A bargain for mergesorts - How to prove your mergesort correct and stable, almost for free

Cyril Cohen, Kazuhiko Sakaguchi

▶ To cite this version:

Cyril Cohen, Kazuhiko Sakaguchi. A bargain for mergesorts - How to prove your mergesort correct and stable, almost for free. 2025. ⟨hal-05111866v1⟩

HAL Id: hal-05111866
https://hal.science/hal-05111866v1

Preprint submitted on 13 Jun 2025 (v1), last revised 14 Aug 2025 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

A bargain for mergesorts

How to prove your mergesort correct and stable, almost for free

CYRIL COHEN, Inria, CNRS, ENS de Lyon, UCBL, LIP, UMR 5668, France
KAZUHIKO SAKAGUCHI, CNRS, ENS de Lyon, UCBL, LIP, UMR 5668, France

We present a novel characterization of stable mergesort functions using relational parametricity, and show that it implies the functional correctness of mergesort. As a result, one can prove the correctness of several variations of mergesort (*e.g.*, top-down, bottom-up, tail-recursive, non-tail-recursive, smooth, and non-smooth mergesorts) by proving the characteristic property for each variation. Thanks to our characterization and the parametricity translation, we deduced the correctness results, including stability, of various implementations of mergesort for lists, including highly optimized ones, in the Rocq Prover (formerly the Coq Proof Assistant).

CCS Concepts: • **Theory of computation → Type theory**; **Program verification**; **Sorting and searching**.

Additional Key Words and Phrases: Interactive theorem proving, Parametricity, Mergesort, Stable sort

1 Introduction

Sorting is ubiquitous in computing. Over the decades, several algorithms, optimization techniques, and heuristics to solve it efficiently have been developed. Among these, mergesort achieves both optimal $O(n \log n)$ worst-case time complexity and stability—the property that a sort algorithm always preserves the order of equivalent elements—and is thus often preferred, particularly for sorting a linked list. On the other hand, defining a sort function and proving its correctness is often a good example to demonstrate some techniques (*e.g.*, how to use advanced recursion and induction) in interactive theorem provers such as Rocq [Leroy [n. d.]][Chlipala 2013, Section 7.1] and Isabelle [Nipkow et al. 2025; Sternagel 2013]. However, these case studies consider only one or a few simple implementations of mergesort and do not scale well to intricate implementations, such as smooth (Section 4.2) tail-recursive (Section 4.1) mergesort, since a methodology to share a large part of the functional correctness proofs between several variants of a sort algorithm has not been studied. In this paper, we address this issue by introducing a characterization of mergesort functions, which is easy to prove and implies several correctness results of mergesort.

The intuition behind our characterization of mergesort is as follows: by replacing all the occurrences of the merge function with concatenation, any stable mergesort function can be turned into the identity function. If it is not the identity function but may permutate some elements, the mergesort function is unstable (Section 3.2). In order to make sure that this replacement is done in the intended way, we first abstract out the mergesort function to take a type parameter representing sorted lists and operators on them, *e.g.*, merge, singleton, and empty. We define the *characteristic property* of mergesort functions as the existence of such a corresponding abstract mergesort function satisfying the following conditions (Sections 3.2 and 4.3.1):

(1) it is the mergesort function when instantiated with "merge"; that is, the abstract mergesort instantiates back to the concrete mergesort,
(2) it is the identity function when instantiated with concatenation; that is, recursively dividing the input list and concatenating them, instead of merging them, gives us the input list, and
(3) it is relationally parametric [Reynolds 1983] (Section 2.2).

Authors' Contact Information: Cyril Cohen, Inria, CNRS, ENS de Lyon, UCBL, LIP, UMR 5668, France, Cyril.Cohen@inria.fr; Kazuhiko Sakaguchi, CNRS, ENS de Lyon, UCBL, LIP, UMR 5668, France, kazuhiko.sakaguchi@cnrs.fr.

This work is licensed under a Creative Commons Attribution 4.0 International License.

Our correctness proof of mergesort is twofold. Firstly, we prove the characteristic property for each mergesort function we wish to verify (Sections 3.3, 4.3.2 and 4.3.3). Among the three conditions above, Equations (1) and (2) can be proved by functional induction and equational reasoning using basic facts about lists, for all the variants of mergesort presented in the paper, and (3) follows from the abstraction theorem [Reynolds 1983] for all ground implementations. Furthermore, (1) holds by definition in the simplest cases. Secondly, we prove correctness results for any mergesort function, whose sole assumption is the characteristic property (Sections 3.4 and 4.3.4). The characteristic property implies an induction principle over *traces*—binary trees reflecting the underlying divide-and-conquer structure of mergesort—to reason about the relation between the input and output of mergesort (Lemmas 3.5 and 4.1), and the naturality of sort (Lemma 3.14). These two consequences are sufficient to deduce several correctness results of mergesort, including stability.

While we first present a simplified version of characterization and proofs that work only for non-tail-recursive mergesort (Section 3), we later extend them (Section 4.3) to the following optimization techniques for mergesort.

- Tail-recursive mergesort (Section 4.1) in call-by-value evaluation, *e.g.*, `List.stable_sort` of the OCaml [Leroy et al. 2024] standard library, has the advantage that it does not use up stack space and thus is efficient.
- Smooth mergesort (Section 4.2) reuses sorted slices in the input in the sorting process and is efficient on almost sorted input. For example, `Data.List.sort` of the GHC (Glasgow Haskell Compiler) [The GHC Team 2024a] libraries is a smooth non-tail-recursive mergesort.

Note that non-tail-recursive mergesort still has the advantage over tail-recursive mergesort that it allows us to compute a first few elements of the output incrementally in call-by-need evaluation. The problem of obtaining the k smallest items of a list of length n is called *partial sorting*, and its online version that allows us to stop the sorting process and resume it later to increase k is called *incremental sorting* [Paredes and Navarro 2006]. In fact, non-tail-recursive mergesort achieves the optimal time complexity $O(n + k \log k)$ of these problems. Thus, the optimal mergesort function depends on the evaluation strategy and whether one wants to use it as a partial or incremental sort. From this perspective, the advantage of our verification approach is that it allows us to verify both optimal implementations of mergesort modularly.

