case where we want to sort strings, but ignoring the case of characters. This sorting criterion can be specified as the standard dictionary ordering on strings together with the function that maps characters to upper case or the function that maps them to lower case serving as key function; or, indeed, using numerous other key orders each yielding the *same* result in the sense of being completely interchangeable for the purposes of sorting.

Definition 2.2 [Equivalent key orders] We say $((K_1, \leq_1), key_1 : S \rightarrow K_1)$ and $((K_2, \leq_2), key_2 : S \rightarrow K_2)$ are *sorting-equivalent key orders* if all algorithms that sorts according to one key order also sorts according to the other. \square

We observe that sorting-equivalent key orders induce the *same* relation on the records that the key functions are applied to.

Proposition 2.3 Let $((K_1, \leq_1), key_1 : S \rightarrow K_1)$ and $((K_2, \leq_2), key_2 : S \rightarrow K_2)$ be key orders. They are sorting-equivalent if and only if $key_1(x) \leq_1 key_1(y) \Leftrightarrow key_2(x) \leq_2 key_2(y)$ for all $x, y \in S$.

In other words, the only role a key function plays with respect to sorting is specifying the relation $r_1 \leq_S r_2 \Leftrightarrow key(r_1) \leq_K key(r_2)$ on S . Any two equivalent key functions will induce the same relation. That relation is not a total order, but it is always a *total preorder*.

Moving from key orders to total preorders is a way of making the sorting criteria more abstract—equivalent key orders correspond to the same total preorder—without, conceptually, going beyond them: Each key order induces a total preorder, and, conversely, conceptually each total preorder can be represented as a key order: choose a unique representative from each equivalence

class and map all the elements of an equivalence class to its representative.³

We can now formulate the final version of the ordering property of a sorting algorithm.

Property 2: The output of a sorting algorithm obeys a total preorder (S, O) : For all its outputs $y_1 y_2 \dots y_n \in S^*$ we have $O(y_i, y_{i+1})$ for all $1 \leq_\omega i <_\omega n$.

2.3 Obliviousness

Sorting algorithms manipulate *records* to be sorted, but only their *keys* are *inspected*.

“Additional data, besides the key, is usually also present: this extra ‘satellite information’ has no effect on sorting except that it must be carried along as part of each record.” [Knu98, p. 4]

“Each record contains a *key*, which is the value to be sorted [sic!], and the remainder of the record consists of *satellite data*, which are usually carried around with the key. In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.” [CLRS01, p. 123]

Note that the above is stated before even the first sorting algorithm is presented! From this alone we can deduce an important property of sorting algorithms: The fact that satellite data are “carried around” with the key implies that they are only copied or moved without being inspected. The sorting algorithm is *oblivious* to satellite data, as complexity theoreticians would say, or, as semanticists will put it, satellite data are treated *parametrically*.

Property 3: A sorting algorithm only copies and moves satellite data without inspecting them.

In terms of a total preorder (S, O) serving as sorting criterion this means that sorting algorithms operate on \equiv_O -equivalent input elements in the “same way”. We shall make this precise shortly.

³ This is not the same as factoring out the equivalence relation of the total preorder to arrive at a total order of equivalence classes to be sorted since we still want to sort observably different elements. Factoring out equivalence classes is a way of making elements indistinguishable.

2.4 Stability

For some applications it is important that equivalent input elements—e.g. records with the same key—be returned in the same order as in the input. For example, in least-significant-digit (LSD) radix sorting individual sorting steps must be stable for the whole result to be correctly sorted. A common application of this is interactive multicriterion sorting. Consider, e.g., a set of email messages. To obtain a listing grouped by sender and, for each sender, the messages in date order, it is sufficient to first sort all messages by date and then by name, but only if the sorting algorithm employed in the second step is stable.

A final property that some, but not all sorting algorithms thus have is *stability*.

Property 4: A stable sorting algorithm returns equivalent elements in the same relative order as they appear in the input.

2.5 Summary

Let us summarize what we have found out about general properties of sorting algorithms. A sorting algorithm:

- (1) operates on sequences of records and permutes them;
- (2) outputs the input records according to a given total preorder;
- (3) is oblivious to satellite data in the input records;
- (4) if it is required to do so, outputs order-equivalent elements in the same relative order as they occur in the input.

These are not properties of specific sorting algorithms, but of their observable behavior. Even the obliviousness property can be formulated as an extensional property using Reynolds' notion of relational parametricity [Rey83,Wad89].)

So we can give a first, tentative answer to our fundamental question “What is a sorting function?”: It is the I/O behavior of any algorithm with the above properties. The answer has a problem, however: It only says what a sorting function is for a *given* total preorder. But the question is “What is a sorting function?” (subsidiarily “What is stable sorting function?”) with no *given* total preorder to hold the function up against. Indeed our intent is to reverse the roles: Given a function, does it define a total preorder and, if so, which? If it does, is it a function that exhibits the above properties of a sorting algorithm? If it does not, how do we know?

In the following we shall formulate the properties as order-independent extensional requirements on functions and develop corresponding classes of functions satisfying the requirements.

3 Permutation functions

Imagine somebody hands you a function f and claims it to be a sorting function, but without giving you an order. How can you tell? What are the characteristic properties of a sorting function? A rather obvious minimum requirement is that it be a permutation function.

Definition 3.1 [Permutation function] A function f is a *permutation function* if $f : S^* \rightarrow S^*$ and $f(\vec{x})$ is a permutation of \vec{x} for all $\vec{x} \in S$. \square

Requirement 1: f must be a permutation function.

Does it have to have any other properties? An obvious answer is: It has to be consistent with an ordering relation.

Definition 3.2 [Permutation function consistent with ordering relation] Let $f : S^* \rightarrow S^*$ be a permutation function. We say f and ordering relation O on S are *consistent* with each other if for all $y_1 y_2 \dots y_n = f(x_1 x_2 \dots x_n)$ we have $O(y_i, y_{i+1})$ for all $1 \leq_\omega i <_\omega n$. \square

Requirement 2: f must be consistent with some ordering relation.

But then follow-up questions beg themselves: Is there any order at all? If so, what if there are several such orders? Is there a *canonical* order in some reasonable sense? The answer to the first question is disappointingly trivial: Each permutation function $f : S^* \rightarrow S^*$ is consistent with $S \times S$, the ordering relation on S that relates all elements to each other. That ordering relation is *least informative*, however, since it is least discriminative amongst candidate relations that are consistent with f : all elements of S are equivalent to each other under ordering relation $S \times S$. Maybe f has a *most discriminative* ordering relation? The good news is that it does: ordering relations are not generally closed under intersection, but those consistent with a given permutation function are.

Definition 3.3 [Canonically induced ordering relation] Let $f : S^* \rightarrow S^*$ be a permutation function. We call O the *ordering relation canonically induced* by f if

- (1) f is consistent with O and
- (2) for all O' consistent with f we have $O \subseteq O'$.

We write \leq_f for O and \equiv_f for the equivalence relation induced by \leq_f . \square

Proposition 3.4 \leq_f exists and is unique. Furthermore,

$$x \leq_f x' \Leftrightarrow \exists \vec{y} : \vec{y} \in \text{Range}(f). \vec{y} = \dots x \dots x' \dots;$$

that is, x occurs to the left of x' in the output of f for some input \vec{x} .

PROOF Straightforward. \square

The bad news is that we, in general, have no handle on the canonically induced ordering relation: A permutation function f will always put its input into order according to \leq_f , but it does not give us an effective way of *deciding* the ordering relation it induces:

Theorem 3.5 (Undecidability of canonically induced ordering relations)

There exists a total, computable permutation function $f : \mathbb{N}_0^ \rightarrow \mathbb{N}_0^*$ such that its canonically induced ordering relation is recursively undecidable.*

PROOF (Sketch) Define f as follows: Let x_1, \dots, x_n be (Gödel numbers denoting) Turing Machine configurations. Run each x_i for at most n steps. Output the x_i in the following order: List those that terminate, whether they reach a terminal state or a stuck state, within n steps first; do so according to the number of steps they have taken, with those taking *most* steps listed first. Thereafter list the Turing Machines that have not terminated within n steps, list them in any order, e.g., according to their Gödel number. Let x_T be a terminated configuration. Observe that $x \leq_f x_T$ if and only if x terminates, which is recursively undecidable. \square

We have seen that each permutation function f has a unique least ordering relation \leq_f according to which it orders its output. But do we want to accept a permutation function as a *sorting function*, even if it does not provide a way of deciding its own ordering relation?

4 Locally consistent permutation functions

Consider permutation function f applied to two elements in either order, $f(x_1x_2)$ and $f(x_2x_1)$. If either one yields the sequence x_1x_2 we can think of this as *constructive evidence* for $x_1 \leq_f x_2$.

Definition 4.1 [O_f] Define $O_f(x_1, x_2) \iff f(x_1x_2) = x_1x_2 \vee f(x_2x_1) = x_1x_2$. \square

Proposition 4.2 *Let f be a permutation function. Then $O_f \subseteq \leq_f$.*

PROOF Immediate from Proposition 3.4. □

But what should we conclude if $(x_1, x_2) \notin O_f$, that is both $f(x_1x_2)$ and $f(x_2x_1)$ yield x_2x_1 ? We would *like* to conclude that $x_2 \not\leq_f x_1$, but, as we have seen, that may be wrong in general since x_2 may occur before x_1 in the output of an input sequence containing x_1 and x_2 together with some *other* input elements.

We shall call a permutation function locally consistent if the conclusion $x_2 \not\leq_f x_1$ holds.

Definition 4.3 [Locally consistent permutation function] A permutation function is *locally consistent* if $\leq_f \subseteq O_f$. □

Local consistence captures the notion of effective definability of \leq_f from f : Given f and two values x, y , how can we figure out whether $x \leq_f y$? The only thing we can do is apply f to the permutations of (x, y) —there are only two—and see what comes out of it. If the output in both cases is the same, say $[x, y]$, we conclude not only $x \leq_f y$ but also $y \not\leq_f x$. In particular, we can use f to decide whether x and y are equivalent under \equiv_f , the equivalence induced by \leq_f .

Definition 4.4 [Q_f] Define

$$Q_f(x_1, x_2) \iff \left(\begin{array}{c} (f(x_1x_2) = x_1x_2 \wedge f(x_2x_1) = x_2x_1) \\ \vee \\ (f(x_1x_2) = x_2x_1 \wedge f(x_2x_1) = x_1x_2) \end{array} \right)$$

□

Proposition 4.5 *Let f be a locally consistent permutation function. Then*

$$x_1 \equiv_f x_2 \iff Q_f(x_1, x_2).$$

PROOF By rewriting the definition of O_f . □

Equivalently, $x_1 \equiv_f x_2$ holds if and only if f^2 acts as the identity on $[x_1, x_2]$ and on $[x_2, x_1]$.

5 Parametric permutation functions

To express the idea of an algorithm “not looking at certain parts” of its input data as a property of the *function* it implements we recall some of the notions of *parametricity* from Reynolds [Rey83] and Wadler [Wad89] and apply them to our setting.

Definition 5.1 [Logical relation] A (*binary*) *logical relation* R over given binary relations R_i and sets S_j is any relation denotable by the following relational expressions:

$$\begin{aligned}
R_i & \\
Id_{S_j} &= \{(x, x) \mid x \in S_j\} \\
R \rightarrow R' &= \{(f, g) \mid \forall (x, y) \in R. (f(x), g(y)) \in R'\} \\
R \times R' &= \{((x, y), (x', y')) \mid (x, x') \in R \wedge (y, y') \in R'\} \\
R^* &= \{(x_1 \dots x_m, y_1 \dots y_n) \mid m = n \wedge (\forall i \in \{1 \dots n\}. (x_i, y_i) \in R)\} \\
\forall X \subseteq R'. R[X] &= \bigcap_{X \subseteq R'} R[X]
\end{aligned}$$

where R, R' denote logical relation expressions and $R[X]$ denotes any logical relation expression with possible free occurrences of a variable X ranging over all binary relations considered as primitive logical relations. We write $f : R$ if $(f, f) \in R$ and call R a *parametricity property* of f . We say R *respects* R' if $R \subseteq R'$. \square

In the following we may use infix notation; e.g. $[x_1, \dots, x_n] R^* [y_1, \dots, y_n]$ for $([x_1, \dots, x_n], [y_1, \dots, y_n]) \in R^*$.

We can now formulate the obliviousness property of sorting algorithms as a parametricity property.

Definition 5.2 [Parametric permutation function] A permutation function $f : S^* \rightarrow S^*$ is (*intrinsically*) *parametric* if it preserves all relations that respect \equiv_f :

$$f : \forall R \subseteq \equiv_f. R^* \rightarrow R^*$$

\square

Requirement 3: f must be parametric.

Informally, this says that f does not distinguish between \equiv_f -equivalent elements during its execution—it cannot “see” the difference between them. Putting it operationally, an implementation of f performs the same execution

steps when given equivalent inputs.

Interestingly, we have the following theorem:

Theorem 5.3 (Parametricity implies locality) *Every intrinsically parametric permutation function is locally consistent.*

In other words, if a parametric permutation function has an output where y occurs to the right of x no matter how long the input sequence, it already does so when applied to one of the 2-element sequences $[x, y]$ and $[y, x]$.

PROOF Let $x, y \in S$ be arbitrary. Then one of the three relations hold: $x <_f y$, $x \equiv_f y$ or $x >_f y$. In each case we show that the corresponding relation on O_f (Definition 4.1) also holds.

- (1) $x <_f y$: In this case we have $y \not\leq_f x$, which means y never occurs to the left of x in an output of f (Proposition 3.4). Consequently we must have $f(xy) = f(yx) = xy$ and $O_f(x, y)$ by Definition 4.1.
- (2) $x \equiv_f y$: Define $R = \{(x, y), (y, x)\}$. Note that R respects \equiv_f since $x \equiv_f y$. There are two cases now: either $f(xy) = xy$ or $f(xy) = yx$.
 - (a) $f(xy) = xy$: By parametricity of f we have that $f(yx) = yx$ since the result $f(yx)$ must be R^* -related to xy . Consequently, we can conclude both $O_f(x, y)$ and $O_f(y, x)$.
 - (b) $f(xy) = yx$: By parametricity of f we have $f(yx) = xy$ and again we can conclude both $O_f(x, y)$ and $O_f(y, x)$.
- (3) Analogous to case 1.

□

Corollary 5.4 (Characterization of parametricity) *A permutation function $f : S^* \rightarrow S^*$ is parametric if and only if it preserves all relations respecting Q_f :*

$$f : \forall R \subseteq Q_f. R^* \rightarrow R^*.$$

PROOF Follows from Theorem 5.3 and Proposition 4.5. □

6 Comparison-based and key-based sorting algorithms

Our claim now is that the I/O behavior of a sorting algorithm is not just a permutation function, but a *parametric* permutation function and, conversely, that the notion of parametric permutation function captures what it means to be a sorting function. This provides the first answer to the question in the title of this paper:

Being a “sorting function” means being an intrinsically parametric permutation function.

Note that being parametric does not refer to any predefined ordering relation. It is an intrinsic property of a permutation function: preserving all relations respecting \equiv_f , a relation defined in terms of f .

In this section we validate the claim that sorting algorithms act as parametric permutation functions. We show that all *comparison-based* and *key-based* (specifically distributive) sorting algorithms satisfy parametricity. In each case we do this in two steps: First we argue that the algorithm has a certain parametricity property reflecting its oblivious treatment of satellite data, then we prove that that parametricity property implies intrinsic parametricity of the permutation function implemented.

6.1 Comparison-based sorting algorithms

A comparison-based sorting algorithm works by applying inequality tests on its input records.⁴

Definition 6.1 [Inequality test] Function $lte : S \times S \rightarrow Bool$ is an *inequality test* if it is the characteristic function of an ordering relation \leq_{lte} on S . \square

Apart from comparing, copying and moving the records or pointers to them around, a comparison-based sorting algorithm has no other operations on records. Examples are Quicksort, Mergesort, Heapsort, Shell sort, Insertion sort, Selection sort, Bubble sort, etc., and all their variants.

We can think of a comparison-based sorting algorithm as a function that is first passed an inequality test lte over the element type S and that then returns a function that does the actual sorting: when given a sequence of S -elements it uses the inequality test and no other operations on elements of S to put them into an order consistent with lte .

Definition 6.2 [Comparison-parameterized function] A function $F : (S \times S \rightarrow Bool) \rightarrow S^* \rightarrow S^*$ is a *comparison-parameterized function* if it has the following parametricity property:

$$F : \forall R \subseteq S \times S. (R \times R \rightarrow Id_{Bool}) \rightarrow R^* \rightarrow R^*.$$

\square

⁴ We can assume that the inequality test operates on complete records since an inequality test lte_k on their keys induces an inequality test on records: $lte_r(x, y) = lte_k(key(x), key(y))$.

Definition 6.3 [Comparison-based sorting function] A function f is a *comparison-based sorting function for ordering relation O* if there exists a comparison-parameterized function $F : (S \times S \rightarrow \text{Bool}) \rightarrow S^* \rightarrow S^*$ such that $f = F(\text{lte})$ where $\text{lte} : S \times S \rightarrow \text{Bool}$ is the inequality test for O and f is a permutation function consistent with O . \square

The parametricity property in Definition 6.2 captures that the algorithm performs the same steps if we replace input elements by R -related elements and keeps them in corresponding positions throughout the computation as long as the inequality test returns the same result for pairwise R -related arguments. Wadler has observed that this parametricity property implies that every comparison-based sorting function commutes with applying an order-mapping function to the elements of the input sequence [Wad89]. A comparison-parameterized function does not necessarily generate a sorting function, though. It does so only when applied to an inequality test.

We now show that a comparison-based sorting function for O is a parametric permutation function whose canonically induced ordering relation is O .