We formalized our functional correctness proofs in the Rocq Prover [The Rocq Development Team 2025a]. In Section 5, we discuss two technical aspects of our formalization. Firstly, we review a technique [Gonthier 2009] to convince Rocq that bottom-up non-tail-recursive mergesort is terminating (Section 5.1.2), which does not require explicit termination proofs (*i.e.*, no use of well-founded recursion), but is done by making the mergesort function structurally recursive [Giménez 1994] without the use of artificial decreasing argument (fuel). Avoiding the use of well-founded recursion has two practical advantages: 1) it makes execution of mergesort inside the Rocq kernel easier and more efficient, and 2) it makes application of the parametricity translation [Bernardy et al. 2012; Bernardy and Lasson 2011; Keller and Lasson 2012] of the Paramcoq plugin [Keller et al. 2014] to the abstract mergesort function straightforward. Furthermore, this technique of making mergesort structurally recursive allows us to implement bottom-up tail-recursive mergesort (Section 5.1.3), and both bottom-up non-tail-recursive and tail-recursive mergesorts can be easily made smooth (Appendices D.1 and D.2) in contrast to top-down mergesorts, *e.g.*, Section 4.1 and `List.stable_sort` of OCaml. Up to our knowledge, Appendix D.2 provides the first implementation of smooth tail-recursive mergesort for lists in functional programming (Table 1). Secondly, we discuss the design and organization of the library (Section 5.2), particularly, the interface for mergesort functions (Section 5.2.1) which allows us to state our correctness lemmas polymorphically for any stable mergesort functions (Section 5.2.3). To construct an instance of this interface

Table 1. Classification of mergesort functions and their optimization techniques, by whether they are top-down or bottom-up, tail-recursive or not, and smooth or not. ✓ means that the classification applies to *one of* the given implementations, and ✗ means that the classification does not apply. Incompatible classifications, *e.g.*, top-down and bottom-up, may apply to the same row, since one row may correspond to a few implementations (the number indicated in the "# implem." column). Since only bottom-up mergesorts can be made smooth, "non-smooth" and "smooth" appear as the sub-columns of the "bottom-up" column.

	# implem.	top-down	bottom-up		non-tail-rec.	tail-rec.
			non-smooth	smooth		
Section 3.1	2	✓	✓	✗	✓	✗
Section 4.1	1	✓	✗	✗	✗	✓
Section 4.2	1	✗	✗	✓	✓	✗
Section 5.1.2	1	✗	✓	✗	✓	✗
Section 5.1.3	1	✗	✓	✗	✗	✓
Appendix D.1 (incl. Section 5.1.2)	4	✗	✓	✓	✓	✗
Appendix D.2 (incl. Section 5.1.3)	4	✗	✓	✓	✗	✓
OCaml (`List.stable_sort`)	1	✓	✗	✗	✗	✓
GHC (`Data.List.sort`) and Leino and Lucio [2015]; Sternagel [2013]	1	✗	✗	✓	✓	✗
Nipkow et al. [2025] (incl. Sternagel [2013])	3	✓	✓	✓	✓	✗
Chlipala [2013]	1	✓	✗	✗	✓	✗
Leroy [[n. d.]]	1	✗	✓	✗	✓	✗

for a concrete mergesort function, we prove (1) and (2) by induction and equational reasoning, and use Paramcoq to automatically prove (3) (Section 5.2.2).

Our approach currently has two limitations (Section 6). Firstly, the proof of (1) cannot be done by definition (by computation in Rocq) for elaborate variations of mergesort, such as tail-recursive (Section 4.1), smooth (Section 4.2), and structurally-recursive (Sections 5.1.2 and 5.1.3) mergesorts. This is because these mergesorts inspect sorted lists by operations not supplied to the abstract mergesort function, and we have to abstract out these mergesorts in a non-trivial way. Since we are primarily interested in structurally-recursive mergesorts in our formalization, the proof of (1) has to be done by hand for most variations. Secondly, our approach does not support the use of n-way merge function that merges $n > 2$ lists and cannot be expressed as a simple combination of the 2-way merge, *e.g.*, `Data.List.sort` of GHC 9.12.1 and later which uses 3-way and 4-way merge functions. In this paper, "`Data.List.sort` of GHC" generally refers to its old version.

To summarize, Table 1 classifies the mergesort functions and their optimization techniques presented in this paper and related work.

Disclaimer. While we extensively use the algorithmic complexities and performance of mergesort as the *motivation* of the present work (especially Section 4), the contribution of the paper is the functional correctness proofs, and we decided to neither informally nor formally show any complexity results of mergesort, beyond describing the well-known fact that the worst-case time complexity of mergesort is $O(n \log n)$ (Section 3.1). Our rationale is that 1) such an addition would obscure the contribution of the paper, 2) our characterization does not rule out sorting algorithms slower than $O(n \log n)$, *e.g.*, insertion sort (Definition 3.1), and 3) the only case where we improve the worst-case time complexity beyond $O(n \log n)$ is non-tail-recursive mergesort in call-by-need evaluation as optimal incremental sorting.

Outline. The paper is organized as follows. In Section 2, we introduce the OCaml functions used in the paper for manipulating lists (Section 2.1) and parametricity (Section 2.2). In Section 3, we

present a simplified version of our characterization (Section 3.2) and correctness proofs (Section 3.4) that work only for non-tail-recursive mergesort. As examples considered in this section, we review top-down and bottom-up mergesort whose underlying traces have different shapes (Section 3.1), and describe how to prove the characteristic property on them (Section 3.3). In Section 4, we review two optimization techniques for mergesort, namely, tail-recursive mergesort (Section 4.1) and smooth mergesort (Section 4.2), and extend the characterization and proof technique presented in Section 3 to these optimization techniques (Section 4.3). In Section 5, we discuss technical aspects of our formalization. We show how to make mergesort structurally recursive (Section 5.1) and discuss the design and organization of the library (Section 5.2). In Section 6, we discuss the limitations of our approach. In Section 7, we discusses related work before concluding the paper in Section 8.