Theorem 6.4 *Let $f : S^* \rightarrow S^*$ be a comparison-based sorting function for O . Then:*

- (1) f is a parametric permutation function.
- (2) $O = \leq_f$

PROOF Let $f = F(\text{lte})$, where lte is the characteristic function of O , as in Definition 6.3.

- (1) By definition, f is a permutation function. We need to show that f is intrinsically parametric.

Let R be a relation such that $R \subseteq \equiv_f$. We claim that $\text{lte} : R \times R \rightarrow \text{Id}_{\text{Bool}}$. To prove this we need to show $O(x, y) \Leftrightarrow O(x', y')$ whenever $R(x, x')$ and $R(y, y')$. (Recall that lte is the characteristic function of O .) Assume $R(x, x')$ and $R(y, y')$. From $R(x, x')$ and $R \subseteq \equiv_f$ we can conclude $x \equiv_f x'$. Since f is consistent with O and \leq_f is, by definition, the smallest relation consistent with f we get that $x \equiv_O x'$. Analogously we obtain $y \equiv_O y'$.

$O(x, y) \Rightarrow O(x', y')$: Assume $O(x, y)$. We have $x' \equiv_O x \leq_O y \equiv_O y'$, writing \leq_O in infix notation for O . By transitivity of O we obtain $x' \leq_O y'$.

$O(x', y') \Rightarrow O(x, y)$: By symmetry.

This proves our claim.

We can now apply the parametricity property of F (Definition 6.2) and conclude that $F(\text{lte}) : R^* \rightarrow R^*$. Since $f = F(\text{lte})$ and R was chosen to be an arbitrary subset of \equiv_f we can conclude $f : \forall R \subseteq \equiv_f . R^* \rightarrow R^*$ and

thus that f is intrinsically parametric (Definition 5.2).

- (2) Since f is consistent with O , by definition of \leq_f we know that $\leq_f \subseteq O$. We need to prove $O \subseteq \leq_f$.

Recall the definition of Q_f : $Q_f(x, y) \Leftrightarrow f[x, y] = [x, y] \vee f[y, x] = [x, y]$. We claim $\forall x, y. O(x, y) \Rightarrow Q_f(x, y)$. We prove this by contradiction. Assume there exist distinct x, y such $O(x, y)$ and $f[x, y] = [y, x] \wedge f[y, x] = [y, x]$. Since the output of f is consistent with O this implies that $O(y, x)$ holds. Combined with the assumption $O(x, y)$ we have $x \equiv_O y$. This means that $\text{lte}(x, y) = \text{lte}(y, x)$. Choosing $R_{x,y} = \{(x, y), (y, x)\}$ we have the following parametricity property: $\text{lte} : R_{x,y} \times R_{x,y} \rightarrow \text{Id}_{\text{Bool}}$. By the parametricity property of F we obtain $f : R_{x,y}^* \rightarrow R_{x,y}^*$. This implies that if $f[x, y] = [y, x]$ then $f[y, x] = [x, y]$ and if $f[y, x] = [y, x]$ then $f[x, y] = [x, y]$. But this contradicts our assumption $f[x, y] = [y, x] \wedge f[y, x] = [y, x]$, which proves the claim.

By Proposition 4.2 we have $Q_f(x, y) \Rightarrow x \leq_f y$ and so $O(x, y) \Rightarrow Q_f(x, y) \Rightarrow x \leq_f y$, and we are done.

□

Observe that parametricity in no way depends on an algorithm being stable. It only reflects an algorithm being oblivious to satellite data.

Example 6.5 Consider an unstable sorting algorithm; e.g., Quicksort with a deterministic pivot rule such as taking the middle element or the median of the first, middle and last element. Consider sorting strings associated with their index in the input sequence according to the standard dictionary order on the strings. As noted in Section 2.1, associating the indexes or indeed any other pairwise distinct values with the input strings is a way of making the permutation computed by the algorithm observable. The algorithm may sort $[("foo", 1), ("bar", 2), ("bar", 3)]$ into $[("bar", 3), ("bar", 2), ("foo", 1)]$ and $[("bar", 2), ("bar", 3)]$ into $[("bar", 2), ("bar", 3)]$. Note that it is unstable on the first input sequence, but stable on the second input sequence. Since pairs with the same string component are equivalent by its parametricity property the algorithm then maps $[("foo", 1), ("bar", 3), ("bar", 2)]$ to $[("bar", 2), ("bar", 3), ("foo", 1)]$ and $[("bar", 3), ("bar", 2)]$ to $[("bar", 3), ("bar", 2)]$. □

6.2 Key-based sorting algorithms

We can think of a key-based sorting algorithm as first being passed a *key function* that maps S to a totally ordered set of *keys*. The algorithm is allowed to operate on keys in any way possible—e.g., treating them as bit strings and applying unrestricted bit operations— but, as for comparison-based sorting algorithms, is only allowed to copy or move records as a whole without

inspecting their satellite data. Examples of key-based algorithms are the distributive sorting algorithms Bucketsort and Radixsort.

Definition 6.6 [Key-parameterized function] A function $G : (S \rightarrow K) \rightarrow S^* \rightarrow S^*$ is a *key-parameterized function* if it has the following parametricity property:

$$G : \forall R \subseteq S \times S. (R \rightarrow Id_K) \rightarrow R^* \rightarrow R^*.$$

□

Definition 6.7 [Key-based sorting function] A function $g : S^* \rightarrow S^*$ is a *key-based sorting function for ordering relation O* if there is a key-parameterized function $G : (S \rightarrow K) \rightarrow S^* \rightarrow S^*$ together with a key order $((K, \leq_K), key : S \rightarrow K)$ such that $g = G(key)$ where $O(x, y) \Leftrightarrow key(x) \leq_K key(y)$ and g is a permutation function consistent with O . □

Proposition 6.8 *Let $f : S^* \rightarrow S^*$ be a permutation function that is consistent with a total order (S, O) . Then f is a key-based sorting function for O .*

PROOF Follows from the definition of key-based sorting function with $S = K$. □

Theorem 6.9 *Let $g : S^* \rightarrow S^*$ be a key-based sorting function for O . Then:*

- (1) g is a parametric permutation function.
- (2) $O = \leq_f$

PROOF Left as an exercise. □

Sorting algorithms define either comparison-based or key-based sorting functions. Observe that sorting algorithms on total orders are trivially key-based by Proposition 6.8. We have shown that, in either case, a sorting function for a given ordering relation is a parametric permutation function with coinciding canonically induced ordering relation.

6.3 Nonexamples

Having argued that sorting algorithms implement parametric permutation functions in the sense of Definition 5.2 it is time to exhibit permutation functions that fail to be parametric.

- (1) Consider $sortBy : (X \times X \rightarrow Bool) \rightarrow X^* \rightarrow X^*$ as defined in the Haskell base library. This is *not* a parametric permutation function for

the simple reason that it does not have the right type. Instead it is a comparison-parameterized function (Definition 6.2) that, when given an inequality test, generates a comparison-based sorting function (Definition 6.3), which, as we have seen, is parametric (Theorem 6.4). When given a function that is not an inequality test, the resulting function does not sort.

- (2) Consider a probabilistic or nondeterministic sorting algorithm, such as Quicksort with random selection of the pivot element. It does not implement a sorting function for the simple reason that it is not a *function*: the same input may be mapped to different outputs during different runs.
- (3) Consider the permutation function of Theorem 3.5. It is a permutation function, but not locally consistent and consequently neither parametric, since it fails to locally decide its canonically induced ordering relation. Even though it orders the input such that the output respects some ordering relation we cannot use it to decide the ordering relation.
- (4) Consider the function $f : \mathbb{N}_0^* \rightarrow \mathbb{N}_0^*$ that first lists the even elements in its input and then the odd ones, in either case in the same order as in the input. f is parametric, even stable. Now modify f as follows:

$$\begin{aligned} f'(6815) &= 8615 \\ f'(\vec{x}) &= f(\vec{x}), \text{ otherwise} \end{aligned}$$

f' is locally consistent, but not parametric. To wit, clearly $x_1 \leq_{f'} x_2$ if and only if x_1 is even or x_2 is odd since any even number occurs to the left of any other number in some output, and any odd number occurs to the right of any other number in some output. If x_1 is even or x_2 is odd we have $f'(x_1x_2) = x_1x_2$ by definition of f' and consequently $O_{f'}(x_1, x_2)$ holds, too. This shows that $x_1 \leq_{f'} x_2$ implies $O_{f'}(x_1, x_2)$ and can conclude that f' is locally consistent.

To see that f' is *not* parametric consider $R = \{(6, 8), (8, 6), (15, 15)\}$. Clearly $R \subseteq \equiv_{f'}$ since both 6 and 8 are even. Now, consider the sequences 6815 and 8615. In particular, we have $R^*(6815, 8615)$. By definition of f' we furthermore have $f'(6815) = 8615$ and $f'(8615) = f(8615) = 8615$. But $R^*(8615, 8615)$ does *not* hold, and so f' is not parametric.

Observe that the ordering canonically induced by f' is determined by the least significant bits of (the binary representations of) the input elements, with the other bits being satellite data. The algorithm for f' , however, needs to inspect the satellite data and perform different computations depending on what it finds there. Since f' does not just copy or move satellite data, but actually inspects them, it violates obliviousness.

7 Stable permutation functions

We have seen that each permutation function $f : S^* \rightarrow S^*$ canonically induces a unique most discriminative ordering relation \leq_f (Proposition 3.4). Permutation functions with the same canonically induced ordering relation may differ in which order they return \equiv_f -equivalent inputs in their output, however. A natural choice for a *canonical* permutation function amongst all permutation functions with the same canonically induced order is the one that permutes its input *stably*.

Definition 7.1 [Stability] Let (S, O) be a total preorder. We say $\vec{y} = y_1 \dots y_n \in S^*$ is a *stable* O -ordered permutation of \vec{x} if

- \vec{y} is a permutation of \vec{x} ;
- \vec{y} is O -ordered: $O(y_i, y_{i+1})$ for all $1 \leq i <_\omega n$;
- \vec{y} is *stable* with respect to \vec{x} : $\vec{y}|_{[z]_{\equiv_O}} = \vec{x}|_{[z]_{\equiv_O}}$ for all $z \in S$.

□

The last condition expresses that the relative order of equivalent elements in \vec{x} is preserved in \vec{y} .

Definition 7.2 [Stable permutation function]

- (1) A permutation function $f : S^* \rightarrow S^*$ is *stable with respect to ordering relation* O if $f(\vec{x})$ is the stable O -ordered permutation of \vec{x} for all $\vec{x} \in S^*$.
- (2) f is (*intrinsically*) *stable* if it is stable with respect to \leq_f .

□

Stable permutation functions are uniquely determined by their ordering relation:

Proposition 7.3 *Let (S, O) be a total preorder. Then:*

- (1) *Each $\vec{x} \in S^*$ has a unique stable O -ordered permutation.*
- (2) *There exists exactly one permutation function f such that f is stable with respect to O .*

PROOF

- (1) Easy.
- (2) Follows immediately from the first statement.

□

Conversely, a permutation function can be stable with respect to at most one ordering relation, and that is the function's canonically induced ordering relation.

Theorem 7.4 *Let $f : S^* \rightarrow S^*$ be a permutation function that is stable with respect to O . Then:*

- (1) f is intrinsically stable.
- (2) $O = \leq_f$.

PROOF

- (1) Follows from the second statement.
- (2) Since f is, by Definition 7.2, consistent with O , we have $\leq_f \subseteq O$. To show $O \subseteq \leq_f$, consider x, y such that $O(x, y)$. Applying f to $[x, y]$ yields $[x, y]$ and thus we immediately obtain $x \leq_f y$.

□

As a consequence we get that ordering relations and stable permutation functions are in one-to-one correspondence:

Theorem 7.5 (Order/stable permutation isomorphism) *For each set S , the set of stable permutation functions on S is in one-to-one correspondence with the set of ordering relations on S .*

PROOF Let S be given. Consider map H , which maps a stable permutation function f to \leq_f , and I , which maps an ordering relation O to the unique stable permutation for O (Proposition 7.3).

Now, consider $I(H(f))$ for arbitrary stable permutation function f . By definition it is stable with respect to \leq_f . Since $H(f) = \leq_f$, $I(H(f))$ is also stable with respect to \leq_f . By Proposition 7.3 we must have $I(H(f)) = f$. Since there is a stable permutation function f with respect to every ordering relation O (Proposition 7.3) and $\leq_f = O$ (Theorem 7.4), we can furthermore conclude that H is surjective. Thus the pair (H, I) is a bijection between the stable permutation functions $f : S^* \rightarrow S^*$ and ordering relations O on S . □

We could now formulate the second answer to our central question: A function f is a stable sorting function if it is a parametric permutation function and if

Requirement 4: f is stable.

As it turns out, parametricity is not required since stability already subsumes it.

Lemma 7.6 (Stability implies parametricity) *Let $f : S^* \rightarrow S^*$ be a stable permutation function. Then:*

- (1) $x \leq_f y \Leftrightarrow f(xy) = xy$
- (2) f is intrinsically parametric.

PROOF

- (1) (a) \Leftarrow : Immediate from Proposition 3.4.
- (b) \Rightarrow : Assume $x \leq_f y$. If $x <_f y$ we immediately have $f(xy) = xy$. If $x \equiv_f y$ we obtain $f(xy) = xy$ from stability of f .
- (2) We have to show that stability implies parametricity. Let $R \subseteq \equiv_f$; that is, by Corollary 5.4, $R(x, y) \Rightarrow f(xy) = xy \wedge f(yx) = yx$. We need to show that for $R^*(\vec{x}, \vec{y})$ we have $R^*(f(\vec{x}), f(\vec{y}))$. Assume that $R^*(\vec{x}, \vec{y})$. The outputs $f(\vec{x})$ and $f(\vec{y})$ can then be constructed as follows:
 - (a) If \vec{x} is empty $\vec{y}, f(\vec{x})$ and $f(\vec{y})$ are also empty; in particular, we have $R^*(f(\vec{x}), f(\vec{y}))$.
 - (b) If \vec{x} is of length $n > 0$, choose index i_0 such that $x_{i_0} \leq_f x_i$ for all $i = 1 \dots n$. Output all the x_i such that $x_i \equiv_f x_{i_0}$ in index-order and remove them from \vec{x} . Note that i_0 can be chosen the same for \vec{y} since $y_{i_0} \equiv_f x_{i_0} \leq_f x_i \equiv_f y_i$. Also, whenever x_i is output we also output y_i and thus R holds in the corresponding output position.

Then continue with Step 2a.

Since the above algorithm terminates this shows that $R^*(f(\vec{x}), f(\vec{y}))$, as desired.

□

We are now in a position to offer the second answer to the title of this paper:

Being a “stable sorting function” means being an intrinsically stable permutation function.

8 Monitoring stability

We have identified parametric and stable permutation functions as the notions that capture the extensional properties of sorting algorithms. Stable permutation functions have the advantage that they are in one-to-one correspondence with ordering relations. But how do we “know” whether or not a function is permutative and stable based only on observations of its I/O behavior?

Clearly, permutativity and stability of a function f are global properties, ones

that cannot be determined affirmatively after observing a finite number of input/output-pairs of the graph of f . What we are interested in then is *monitoring* the calls to f and checking whether or not the observed I/O-pairs are consistent with at least one stable permutation function.

There are two variations of monitoring we consider: In *active monitoring* we have no storage space for remembering any information about previously observed I/O-pairs, but we may invoke f during monitoring. In *passive monitoring* we do have storage, but we are not allowed to invoke f for monitoring purposes.

Why would we be interested in monitoring? For one, a purportedly stable permutation function f may be used in a software component (context) $C[]$. Now assume an error is observed in the execution of $C[f]$. As an aid to defect localization monitoring the calls of f is a way of checking at run time whether or not f obeys its *interface contract* of being permutative and stable. As we shall see the run-time complexity of monitoring is prohibitive in comparison to the execution of f , making it impractical in normal mode execution, but in debugging mode it can be expected to be useful for localizing errors.

8.1 Local characterization of stability

Here we show that stability has an “operational” characterization that captures a simple intuition: A permutation function is stable if and only if it preserves the relative order of two elements in the input whenever it does so for the two-element sequence consisting of those elements by themselves.

Theorem 8.1 (Characterization of stable permutation function) *Let $f : S^* \rightarrow S^*$. The following statements are equivalent:*

- (1) f is a stable permutation function.
- (2) f is consistently permutative: For each sequence $x_1 \dots x_n \in S^*$ there exists $\pi \in \mathcal{S}_{|\bar{x}|}$ 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).

The permutation π in Theorem 8.1 can be thought of as mapping the *rank* of an (occurrence of an) element—where it occurs in the output of f —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 f must always be the same.

PROOF

- $1 \implies 2$: Let f be a stable permutation function. Consider $x_1x_2 \dots x_n$. The algorithm in the proof of Theorem 7.6 can be instrumented to produce the permutation π^{-1} as follows: Whenever x_i is output into slot j of the output, define $\pi^{-1}(i) = j$. Now consider $1 \leq_\omega i \leq_\omega j \leq_\omega n$:
 - If $x_i \leq_f x_j$ then $f(x_ix_j) = x_ix_j$ by Theorem 7.6 and $\pi^{-1}(i) \leq_\omega \pi^{-1}(j)$ by construction of π^{-1} .
 - If $x_i >_f x_j$ then $f(x_ix_j) = x_jx_i$ and $\pi^{-1}(i) >_\omega \pi^{-1}(j)$.
Thus $f(x_ix_j) = x_ix_j$ if and only if $\pi^{-1}(i) \leq_\omega \pi^{-1}(j)$, which shows that f is consistently permutative.
- $1 \longleftarrow 2$: Assume f is consistently permutative.