Because of the structure of our correctness proofs, the paper roughly consists of two parts: sections concerning concrete mergesorts, including optimization techniques and the proofs that they satisfy our characteristic property (Sections 3.1, 3.3, 4.1, 4.2, 4.3.2, 4.3.3 and 5.1), and sections concerning the characteristic property and how it solely implies several correctness results (Sections 3.2, 3.4, 4.3.1, 4.3.4 and 5.2). In order to present, extend, and formalize our proof technique incrementally, these two kinds of sections appear alternately in the paper. The former sections are marked with † to guide the readers.

Appendices. Appendix A provides a list of basic definitions and lemmas in the MathComp library [Mahboubi and Tassi 2022] used for the proofs in Sections 3.3, 3.4 and 4.3.4 and Appendix B. While we refer definitions and lemmas from this appendix in proofs for the sake of preciseness, these references come with enough extra information, *e.g.*, Lemma A.13 (associativity of ++), to be safely ignored. Appendix B provides the list of all lemmas about stable sort functions solely derived from the characterization, their formal statements in Rocq, and their informal proofs. In Appendix C, we compare the statements of our stability results with ones in the literature [H. Cormen et al. 2022; Leino and Lucio 2015; Leroy [n. d.]; Sternagel 2013] and argue that the former is slightly more general than the latter. In Appendix D, we present the full Rocq implementations of structurally-recursive non-tail-recursive and tail-recursive mergesorts including smooth ones, since Section 5.1 presents only the non-smooth ones in the OCaml syntax.

2 Preliminaries

Throughout the paper except appendix, mergesort functions are presented in the OCaml syntax. This preliminary section introduces the OCaml functions used in the paper for manipulating lists (Section 2.1) and binary relational parametricity (Section 2.2).

Without loss of generality, we consider only sorting functions that take a "less than or equal" relation of type $T \to T \to$ bool, which we denote by $\leq$ or `(<=)`, and sort items in ascending order.

2.1 Functons from the List library

The mergesort functions presented in the paper use the following OCaml functions for manipulating lists. These functions are defined in the `List` module of the standald library, except for `List.split_n` defined in the core standard library overlay [Jane Street Group, LLC 2024].

```
val length : 'a list -> int
```
`List.length l` returns the length (number of elements) of `l`.

```
val rev : 'a list -> 'a list
```
List reversal.

```
val append : 'a list -> 'a list -> 'a list
```
Concatenates two lists.

```
val rev_append : 'a list -> 'a list -> 'a list
```
`List.rev_append l1 l2` reverses `l1` and concatenates it with `l2`.
```
val flatten : 'a list list -> 'a list
```
`List.flatten` returns the list obtained by flattening (folding by concatenation) the given list of lists.
```
val split_n : 'a list -> int -> 'a list * 'a list
```
`List.split_n l n` splits `l` into two lists of the first `n` elements and the remaining, and returns the pair of them.
```
val map : ('a -> 'b) -> 'a list -> 'b list
```
`List.map f [a1; ...; an]` returns the list `[f a1; ...; f an]`.
```
val filter : ('a -> bool) -> 'a list -> 'a list
```
`List.filter p l` returns the list collecting all the elements of `l` that satisfy the predicate `p`, and preserves the order of the elements in the input.

Hereinafter, we omit the prefix `List` for the above functions, assuming that the `List` module is open. Among these, we assume that `length`, `rev`, and `rev_append` are tail recursive, and `append`, `flatten`, `split_n`, `map` and `filter` are not, as in their standard implementations. The basic definitions and facts in Rocq, including the Rocq counterpart of the above functions, used for our correctness proofs are listed in Appendix A.

We use the following notations in the part of the paper concerning proofs:

$$\begin{aligned} \texttt{map}_f &:= \texttt{map f}, & [f\,x \mid x \leftarrow xs] &:= \texttt{map}_f\,xs, \\ \texttt{filter}_p &:= \texttt{filter p}, & [x \mid x \leftarrow xs, p\,x] &:= \texttt{filter}_p\,xs, \\ xs +\!\!+\, ys &:= \texttt{append}\,xs\,ys, & [f\,x \mid x \leftarrow xs, p\,x] &:= \texttt{map}_f\,(\texttt{filter}_p\,xs). \end{aligned}$$

2.2 Parametricity

We assume the existence of an interpretation function $[\![\,\cdot\,]\!]$ from types in $\mathcal{U}$ (resp. type families ranging in $\mathcal{U}$) of our calculus to (resp. families of) relations on terms, which is called a binary parametricity translation. In this paper, we generally abbreviate "binary parametricity" to "parametricity". Arrow types $S \to T$ are interpreted as morphisms for relations $[\![\,S\,]\!]$ and $[\![\,T\,]\!]$, *i.e.*, given $f_1, f_2 : S \to T$, we have $(f_1, f_2) \in [\![\,S \to T\,]\!]$ iff $\forall x_1\,x_2, (x_1, x_2) \in [\![\,S\,]\!] \Rightarrow (f_1\,x_1, f_2\,x_2) \in [\![\,T\,]\!]$. Finally, polymorphic types are interpreted as follows: given $f_1, f_2 : (\forall S : \mathcal{U}, T)$, we have $(f_1, f_2) \in [\![\,\forall S : \mathcal{U}, T\,]\!]$ iff $\forall\,(S_1\,S_2 : \mathcal{U})\,(\sim_S\, \subseteq S_1 \times S_2), (f_1\,S_1, f_2\,S_2) \in [\![\,T\,]\!]$, where $[\![\,T\,]\!]$ interprets S as $[\![\,S\,]\!] := \sim_S$. For a full account on parametric interpretations and parametricity translations in this style, we refer to Reynolds [1983] and Wadler [1989].

Definition 2.1 (Parametricity). We say a term t of type T is parametric if $(t, t) \in [\![\,T\,]\!]$.