First we prove that $x \leq_f y \Leftrightarrow f(xy) = xy$. The \leftarrow -direction is obvious. For the \Rightarrow -direction, contrapositively assume $x \leq_f y$ and $f(xy) = yx$, with $x \neq y$. By consistent permutativity we can conclude that for all $\vec{x} = x_1 \dots x_n$ where $x = x_i$ and $y = x_j$ and all permutations $\pi \in S_n$ we have $\pi^{-1}(i) > \pi^{-1}(j)$. This means that the rank of x_i is always greater than the rank of x_j ; in other words, x occurs only to the right of y in all outputs of f . By Proposition 3.4 this means that $x >_f y$, which contradicts our assumption $x \leq_f y$.

Now, consider $\vec{x} = x_1 \dots x_n$ and let π be a permutation such that $\pi^{-1}(i) \leq_\omega \pi^{-1}(j) \Leftrightarrow f(x_ix_j) = x_ix_j \Leftrightarrow x_i \leq_f x_j$, with the last equivalence following from the above. Consider $i < j$ such that $x_i \leq_f x_j$. From $\pi^{-1}(i) \leq_\omega \pi^{-1}(j)$ we obtain immediately that x_i occurs to the left of x_j in $f(\vec{x})$. This holds in particular if $x_i \equiv_f x_j$. So π maps \equiv_f -equivalent elements in \vec{x} to the output such that their relative order is preserved, and we can conclude that f is stable.

□

8.2 Noncharacterizations

The Characterization Theorem for stable permutation functions (Theorem 8.1) requires the use of permutations. A natural question is whether there are simpler or otherwise plausible looking characterizations. This section is about a number of such attempts that do *not* provide characterizations despite looking appealing at first sight.

We start by stating some properties of stable permutation functions with the intent of eventually using them in our alternative characterization attempts.

Proposition 8.2 (Stability implies idempotency) *Every stable permutation function f is idempotent: $f \circ f = f$.*

PROOF Consider \vec{x} . By definition of \leq_f we have that $f(\vec{x})$ is \leq_f -ordered.

Since f is stable with respect to \leq_f it acts as the identity on $f(\vec{x})$, and we have $f(f(\vec{x})) = f(\vec{x})$. \square

Definition 8.3 [Filter consistency] We call a function $f : S^* \rightarrow S^*$ *filter consistent* if $(f(\vec{x}))|_P = f(\vec{x}|_P)$ for all $P \subseteq S$. \square

Proposition 8.4 (Stability implies filter consistency) *Not every parametric permutation function is filter consistent, but every stable permutation function is so.*

PROOF (Hint) For the first part use the parametric permutation function in the proof of Proposition 8.7 below. \square

Having seen that stable permutation functions are idempotent and filter consistent, let us see whether the converse also holds. As it turns out, it does not hold.

Proposition 8.5 *There exists a permutation function $f : S^* \rightarrow S^*$ that is idempotent and filter consistent, but is not locally consistent, and consequently neither parametric nor stable.*

PROOF Consider function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ defined by sorting the input string in 0-1-order: 0s first, then 1s. This is a stable permutation function with $x \leq_f y \Leftrightarrow x \leq_\omega y$.

Now define $f'[0, 1, 0] = [0, 1, 0]$ and $f'(\vec{x}) = f(\vec{x})$ otherwise. f' is idempotent: Applying f' twice to $[0, 1, 0]$ yields $[0, 1, 0]$, the same as applying it once. For other values idempotency follows from the idempotency of f . f' is also filter-consistent: For $P = \{0\}$ and $P = \{1\}$ we we have $(f'(\vec{x}))|_P = (f(\vec{x}))|_P = f(\vec{x}|_P) = f'(\vec{x}|_P)$ by definition of f' , Proposition 8.4 and stability of f . For $P = \emptyset$ and $P = S$, the only two other choices for P , filter consistency holds trivially.

But f' is not locally consistent: $x \leq_{f'} y$ clearly holds for all $x, y \in \{0, 1\}$ due to 0 occurring to the right of 1 in the output of $f'[0, 1, 0] = [0, 1, 0]$, but $O_f(x, y) \Leftrightarrow x \leq_\omega y$, which shows $O_f \neq \leq_{f'}$. \square

The need in Theorem 8.1 for referring explicitly to permutations may lead us to the idea of trying to characterize stability without the permutation tying inputs and outputs together.

Definition 8.6 [Pairwise stable function] Let $f : S^* \rightarrow S^*$ be a permutation function. We say f is *pairwise stable* if for all $y_1 y_2 \dots y_n$ in the range of f we have $\forall i, j. i \leq_\omega j \Rightarrow f(y_i y_j) = y_i y_j$. \square

Even combined with parametricity pairwise stability does not yield stability, however.

Proposition 8.7 *There exists a pairwise stable parametric permutation function that is not stable.*

PROOF Consider function $f : \mathbb{N}_0^* \rightarrow \mathbb{N}_0^*$ that first lists the even elements in its input, then the odd elements. For even-length inputs it acts stably on its input; for odd-length inputs it acts reverse-stably: equivalent inputs are listed in reverse order relative to their input order of occurrence.

It can be shown that f is a parametric permutation function with $x \leq_f y$ if and only if x is even or y is odd: f is evidently a permutation function, and as for parametricity it can be seen that any relation that relates even elements to evens and odds to odds is preserved by f .

f is not stable, however: To wit, $f[4, 6, 8] = [8, 6, 4]$. Since all even numbers are \equiv_f -equivalent, f would have to produce $[4, 6, 8]$ if it were stable. \square

Having seen that pairwise stable permutation functions are not necessarily stable, we may be tempted to change the definition of consistent permutativity just slightly: It is notable that consistency is only required for one of the permutations that transform the input into the functions output. What about requiring that that property hold universally, for all such permutations?

Definition 8.8 [Strongly consistent permutation function] Let $f : S^* \rightarrow S^*$. We say f is a *strongly consistent permutation function* if

- (1) f is a permutation function.
- (2) For all permutations π such that $f(x_1 \dots x_n) = x_{\pi(1)} \dots x_{\pi(n)}$ we have $\forall i, j \in [1 \dots n] : f(x_i x_j) = x_i x_j \Leftrightarrow \pi^{-1}(i) \leq_\omega \pi^{-1}(j)$.

\square

As it turns out, the notion of strongly consistent permutativity is nonsensical:

Proposition 8.9 *There does not exist a strongly consistent permutation function.*

Note that we generally assume that $S \neq \{\}$.

PROOF (Hint) Let f be any permutation function and apply it to a sequence of at least two equal elements. \square

Let \vec{y} be the result of $f(\vec{x})$.

- (1) Build permutation π such that $\pi(\vec{x}) = \vec{y}$ and $i < j \Leftrightarrow \pi(i) < \pi(j)$ for all i, j where $x_i = x_j$. If there is no such permutation, signal error “not a permutation function”.
- (2) Then perform the check $f(x_i x_j) = x_i x_j \Leftrightarrow \pi^{-1}(i) \leq_{\omega} \pi^{-1}(j)$ for all $i, j \in [1 \dots n]$ from Theorem 8.1. If it fails for a pair i, j , signal error “not stable”.

Fig. 1. Active monitoring of permutativity and stability of a function f

8.3 Active monitoring

Theorem 8.1 can be used as the basis of active run-time monitoring of the calls to a function to validate its permutativity and stability. See Figure 1.

The quadratic number of calls in the last step of the algorithm in Figure 1 seems unavoidable. This appears to make active run-time monitoring of a function practically applicable only in debugging mode since the monitoring dominates the cost of executing the function.

8.4 Passive monitoring

The passive monitoring problem is the following: Given a set of I/O-pairs, determine whether or not it can be extended to the graph of a stable permutation function. Furthermore, make the determination incrementally, after each addition of an I/O pair.

Definition 8.10 [Inequality constraints] Let C be a set of formal constraints of the form $x \leq y$ or $x < y$ where $x, y \in S$ for some set S . We say total preorder (S, O) satisfies C and write $(S, O) \models C$ if $O(x, y)$ for all $x \leq y \in C$, and $O(x, y) \wedge \neg O(y, x)$ for all $x < y$. We say C is *satisfiable* if there exists an ordering relation O such that $(S, O) \models C$. \square

Our approach is as follows: We maintain a set of inequality constraints. Upon arrival of a new I/O pair we extract a set of constraints, add them to the existing constraints and then determine whether the extended constraint set is still satisfiable. See Figure 2. It is easy to see that satisfiability can be maintained using standard techniques, e.g. by reduction to the all-pair shortest (actually: longest) paths problem, in time $O(|C|)$ for each addition of constraint $x \leq y$ or $x < y$ to C . Just as active monitoring, passive run-time monitoring is not practical in normal mode since its time and space complexity dwarfs the execution cost of f even if f is implemented by an asymptotically inefficient sorting algorithm.

Let C be a variable containing a set of inequality constraints, initially empty. For each input/output pair $\vec{y} = f(\vec{x})$ do the following:

- (1) Check if \vec{y} is a permutation of \vec{x} . If not, signal error “not a permutation function”.
- (2) Consider distinct elements x, y occurring in a sequence. Then either all occurrences of one element occur before all the occurrences of the other element or the occurrences of the elements are intermixed: there is an occurrence of one element that has an occurrence of the other element both to the left and to the right. all occurrences of y occur before the occurrences of x We say all occurrences of an element

For each pair of elements x, y occurring in \vec{x} and thus also in \vec{y} , add constraints to C according to the following table:

input (\vec{x})	output (\vec{y})	constraints added
all x before all y	all x before all y	$x \leq y$
all x before all y	all y before all x	$y < x$
all x before all y	x and y intermixed	\perp
x and y intermixed	all x before all y	$x < y$
x and y intermixed	all y before all x	$y < x$
x and y intermixed	x and y intermixed, same relative order	$x \equiv y$
x and y intermixed	x and y intermixed, different relative order	\perp

Here $x \equiv y$ denotes $x \leq y \wedge y \leq x$, and \perp denotes an unsatisfiable constraint: The I/O pair in question cannot occur in any stable permutation function.

- (3) Check whether C is satisfiable. If not, signal error “not stable”.

Fig. 2. Passive monitoring of permutativity and stability of function f

9 Structures representing orders

Imagine we are interested in providing clients access to an ordered datatype. How should the ordering relation be represented as an *operation*?

We shall call a structure consisting of a set S and a function of some type (full function space) over S a (*functional*) *concrete representation* of the order if the function corresponds to the ordering relation in some (precise) sense. Only a subspace of the functions will represent ordering relations, however. Since we want to make it possible for a programmer to *define* an ordering relation by giving a function from the full function space it is important to be able to

determine whether a given function corresponds to any ordering relation at all.

The program of this section is as follows.

- (Embedding) Embed orders (S, O) into a category of structures consisting of S and an operation (function) f of a certain type (full function space) over S .
- (Characterization) Find an intrinsic characterization that can be used for run-time monitoring to discover when an element of the function space fails to be in the subspace corresponding to ordering relations.

We do this for structures with inequality tests (type: $S \times S \rightarrow Bool$), comparators (type: $S \times S \rightarrow S \times S$) and finally stable permutation functions (type: $S^* \rightarrow S^*$).

9.1 Structures with inequality test

The canonical way of representing an ordering relation on S is by providing an inequality test (Definition 6.1).

Definition 9.1 [Structure with inequality test] A *structure with inequality test* (S, lte) is a set S together an inequality test $lte : S \times S \rightarrow Bool$. \square

Clearly, the ordering relation and the inequality test determine each other:

Proposition 9.2 (Order/inequality isomorphism) For each set S , the set of inequality tests on S is in one-to-one correspondence with the set of ordering relations on S .

PROOF By definition. \square

By Proposition 9.2 inequality tests *embed* ordering relations into the function space $S \times S \rightarrow Bool$. We can thus think of a structure with an inequality test as a *concrete representation* of an order. The concreteness lies in offering clients—programs using the structure—a *particular operation* for getting access to the ordering relation, but exposing no other information than the ordering relation itself.

When given a function $f : S \times S \rightarrow Bool$, it may or may not be an inequality test. The following proposition formulates an intrinsic characterization of inequality tests that can be used for monitoring the calls to f to ensure that it is an inequality test.

Proposition 9.3 *Let $f : S \times S \rightarrow \text{Bool}$. The following statements are equivalent:*

- (1) *f is an inequality test.*
- (2) *For all $x, y, z \in S$:*
 - (a) *$f(x, y) = \text{true} \wedge f(y, z) = \text{true} \implies f(x, z) = \text{true}$*
 - (b) *$f(x, y) = \text{true} \vee f(y, x) = \text{true}$.*

So if we ever encounter a situation where neither $f(x, y)$ nor $f(y, x)$ yield *true* or where both $f(x, y)$ and $f(y, z)$ do, but $f(x, z)$ does not, we have *finite* evidence that f is not an inequality test. This can be turned into a passive run-time monitoring algorithm for inequality tests: Maintain a set of inequality constraints C as in Definition 8.10. Upon observing I/O pair $((x, y), \text{true})$ add $x \leq y$ to C ; if the I/O pair is $((x, y), \text{false})$, add $y < x$ to C . Then check satisfiability.

Active monitoring, without auxiliary space to store information about previous calls to f , is not possible. Without auxiliary storage we can only check totality, but not transitivity.

9.2 Comparators

We have seen that inequality tests are a concrete representation of ordering relations. Another concrete representation is by *comparators*.

Definition 9.4 [Comparator] A *comparator structure* (S, comp) is a set S together with a permutative function $\text{comp} : S \times S \rightarrow S \times S$ such that (S, \leq_{comp}) defined by $x \leq_{\text{comp}} y \iff \text{comp}(x, y) = (x, y)$ is an order. We call comp a *comparator (function)* on S .⁵ \square

By definition, the ordering relation \leq_{comp} is uniquely determined by comp . Conversely, given an ordering relation \leq there is exactly one comparator comp_{\leq} for it.

Lemma 9.5 (Injectivity of \leq_{comp}) *Let (S, comp) and (S, comp') be comparator structures such that $\leq_{\text{comp}} = \leq_{\text{comp}'}$. Then $\text{comp} = \text{comp}'$.*

PROOF Straightforward. \square

⁵ Note that the term comparator is also used in the sense of 3-valued inequality test in Java [GJSB05]. We follow the established usage of the word from electronics and sorting networks [Knu98] here, though.

Definition 9.6 [$comp_R$] Let $R \subseteq S \times S$. Define

$$comp_R(x, y) = \begin{cases} (x, y), & \text{if } R(x, y) \\ (y, x), & \text{otherwise} \end{cases}$$

□

Lemma 9.7 Let (S, R) be an order. Then $comp_R$ is a comparator. Furthermore, $\leq_{comp_R} = R$.

PROOF Straightforward. □

Theorem 9.8 (Order/comparator isomorphism) For each set S , the set of comparators on S is in one-to-one correspondence with the set of ordering relations on S .

PROOF Follows from injectivity (Lemma 9.5) and surjectivity (Lemma 9.7). □

Having proved that ordering relations are represented by a subspace of the functions of type $S \times S \rightarrow S \times S$ we are interested in finding an intrinsic characterization of comparators that, in principle, can be used to discover, using finite evidence only, whenever such a function fails to be a comparator.

Theorem 9.9 (Characterization of comparators) Let $f : S \times S \rightarrow S \times S$. The following statements are equivalent:

- (1) f is a comparator.
- (2) For all $x, y, z \in S$:
 - (a) (permutativity) $comp(x, y) = (x, y) \vee comp(x, y) = (y, x)$;
 - (b) (transitivity) $comp(x, y) = (x, y) \wedge comp(y, z) = (y, z) \implies comp(x, z) = (x, z)$;
 - (c) (idempotency) $comp(x, y) = (y, x) \implies comp(y, x) = (y, x)$.

PROOF Straightforward. □

Using Theorem 9.9 it is possible to develop a passive run-time monitoring algorithm for comparators (exercise).

Definition 9.10 [Weak comparator] Call $f : S \times S \rightarrow S \times S$ a *weak comparator* if it is permutative and *weakly transitive*: For all $k, l \in \mathbb{N}_0$, if $f^k(x, y) = (x, y)$ and $f^l(y, z) = (y, z)$ then there exists $m \in \mathbb{N}_0$ such that $f^m(x, z) = (x, z)$. □

Proposition 9.11 *If f is a weak comparator, f^2 is a comparator.*

PROOF Permutativity is assumed. Observe: Because of permutativity, if $f^k(x, y) = (x, y)$ then $f^2(x, y) = (x, y)$. Weak transitivity then gives that f^2 is transitive. Finally, idempotency of f^2 holds for any permutative f : $f^2(x, y) = (y, x)$ can only hold if $f(x, y) = (y, x)$ and $f(y, x) = (y, x)$. But then we have $f^2(y, x) = f(f(y, x)) = f(y, x) = (y, x)$. \square

9.3 Stable permutation functions

Definition 9.12 [Sorting structure] A *sorting structure* (S, sort) is a set S together with a stable permutation function sort . \square

Note that the function in a sorting structure must be a *stable* permutation function.

The point of the previous sections was to demonstrate that stable permutation functions are also concrete representations of ordering relations:

- They embed ordering relations on S into the space of functions of type $S^* \rightarrow S^*$ (Theorem 7.5).
- Stable permutation functions have an intrinsic characterization for verifying when a function $f : S^* \rightarrow S^*$ is not a stable permutation function (Theorem 8.1).

9.4 Isomorphisms

We have shown that orders, structures with inequality tests, comparator structures and sorting structures with the same underlying set S are in one-to-one correspondences. What is more, the isomorphisms between inequality tests, stable permutation functions and comparators can be defined parametrically polymorphically.