We also assume that the interpretation of ground types (*e.g.*, $[\![\,\text{bool}\,]\!]$ and $[\![\,\text{nat}\,]\!]$) is equality, and that $[\![\,\text{list}\,T\,]\!] := [\![\,\text{list}\,]\!]_{[\![\,T\,]\!]}$ is the pointwise lifting of relation $[\![\,T\,]\!]$ to lists, defined as follows: $([x_1, \ldots, x_n], [y_1, \ldots, y_m]) \in [\![\,\text{list}\,]\!]_\sim$ iff the two lists have the same length $n = m$ and they are pairwise related, *i.e.*, $x_i \sim y_i$ for any $1 \leq i \leq n$.

In Sections 3 and 4, we assume we work in a fragment of System F, extended with recursors, which satisfies the abstraction theorem.

Assumption 2.2 (Abstraction theorem). Closed terms in the empty context are parametric.

Relations obtained by the interpretation $[\![\,\cdot\,]\!]$ cannot be expressed in such a calculus, and thus, we consider them as expressed in the meta language. However, in Section 5.2, we formalize our arguments in Rocq where relations are interpreted as functions $S \to T \to \mathcal{U}$, which justifies the

abuse of ∀ notation for polymorphic types in the calculus, *e.g.*, $\forall T : \mathcal{U}, S \to V$, and universally quantified formulas in the meta language, *e.g.*, $\forall T : \mathcal{U}, \phi \Rightarrow \psi$.

3 Non-tail-recursive mergesorts

In this section, we present a simplified version of the characteristic property (Section 3.2) and the correctness proofs (Sections 3.3 and 3.4) that work only for non-tail-recursive mergesort. As variations of mergesort covered by this section, we consider top-down and bottom-up non-tail-recursive mergesorts (Section 3.1). The proofs are twofold. Firstly, we prove the characteristic property for each mergesort function (Section 3.3). Secondly, we prove that any mergesort function satisfying the characteristic property is correct (Section 3.4). We will later extend our approach (Section 4.3.1) to support tail-recursive mergesorts (Section 4.1) and smooth mergesorts (Section 4.2).

3.1 Top-down and bottom-up approaches†

Mergesort is a sort algorithm that works by merging sorted sequences. Merging two input lists `xs` and `ys` sorted w.r.t. (with regard to) a total preorder `(<=)`, *i.e.*, a total and transitive binary relation, obtains another sorted list which contains each element of the input lists exactly one time, as in Figure 1a. In order to sort an arbitrary input list, mergesort has to divide the input into sorted slices, *e.g.*, singleton lists, and recursively merge them. There are two approaches to do so: *top-down* and *bottom-up*. The top-down approach implemented in Figure 2a (1) divides the input into two lists of roughly the same length, (2) recursively sorts each half, and (3) merges two sorted halves. The input is eventually divided into singleton lists, and recursion stops there. The bottom-up approach implemented in Figure 2b (1) divides the input into singleton lists, (2) obtains sorted lists of length 2, then 4, then 8, and so on, by merging each adjacent pair, and (3) eventually obtains one sorted list.

In the top-down mergesort, two lists being merged have the same length, or the latter has one extra element. In the bottom-up mergesort, the length of the former list is a power of 2 and is not less than the length of the latter list. Therefore, when the length of the input is not a power of two, these two mergesort functions split the input differently, which can be illustrated using *traces* as in Figure 3. A trace is a binary tree that reflects the divide-and-conquer structure of computation of mergesort, whose leaves and internal nodes denote a singleton list and merge, respectively. A trace of the top-down mergesort is always an AVL tree; that is, the heights of the two children of any node differ by at most one. In contrast, the left child of any node is a perfect binary tree in a trace of the bottom-up mergesort. Regardless of the approach, the height of any trace must be logarithmic in the length n of the input to achieve the optimal time complexity, *e.g.*, the height is always $\lceil \log_2 n \rceil$ in our implementations in Figure 2. Since the merge function costs a linear time in the sum of lengths of input, merge operations of each level in the trace cost a linear time $O(n)$ in total. Thus, we can conclude that mergesort has a quasilinear time complexity $O(n \log n)$. More rigorous complexity analyses of mergesort that use a recurrence relation can be found in some textbooks on algorithms, *e.g.*, [H. Cormen et al. 2022, Section 2.3.2].

3.2 Characterization of stable non-tail-recursive mergesort functions

In this section, we present the characterization of stable non-tail-recursive mergesort functions. A sort algorithm is *stable* if equivalent elements, such as x and y satisfying $x \equiv y \coloneqq x \leq y \land y \leq x$, always appear in the same order in the input and output. To maintain the stability of a mergesort function, one first has to notice that the ordering of the last two arguments of `merge` (`xs` and `ys` in Figure 1a) matters. When two elements being compared in `merge` are equivalent, the element from the first list `xs` appears earlier than the other in the output. Thus, these two lists being merged should be obtained by sorting two adjacent (former and latter) slices of the input, respectively.

```
let rec merge (<=) xs ys =
  match xs, ys with
  | [], ys -> ys
  | xs, [] -> xs
  | x :: xs', y :: ys' ->
    if x <= y then
      x :: merge (<=) xs' ys
    else
      y :: merge (<=) xs ys'
```

(a) Non-tail-recursive merge.

```
let rec revmerge (<=) xs ys accu =
  match xs, ys with
  | [], ys -> rev_append ys accu
  | xs, [] -> rev_append xs accu
  | x :: xs', y :: ys' ->
    if x <= y then
      revmerge (<=) xs' ys (x :: accu)
    else
      revmerge (<=) xs ys' (y :: accu)
```

(b) Tail-recursive merge.

Fig. 1. Non-tail-recursive and tail-recursive merge functions.

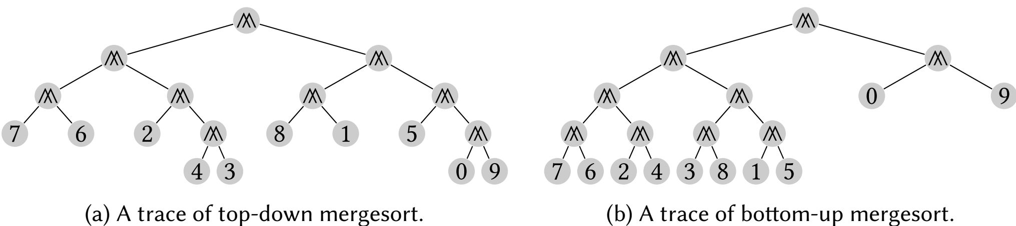
```
let rec sort (<=) = function
  | [] -> []
  | [x] -> [x]
  | xs ->
    let k = length xs / 2 in
    let (xs1, xs2) = split_n xs k in
    merge (<=)
      (sort (<=) xs1)
      (sort (<=) xs2)
```

(a) Top-down mergesort.

```
let sort (<=) xs =
  let rec merge_pairs = function
    | a :: b :: xs ->
      merge (<=) a b :: merge_pairs xs
    | xs -> xs in
  let rec merge_all = function
    | [] -> []
    | [x] -> x
    | xs -> merge_all (merge_pairs xs)
  in
  merge_all (map (fun x -> [x]) xs)
```

(b) Bottom-up mergesort.

Fig. 2. Naive implementations of top-down and bottom-up mergesort algorithms.



(a) A trace of top-down mergesort.

(b) A trace of bottom-up mergesort.

Fig. 3. The traces of the sorting processes for the input list `[7; 6; 2; 4; 3; 8; 1; 5; 0; 9]` in the naive top-down (a) and bottom-up (b) mergesort algorithms. Each number placed as a leaf denotes a singleton list consisting of it, and $\wedge\!\!\!\wedge$ placed as an internal node denotes the merging of its children. The difference of their shapes reflects the difference of associativity of merge.

Taking this observation into account, the intuition behind the characterization is that the mergesort function can be turned into the identity function by replacing `merge` with concatenation. In other words, we should always be able to get the input by flattening traces, *e.g.*, Figure 3.

In order to ensure this replacement is done in the intended way and to state the characterization formally, we first abstract out the mergesort function `sort` of type:

$$\mathrm{T}_{\mathtt{sort}} \coloneqq \forall (T : \mathcal{U}),\ \underbrace{(T \to T \to \mathrm{bool})}_{\text{comparison function}} \to \underbrace{\mathrm{list}\, T}_{\text{input}} \to \underbrace{\mathrm{list}\, T}_{\text{output}}$$

to `asort` of type:

$$\mathrm{T}_{\mathtt{asort}} \coloneqq \forall (T\, R : \mathcal{U}), \underbrace{(R \to R \to R)}_{\text{merge}} \to \underbrace{(T \to R)}_{\text{singleton}} \to \underbrace{R}_{\text{empty}} \to \underbrace{\mathrm{list}\, T}_{\text{input}} \to \underbrace{R}_{\text{output}}.$$

The abstract sort function `asort` is abstracted over the type R of sorted lists and the basic operations on them, namely, the merge, singleton, and empty constructions. Therefore, we expect to get `sort` by instantiating `asort` with $\wedge\!\!\!\wedge_{\leq}$ ($\coloneqq \mathtt{merge}\ (\leq)$):

$$\forall (T : \mathcal{U})\ (\leq\ : T \to T \to \mathrm{bool})\ (xs : \mathrm{list}\, T), \mathtt{asort}\ (\wedge\!\!\!\wedge_{\leq})\ [\cdot]\ []\ xs = \mathtt{sort}_{\leq}\ xs \tag{1}$$

where $[\cdot]$ and $[]$ are the singleton list function $(\lambda(x : T), [x])$ and the empty list, respectively. The replacement of merge ($\wedge\!\!\!\wedge_{\leq}$) with concatenation ($+\!\!\!+$) can be done by instantiating `asort` with concatenation. We expect to get the identity function by this instantiation:

$$\forall (T : \mathcal{U})\ (xs : \mathrm{list}\, T), \mathtt{asort}\ (+\!\!\!+)\ [\cdot]\ []\ xs = xs. \tag{2}$$

Although both Equations (1) and (2) instantiate the type parameter R of the abstract sort function `asort` with $\mathrm{list}\, T$, abstracting it out as a type parameter is crucial in ensuring that `asort` builds the output by using the three operators on R inductively and uniformly, *i.e.*, in such a way that `asort` is parametric as defined in Section 2.2: $(\mathtt{asort}, \mathtt{asort}) \in [\![\, \mathrm{T}_{\mathtt{asort}}\,]\!]$, which holds iff:

$$\begin{aligned}
&\forall (T_1\, T_2 : \mathcal{U})\ (\sim_T\ \subseteq T_1 \times T_2),\\
&\forall (R_1\, R_2 : \mathcal{U})\ (\sim_R\ \subseteq R_1 \times R_2),\\
&\forall (\mathit{merge}_1 : R_1 \to R_1 \to R_1)\ (\mathit{merge}_2 : R_2 \to R_2 \to R_2), && \text{(merge)}\\
&\quad (\forall (xs_1 : R_1)\ (xs_2 : R_2), xs_1 \sim_R xs_2 \Rightarrow\\
&\quad\ \ \forall (ys_1 : R_1)\ (ys_2 : R_2), ys_1 \sim_R ys_2 \Rightarrow (\mathit{merge}_1\, xs_1\, ys_1) \sim_R (\mathit{merge}_2\, xs_2\, ys_2)) \Rightarrow\\
&\forall ([\cdot]_1 : T_1 \to R_1)\ ([\cdot]_2 : T_2 \to R_2), && \text{(singleton)}\\
&\quad (\forall (x_1 : T_1)\ (x_2 : T_2), x_1 \sim_T x_2 \Rightarrow [x_1]_1 \sim_R [x_2]_2) \Rightarrow\\
&\forall ([]_1 : R_1)\ ([]_2 : R_2), []_1 \sim_R []_2 \Rightarrow && \text{(empty)}\\
&\forall (xs_1 : \mathrm{list}\, T_1)\ (xs_2 : \mathrm{list}\, T_2), (xs_1, xs_2) \in [\![\, \mathrm{list}\,]\!]_{\sim_T} \Rightarrow\\
&\quad (\mathtt{asort}\ \mathit{merge}_1\ [\cdot]_1\ []_1\ xs_1) \sim_R (\mathtt{asort}\ \mathit{merge}_2\ [\cdot]_2\ []_2\ xs_2)
\end{aligned}$$

To sum up, we define the characteristic property of stable mergesort functions as the existence of an abstract mergesort function `asort` that is parametric and satisfies Equations (1) and (2).