Theorem 9.13 *The isomorphisms between inequality tests, stable permutation functions and comparators can be defined parametrically polymorphically:*

- (1) $\text{sort}^{\text{lte}} : \forall X \subseteq S \times S. (X \times X \rightarrow \text{Bool}) \rightarrow (X^* \rightarrow X^*);$
- (2) $\text{lte}^{\text{sort}} : \forall X \subseteq \text{Id}_S. (X^* \rightarrow X^*) \rightarrow (X \times X \rightarrow \text{Bool});$
- (3) $\text{sort}^{\text{comp}} : \forall X \subseteq S \times S. (X \times X \rightarrow X \times X) \rightarrow (X^* \rightarrow X^*);$
- (4) $\text{comp}^{\text{sort}} : \forall X \subseteq S \times S. (X^* \rightarrow X^*) \rightarrow (X \times X \rightarrow X \times X);$
- (5) $\text{comp}^{\text{lte}} : \forall X \subseteq S \times S. (X \times X \rightarrow \text{Bool}) \rightarrow (X \times X \rightarrow X \times X);$
- (6) $\text{lte}^{\text{comp}} : \forall X \subseteq \text{Id}_S. (X \times X \rightarrow X \times X) \rightarrow (X \times X \rightarrow \text{Bool}).$

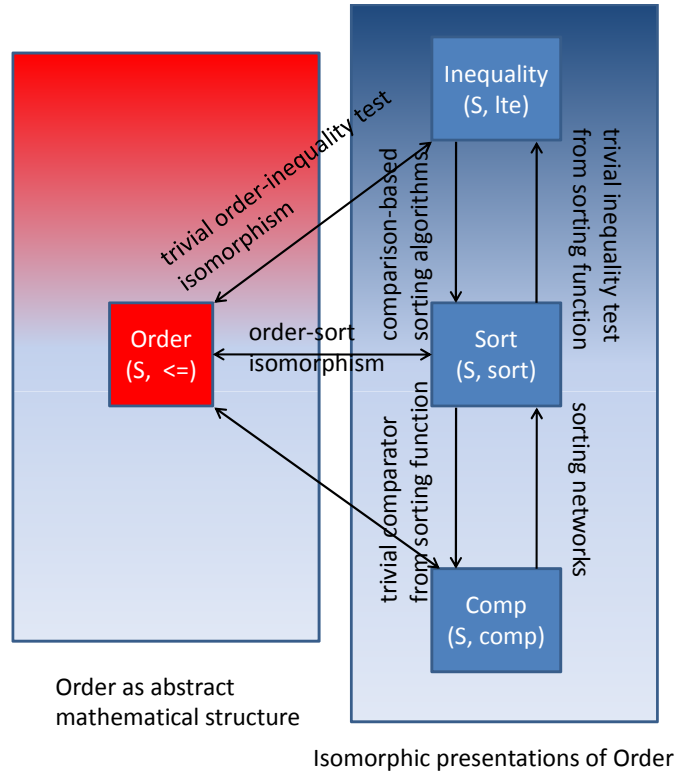


Fig. 3. Isomorphisms

PROOF The proof is by construction. See Appendix A for explicit definitions of the mappings in Haskell. The definition of stable permutation functions from inequality tests implements Mergesort [Knu98, Sec. 5.2.4]; the implementation of stable permutation functions from comparators implements Odd-even mergesort [Bat68]. Any other comparator-based sorting algorithm (sorting network) would do as well here. Note that lte^{sort} and lte^{comp} are only parametric over relations that respect equality [Wad89, Section 3.4] since we need to use an effective equality test on the underlying set to define them. The use of Odd-even mergesort, a comparator-based sorting algorithm, proves that this is not necessary for the definition of stable permutation functions from comparators. \square

The isomorphisms between orders and their representations as structures with inequality tests, stable permutation functions and comparators, respectively, and the parametric polymorphic definitions of Theorem 9.13 are pictured in Figure 3.

Parametric polymorphic definability guarantees that each operation is maximally abstract in the sense that the mappings making up the isomorphisms

only require and expose information about the ordering relation of the underlying set. This guarantees representation independence [Mit86]: The implementation of an ordered data type T whose ordering is exposed as a stable permutation function can be freely changed without breaking *any* code that uses the stable permutation function so long as its ordering relation is the same.

Contrast this to providing a *ranking* function, say $rank : T \rightarrow \mathbb{Z}$ with the standard ordering on \mathbb{Z} , to clients of T as a way of exposing the ordering relation on T . Now fewer implementation changes are possible since there may be client code that uses the ranking function for other purposes than comparing or sorting.

10 Conclusion

10.1 Summary

We have posed a simple question: What is a sorting function? What makes the question interesting is that it does not ask for what it means to be a sorting function *for a given order*. Indeed the point of the question is to arrive at ways of *specifying* orders by sorting functions instead of having to be given one in the first place.

First we formulate four basic properties of sorting algorithms as requirements: they permute, put in order, are oblivious to satellite data, and may be stable.

We define increasingly restrictive subclasses of permutation functions: locally persistent permutation functions, intrinsically parametric permutation functions, and intrinsically stable permutation functions. We show that a sorting function for an ordering relation O , whether comparison-based or key-based (distributive), is an intrinsically parametric permutation function whose canonically induced ordering relation coincides with O .

A stable sorting function with respect to O is shown to be an intrinsically stable permutation function whose canonically induced ordering relation coincides with O .

Thus we arrive at the following answers:

- A function is a “*sorting function*” if it is an intrinsically parametric permutation function.
- A function is a “*stable sorting function*” if it is an intrinsically stable permutation function.

We exhibit a characterization of stable permutation functions that provides a basis for run-time monitoring of the I/O behavior of a function to check that it really behaves like a stable permutation function.

Finally we show that inequality tests, comparators and intrinsically stable permutation functions are isomorphic ways of *specifying* orders, which furthermore only reveal information about the ordering relation of a type, nothing else.

10.2 Extensions

The notion of sorting *function* does not cover a probabilistic unstable sorting algorithm such as Quicksort with random pivot where multiple runs of the algorithm may produce different outputs for the same input. We believe that the notions and results here can be straightforwardly generalized to sorting *relations*, which may associate multiple results with the same input.

We have described active and passive run-time monitoring for checking that a function behaves like a stable permutation function. Doing the same for parametric sorting functions remains to be investigated. We expect Corollary 5.4 to play a central role in this.

10.3 Discussion

We have broken the definitional predominance of orders over sorting functions since it is not necessary to know what an order is for defining what a “sorting” or “stable sorting” function is. To illustrate this, we can flip their traditional roles on the head: We can define the very notion of “ordering relation” from the notion of stable permutation function.

We start by defining an *ordering relation for stable permutation function* $f : S^* \rightarrow S^*$: it is the binary relation $O \subseteq S \times S$ such that $O(x, y) \iff f(xy) = xy$. Now we can move on to defining what an ordering relation *eo ipso* is—not with respect to a given stable permutation function, but intrinsically. This corresponds to the task of defining what a sorting function is in this paper.⁶ It is an ordering relation for *some* stable permutation function. And finally we can move on to prove that ordering relations defined in this fashion have the following *intrinsic characterization*: A relation is an ordering relation if and only if it is transitive and total. And since this characterization does not mention the notion of stable permutation function we will have made the

⁶ Imagine writing an article entitled “What is an ordering relation?”

notion of ordering relations independent of the notion of stable permutation functions!

Inequality tests, comparators and stable permutation functions are conceptually and behaviorally interchangeable. Does it matter then whether an inequality test, comparator or stable sorting function is implemented “natively” and subsequently exported for an ordered datatype? Inequality tests and comparators provide information on the ordering relation for only two elements at a time. A well-known consequence is that any comparison-based sorting algorithm requires $\Omega(n \log n)$ applications of the inequality test to sort n elements [Knu98, Section 5.3.1]. In contrast, *distributive* sorting algorithms [Knu98, Section 5.2.5] such as Radixsort run in linear time. Radix sorting is known to be doable with no extra space using practical techniques [FMP07], which may make it competitive with space-efficient comparison-based sorting algorithms. (Space complexity is often a more serious concern in sorting than time complexity, which has favored comparison-based sorting algorithms.) Interestingly, such distributive algorithms can be defined and extended *generically* to arbitrary first-order types while preserving their linear-time performance [Hen08]. In other words: It is possible to implement a time-efficient sorting function for a new data type if we have access to time-efficient sorting functions or similar bulk processing functions for the (ordered) types used in its implementation. If instead only inequality tests or comparators are available we are back to the comparison-based sorting bottleneck.

The prospect of having only a purported sorting function as interface to an ordered type motivated the question addressed in this paper: What is it about a function from sequences to sequences that makes it be a (stable) sorting function, without access to an inequality test to compare it to?

10.4 Related work

Given the central importance of sorting in computer science a semantic analysis of the space of (observable behaviors of) sorting algorithms has been curiously unstudied so far. This is in contrast to the algorithms’ combinatorial, complexity-theoretic and engineering properties; see Knuth [Knu98] and Cormen, Leiserson, Rivest and Stein [CLRS01] for entries into the vast literature.