3.3 Proofs of the characteristic property†

In this section, we briefly show that the two mergesort functions presented in Figure 2 satisfy the characteristic property presented in Section 3.2. Before doing so, we present an insertion sort (Definition 3.1) as the simplest instance of the characterization (Lemma 3.2).

Definition 3.1 (Insertion sort). Noticing that one-element insertion to a sorted list is a special case of merge, an insertion sort can be defined as follows.

```
let rec sort (<=) = function [] -> [] | x :: xs -> merge (<=) [x] (sort (<=) xs)
```

```
let asort merge singleton empty =
  let rec asort_rec = function
    | [] -> empty
    | [x] -> singleton x
    | xs ->
      let k = length xs / 2 in
      let (xs1, xs2) = split_n xs k in
      merge
        (asort_rec xs1)
        (asort_rec xs2)
  in asort_rec
```

(a) Top-down abstract mergesort.

```
let asort merge singleton empty xs =
  let rec merge_pairs = function
    | a :: b :: xs ->
      merge a b :: merge_pairs xs
    | xs -> xs in
  let rec merge_all = function
    | [] -> empty
    | [x] -> x
    | xs -> merge_all (merge_pairs xs)
  in
  merge_all (map singleton xs)
```

(b) Bottom-up abstract mergesort.

Fig. 4. Two abstract mergesort functions derived from the top-down and bottom-up mergesort functions (Figure 2). The use of abstract operators on sorted lists `merge`, `singleton`, and `empty` are underlined.

Lemma 3.2. *The insertion sort (Definition 3.1) satisfies the characteristic property.*

Proof. We first abstract the three basic operators on sorted list out of Definition 3.1.

```
let rec asort merge singleton empty = function
  [] -> empty | x :: xs -> merge (singleton x) (asort merge singleton empty xs)
```

Equation (1) holds by definition. The parametricity of `asort` follows from the abstraction theorem (Assumption 2.2). Therefore, we are left to prove that Equation (2), *i.e.*, $\phi\, xs = xs$ where $\phi :=$ `asort` $(+\!\!+)\ [\cdot]\ []$, by structural induction on xs. If $xs = []$, it is trivial. Otherwise $xs = x :: xs'$ and

$$
\begin{aligned}
\phi\,(x :: xs') &= [x] +\!\!+ \phi\, xs' && \text{(Definition of asort)}\\
&= x :: \phi\, xs' && \text{(Definition A.12 } (+\!\!+)\text{)}\\
&= x :: xs' && \text{(I.H.)}
\end{aligned}
$$

□

To show that top-down and bottom-up mergesort functions (Figure 2) satisfy the characteristic property, we abstract them out as in Figure 4. Again, we are left to prove that `asort` $(+\!\!+)\ [\cdot]\ []$ is the identity function, for each variation. The Rocq counterpart of this section is Section 5.2.2.

Lemma 3.3 (Equation (2) for the top-down mergesort in Figure 2a). `asort` *in Figure 4a satisfies* `asort` $(+\!\!+)\ [\cdot]\ []\ xs = xs$ *for any* xs.

Proof. Let ϕ be `asort` $(+\!\!+)\ [\cdot]\ []$. The proof is by strong mathematical induction on the length n of xs. If $n < 2$, xs is either an empty or singleton list, and the equation holds by definition. Otherwise, xs is split into the list of the first $k := \lfloor \frac{n}{2} \rfloor$ elements xs_1 and the rest xs_2 (of length $n - k$). Since $0 < k < n$ and thus $n - k < n$, $\phi\, xs_1 = xs_1$ and $\phi\, xs_2 = xs_2$ follow from the induction hypothesis. Therefore, $\phi\, xs = \phi\, xs_1 +\!\!+ \phi\, xs_2 = xs_1 +\!\!+ xs_2$, which is equal to xs (Lemma A.22). □

Lemma 3.4 (Equation (2) for the bottom-up mergesort in Figure 2b). `asort` *in Figure 4b satisfies* `asort` $(+\!\!+)\ [\cdot]\ []\ xs = xs$ *for any* xs.

Proof. First, we prove

$$\texttt{flatten}\,(\texttt{merge_pairs}\; xs) = \texttt{flatten}\; xs \tag{4}$$

for any list of lists xs by structural induction on xs. If xs has length $n < 2$ the equation holds by definition. Otherwise $xs = a :: b :: xs'$ and

$$
\begin{aligned}
& \texttt{flatten}\,(\texttt{merge_pairs}\,xs) \\
&= \texttt{flatten}\,((a +\!\!+ b) :: \texttt{merge_pairs}\,xs') && \text{(Definition Figure 4b)} \\
&= (a +\!\!+ b) +\!\!+ \texttt{flatten}\,(\texttt{merge_pairs}\,xs') && \text{(Definition A.15 (flatten))} \\
&= (a +\!\!+ b) +\!\!+ \texttt{flatten}\,xs' && \text{(I.H.)} \\
&= a +\!\!+ (b +\!\!+ \texttt{flatten}\,xs') && \text{(Lemma A.13 (associativity of } +\!\!+ \text{))} \\
&= \texttt{flatten}\,xs. && \text{(Definition A.15 (flatten))}
\end{aligned}
$$

Second, we prove

$$\texttt{merge_all}\,xs = \texttt{flatten}\,xs \tag{5}$$

for any list of lists xs by strong mathematical induction on the length n of xs. If $n < 2$, xs is either an empty or singleton list, and the equation holds by definition. Otherwise,

$$\texttt{merge_all}\,xs = \texttt{merge_all}\,(\texttt{merge_pairs}\,xs) \qquad \text{(Definition Figure 4b)}$$

Noticing that the length of $\texttt{merge_pairs}\,xs$ is $\lceil \frac{n}{2} \rceil < n$, we can apply the induction hypothesis:

$$
\begin{aligned}
&= \texttt{flatten}\,(\texttt{merge_pairs}\,xs) && \text{(I.H.)} \\
&= \texttt{flatten}\,xs. && \text{(Equation (4))}
\end{aligned}
$$