A key point is that we study sorting on total preorders instead of sorting on total orders, which may be canonically obtained from a total preorder by factoring out the equivalence relation. Informally, classical algorithmic and combinatorial analysis of sorting deals with sorting algorithms as operating on equivalence classes—making records indistinguishable by throwing their

satellite data away or ignoring them, which amounts to the same—whereas our analysis addresses sorting on the actual elements and what happens to them *inside* the equivalence classes of the total preorder. This is evidenced by the fact that our results do not say anything interesting for total orders: Every permutation function consistent with a total order is stable and thus parametric and locally consistent. Its parametricity property is trivial: it is just the simple type property.

We employ Parametricity Theory pioneered by Reynolds for System F [Rey83]. Wadler has demonstrated that interesting extensional properties can be derived for ML-polymorphic functions, in particular commutativity of a comparison-based sorting function with an order-mapping⁷ function applied to its arguments [Wad89]. Parametricity properties can be used to reduce the search space for verifying certain extensional properties. Day, Launchbury and Lewis [DLL99] showed that Knuth’s well-known 0-1 principle [Knu98, Section 5.3.4] is a corollary of the parametricity property embodied in the type of a sorting network, which is, in Haskell syntax, $((a, a) \rightarrow (a, a)) \rightarrow [a] \rightarrow [a]$. Voigtländer [Voi08] shows that the parametricity property of the (*parallel prefix*) *scan* operator implies a 0-1-2 principle: A function with the same parametric polymorphic type as *scan* implements *scan* correctly on all possible inputs if it does so on sequences of a three-element type.

We can observe that parametricity developed within semantics [Rey83] relates to obliviousness developed in complexity theory [PF79]. Henglein [Hen08] uses parametricity to facilitate an amortization argument in the asymptotic complexity analysis of generic discriminators, which are variants of sorting functions. We are not aware of other studies that systematically connect obliviousness and parametricity so as to bring techniques from both worlds to bear.

The monitoring of a function’s behavior to check that it behaves in compliance with its stipulated properties (permutativity, stability) is related to programming with *contracts*, which was pioneered by Meyer [Mey88] and extended to higher-order functions by Findler and Felleisen [FF02]. It is noteworthy that active monitoring, being stateless, fits into the operational framework of contract checking, but passive monitoring apparently does not, since it accumulates information about the previously observed I/O pairs. We are not aware of prior work on monitoring sorting functions.

More fundamentally, we are not aware of other studies that have asked what sorting “is” *without* taken orders as a given.

⁷ A function f between orders (S, \leq) and (S', \leq') is *order-mapping* if $x \leq y \Leftrightarrow f(x) \leq' f(y)$ for all $x, y \in S$. Note the bijection: This is a stronger property than *monotonicity* (*order preservation*).

Acknowledgements

This work has been partially supported by the Danish Natural Science Research Council (Forskningsråd for Natur og Univers) under Project *Applications and Principles of Programming Languages (APPL)*.

A brief presentation of the Characterization Theorem for stable sorting functions, but without parametricity to capture sorting functions, has been given at the 19th Nordic Workshop on Programming Theory (NWPT) in Oslo, Norway, October 10-12 [Hen07b], based on a preliminary technical report [Hen07a], where the Characterization Theorem (Theorem 8.1) served as a definition (!) of “sorting function”.

This paper contains substantially more and reworked material in comparison to the papers. In particular, central notions such as the canonically induced ordering relation, parametricity, local consistency, active and passive monitoring were discovered subsequent to NWPT 2007.

I would like to express my thanks to the editors of the special issue of JLAP on NWPT 2007 contributions for inviting this submission and to the referees of this paper for suggestions that have led to an undoubtedly clearer exposition, specifically by avoiding the word “sorting” in the development of what are now called locally consistent, intrinsically parametric and intrinsically stable permutation functions, respectively. Thanks to Bob Harper, Anders Schack-Nielsen and Phil Wadler for helpful comments and valuable corrections.

A Haskell definitions of isomorphisms

Haskell definitions of isomorphisms between inequality tests, sorting functions and comparators.

```
-- Signature of structures with inequality test
class Ineq a where
  lte :: (a, a) -> Bool
-- Signature of sorting structures
class Sort a where
  sort :: [a] -> [a]
-- Signature of comparator structures
class Comp a where
  comp :: (a, a) -> (a, a)

-- Sorting function defined from inequality test: Simple mergesort;
-- Alternatively any comparison-based sorting algorithm will do
sortByLte :: (Ineq a) => [a] -> [a]
sortByLte [] = []
sortByLte [x] = [x]
sortByLte xs = merge leftSorted rightSorted
  where (leftHalf, rightHalf) = splitAt (length xs `div` 2) xs
```

```

    leftSorted = sortByLte leftHalf
    rightSorted = sortByLte rightHalf
    merge [] ys = ys
    merge xs [] = xs
    merge (x : xs) (y : ys) | lte (x, y) = x : merge xs (y : ys)
                             | otherwise = y : merge (x : xs) ys
-- Inequality test from sorting function
lteBySort :: (Sort a, Eq a) => a -> a -> Bool
lteBySort x y = u == x
  where u : _ = sort [x, y]

-- Comparator from inequality test
compByLte :: (Ineq a) => (a, a) -> (a, a)
compByLte (x, y) | lte (x, y) = (x, y)
                 | otherwise = (y, x)
-- Inequality test from comparator
lteByComp :: (Comp a, Eq a) => (a, a) -> Bool
lteByComp (x, y) = u == x
  where (u, _) = comp (x, y)

-- Comparator from sorting function
compBySort :: (Sort a) => (a, a) -> (a, a)
compBySort (x, y) = (u, v)
  where [u, v] = sort [x, y]
-- Sorting function from comparator: Odd-even mergesort
-- Alternatively any other sorting network will do.
-- Note that using sorting networks proves that an equality test is
-- not necessary (compare to lteByComp and lteBySort)
sortByComp :: (Comp a) => [a] -> [a]
sortByComp [] = []
sortByComp [x] = [x]
sortByComp xs = oddEvenMerge leftSorted rightSorted
  where (leftHalf, rightHalf) = splitAt (length xs `div` 2) xs
        (leftSorted, rightSorted) = (sortByComp leftHalf, sortByComp rightHalf)
        oddEvenMerge [] ys = ys
        oddEvenMerge xs [] = xs
        oddEvenMerge left (fstR : right) = cmerge oeMergedEvens oeMergedOdds
          where (leftEvens, leftOdds) = pairs left
                (rightEvens, rightOdds) = pairs right
                oeMergedEvens = oddEvenMerge leftEvens rightEvens
                oeMergedOdds = oddEvenMerge leftOdds (fstR : rightOdds)
        pairs (x : y : xs) = (x : evens, y : odds)
          where (evens, odds) = pairs xs
        pairs [x] = ([x], [])
        pairs [] = ([], [])
        cmerge (x : xs) (y : ys) = u : v : cmerge xs ys
          where (u, v) = comp (x, y)
        cmerge [] ys = ys
        cmerge xs [] = xs

```

References

- [Bat68] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, 2d edition edition, 2001. ISBN 0-262-03293-7 (MIT Press) and ISBN 0-07-013151-1 (McGraw-Hill).

- [DLL99] Nancy A. Day, John Launchbury, and Jeff Lewis. Logical abstractions in haskell. In *Proc. Haskell Workshop (Utrecht, The Netherlands)*, number UU-CS-1999-28 in Technical Report. Utrecht University, 1999.
- [FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Notices*, 37(9):48–59, 2002.
- [FMP07] Gianni Franceschini, S. Muthukrishnan, and Mihai Pătraşcu. Radix sorting with no extra space. In *Proc. 15th European Symposium on Algorithms (ESA), Eilat, Israel*, volume 4698 of *Lecture Notes in Computer Science (LNCS)*, pages 194–205. Springer, October 2007.
- [GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 3d edition, 2005. ISBN 0-321-24678-0.
- [Hen07a] Fritz Henglein. Intrinsically defined sorting functions. TOPPS Report D-565, Department of Computer Science, University of Copenhagen (DIKU), September 2007. <ftp://ftp.diku.dk/diku/semantics/papers/D-565.pdf>.
- [Hen07b] Fritz Henglein. What is a sort function? In *Proc. 19th Nordic Workshop on Programming Theory (NWPT), Oslo, Norway*, October 10-12 2007.
- [Hen08] Fritz Henglein. Generic discrimination: Sorting and partitioning unshared data in linear time. In *Proc. 13th International Conference on Functional Programming (ICFP), Victoria, Canada*. ACM Press, September 2008.
- [Knu98] Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.
- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1st edition edition, 1988.
- [Mit86] J. Mitchell. Representation independence and data abstraction. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 263–276, St. Petersburg Beach, Florida, January 1986. ACM.
- [PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.
- [Rey83] J. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, pages 513–523, 1983.
- [Voi08] Janis Voigtländer. Much ado about two—a pearl on parallel prefix computation. In *Proc. 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, San Francisco, California, January 2008. ACM Press.
- [Wad89] P. Wadler. Theorems for free! In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 347–359. ACM Press, September 1989.