Finally, we show

$$
\begin{aligned}
\texttt{asort}\,(+\!\!+)\,[\cdot]\,[]\,xs &= \texttt{merge_all}\,[[x] \mid x \leftarrow xs] && \text{(Definition Figure 4b)} \\
&= \texttt{flatten}\,[[x] \mid x \leftarrow xs] && \text{(Equation (5))} \\
&= xs. && \text{(Structural induction on } xs\text{)}
\end{aligned}
$$

□

3.4 Correctness proofs

In this section, we deduce several correctness results of mergesort solely from the characteristic property (Section 3.2). We universally quantify the mergesort function `sort` in this section, and thus, all the correctness results below apply to any mergesort function satisfying the characteristic property, including the top-down and bottom-up mergesorts in Figure 2. The use of the parametricity of the abstract sorting function `asort` in our correctness proofs is twofold: deducing an induction principle over traces (Section 3.4.1), and deducing the naturality of `sort` (Section 3.4.2).

3.4.1 Induction over traces.

Lemma 3.5 (An induction principle over traces of `sort`). *Suppose* $\leq$ *and* $\sim$ *are binary relations on* T *and* list T, *respectively, and* xs *is a list of type* list T. *Then,* $xs \sim \texttt{sort}_{\leq}\,xs$ *holds whenever the following three induction cases hold:*

- *for any lists* xs, xs', ys, *and* ys' *of type* list T, $(xs +\!\!+ ys) \sim (xs' \wedge\!\!\!\wedge_{\leq} ys')$ *holds whenever* $xs \sim xs'$ *and* $ys \sim ys'$ *hold,*
- *for any* x *of type* T, $[x] \sim [x]$ *holds, and*
- $[] \sim []$ *holds.*

Proof. Thanks to Equations (1) and (2), $xs \sim \texttt{sort}_{\leq}\,xs$ holds if and only if

$$\texttt{asort}\,(+\!\!+)\,(\lambda(x : T), [x])\,[]\,xs \sim \texttt{asort}\,(\wedge\!\!\!\wedge_{\leq})\,(\lambda(x : T), [x])\,[]\,xs.$$

We apply the parametricity of `asort` by instantiating $\sim_T$ with the equality over T and $\sim_R$ with $\sim$. The premise $(xs_1, xs_2) \in [\![\, \text{list} \,]\!]_{\sim_T}$ holds because both xs_1 and xs_2 are instantiated with xs and

$[\![\, \mathrm{list} \,]\!]_{\sim_T}$ is just the equality over list T. The other three premises exactly correspond to the three induction cases. □

Remark 3.6. Lemma 3.5 is a variant of the general fact that parametricity implies induction principles on Church-encoded datatypes [Altenkirch et al. 2024; Kaposi et al. 2019; Tassi 2019; Wadler 1990], except that we use the binary version of parametricity to provide a simultaneous induction principle relating the input and output of mergesort. If we instantiate the abstract mergesort with a concrete type of elements T and an input of type list T, we get a term of type

$$\forall (R : \mathcal{U}), (R \to R \to R) \to (T \to R) \to R \to R,$$

which represents Church-encoded binary trees with singleton and empty constructs as their leaves.

As applications of Lemma 3.5, we prove the permutation property of `sort` (Lemma 3.7 and Corollary 3.8) and sortedness and stability results of `sort` (Lemma 3.10, Theorem 3.11, and Corollary 3.12).

Lemma 3.7. *For any relation $\leq$ on type T and xs of type* list T, $\mathtt{sort}_{\leq}\ xs =_{\mathtt{perm}} xs$ *holds; that is,* $\mathtt{sort}_{\leq}\ xs$ *is a permutation of xs* (cf., *Definition A.49 for a precise definition of* $=_{\mathtt{perm}}$*).*

Proof. We prove it by induction on $\mathtt{sort}_{\leq}\ xs$ (Lemma 3.5). Since the last two cases are obvious, suffice it to show that $xs' \mathbin{⩕}_{\leq} ys' =_{\mathtt{perm}} xs \mathbin{+\!\!+} ys$ whenever $xs' =_{\mathtt{perm}} xs$ and $ys' =_{\mathtt{perm}} ys$.

$$xs' \mathbin{⩕}_{\leq} ys' =_{\mathtt{perm}} xs' \mathbin{+\!\!+} ys' \qquad \text{(Lemma A.72)}$$

Since $\mathbin{+\!\!+}$ is congruent with respect to $=_{\mathtt{perm}}$ (Lemma A.54),

$$=_{\mathtt{perm}} xs \mathbin{+\!\!+} ys.$$ □

Corollary 3.8. *For any relation $\leq$ on type T and xs of type* list T, $\mathtt{sort}_{\leq}\ xs$ *has the same set of elements as xs,* i.e., $x \in \mathtt{sort}_{\leq}\ xs$ *iff* $x \in xs$ *for any* $x \in T$.

We define two versions of sortedness of lists to state Lemma 3.10 and Theorem 3.11 in their general form.

Definition 3.9 (Sortedness). Suppose R is a relation on type T. A list $xs \coloneqq [x_0, \ldots, x_n]$ of type list T is said to be:

- *sorted* w.r.t. R if the relation R holds for each adjacent pair, *i.e.*, $x_0\ R\ x_1 \wedge \cdots \wedge x_{n-1}\ R\ x_n$, and
- *pairwise sorted* w.r.t. R if the relation R holds for any x_i and x_j such that $i < j \leq n$, *i.e.*,

$$x_0\ R\ x_1 \wedge \cdots \wedge x_0\ R\ x_n \wedge x_1\ R\ x_2 \wedge \cdots \wedge x_1\ R\ x_n \wedge \cdots \wedge x_{n-1}\ R\ x_n.$$

Lemma 3.10. *For any s of type* list T *pairwise sorted w.r.t.* $\leq\, \subseteq T \times T$, $\mathtt{sort}_{\leq}\ s = s$ *holds.*

Proof. Since we use this lemma only for the proof of Corollary 3.17, we omit the proof, which is done by induction on $\mathtt{sort}_{\leq}\ s$ (Lemma 3.5). See Lemma B.3 for the complete proof. □

Theorem 3.11 (Sortedness and stability of `sort`). *Suppose $\leq_1$ and $\leq_2$ are binary relations on type T, $\leq_1$ is total, and xs is a list of type* list T. *Then,* $\mathtt{sort}_{\leq_1}\ xs$ *is sorted w.r.t. the following lexicographic order:*

$$x \leq_{\mathrm{lex}} y \coloneqq x \leq_1 y \wedge (y \not\leq_1 x \vee x \leq_2 y)$$

whenever xs is pairwise sorted w.r.t. $\leq_2$.

Proof. We prove a generalized proposition:

$$(\mathtt{sort}_{\leq_1}\ xs \subseteq xs) \wedge (xs \text{ is pairwise sorted w.r.t. } \leq_2 \implies \mathtt{sort}_{\leq_1}\ xs \text{ is sorted w.r.t. } \leq_{\mathrm{lex}})$$

by induction on $\mathtt{sort}_{\leq_1}\ xs$ (Lemma 3.5). Since $\subseteq$ is reflexive and a list whose length is less than 2 is always sorted, the last two cases are obvious.

For the first component of the conjunction in the first induction case, suffice it to show that

$$(xs' \subseteq xs) \wedge (ys' \subseteq ys) \Rightarrow (xs' \mathbin{⩕}_{\leq_1} ys' \subseteq xs +\!\!+\, ys)$$

which is obvious since $xs' \mathbin{⩕}_{\leq_1} ys'$ is a permutation of $xs' +\!\!+\, ys'$ (Lemma A.72).

For the second component of the conjunction, suffice it to show that $xs' \mathbin{⩕}_{\leq_1} ys'$ is sorted w.r.t. $\leq_{\text{lex}}$ whenever:

(i) xs' (resp. ys') is a subset of xs (resp. ys),
(ii) xs' (resp. ys') is sorted w.r.t. $\leq_{\text{lex}}$ if xs (resp. ys) is pairwise sorted w.r.t. $\leq_2$, and
(iii) $xs +\!\!+\, ys$ is pairwise sorted w.r.t. $\leq_2$.

Among these, (iii) is equivalent to the following conjunction (Lemma A.58):

(iv) both xs and ys are pairwise sorted w.r.t. $\leq_2$, and
(v) $x \leq_2 y$ holds for any $x \in xs$ and $y \in ys$.

Hypotheses (ii) and (iv) imply that both xs' and ys' are sorted w.r.t. $\leq_{\text{lex}}$. Hypotheses (i) and (v) imply that $x \leq_2 y$ holds for any $x \in xs'$ and $y \in ys'$ (Lemma A.30). These two facts suffice to show that $xs' \mathbin{⩕}_{\leq_1} ys'$ is sorted w.r.t. $\leq_{\text{lex}}$ (Lemma A.74). □

The following corollary also holds since the sortedness and the pairwise sortedness are equivalent for any transitive relation (Lemma A.60).

Corollary 3.12. *Suppose $\leq_1$ and $\leq_2$ are binary relations on type T, $\leq_1$ is total, $\leq_2$ is transitive, and xs is a list of type* list T. *Then,* $\texttt{sort}_{\leq_1}$ *xs is sorted w.r.t. the lexicographic order $\leq_{\text{lex}}$ of $\leq_1$ and $\leq_2$ whenever xs is sorted w.r.t. $\leq_2$.*

3.4.2 Naturality. The induction principle over traces (Lemma 3.5) does not allow us to relate two sorting processes that behave parametrically. Instead, we obtain the parametricity (Lemma 3.13) and the naturality (Lemma 3.14) of `sort` from the parametricity of `asort`.

Lemma 3.13 (The parametricity of `sort`). *Any* `sort` *function satisfying the characteristic property is parametric (Section 2.2),* i.e., $(\texttt{sort}, \texttt{sort}) \in [\![\, \mathrm{T}_{\texttt{sort}} \,]\!]$, *which holds iff:*

$$\begin{aligned}
&\forall (T_1\, T_2 : \mathcal{U})\, ({\sim_T} \subseteq T_1 \times T_2),\\
&\forall ({\leq_1} : T_1 \to T_1 \to \mathrm{bool})\, ({\leq_2} : T_2 \to T_2 \to \mathrm{bool}),\\
&\quad (\forall (x_1 : T_1)\, (x_2 : T_2), x_1 \sim_T x_2 \Rightarrow \forall (y_1 : T_1)\, (y_2 : T_2), y_1 \sim_T y_2 \Rightarrow (x_1 \leq_1 y_1) = (x_2 \leq_2 y_2)) \Rightarrow\\
&\forall (xs_1 : \mathrm{list}\, T_1)\, (xs_2 : \mathrm{list}\, T_2), (xs_1, xs_2) \in [\![\, \mathrm{list} \,]\!]_{\sim_T} \Rightarrow (\texttt{sort}_{\leq_1} xs_1, \texttt{sort}_{\leq_2} xs_2) \in [\![\, \mathrm{list} \,]\!]_{\sim_T}.
\end{aligned}$$

Proof. $\texttt{sort}_{\leq_i}$ is extensionally equal to $\texttt{asort}\, (\mathbin{⩕}_{\leq_i})\, (\lambda(x : T), [x])\, []$ for each $i \in \{1, 2\}$ thanks to the characterization. Since `asort` and its arguments are parametric, `sort` is parametric as well. □

Lemma 3.14 (The naturality of `sort`). *Suppose $\leq_T$ is a relation on type T, f is a function from T' to T, and xs is a list of type* list T'. *Then, the following equation holds:*

$$\texttt{sort}_{\leq_T} [f\, x \mid x \leftarrow xs] = [f\, x \mid x \leftarrow \texttt{sort}_{\leq_{T'}} xs]$$

where $x \leq_{T'} y := f\, x \leq_T f\, y$.

Proof. We instantiate Lemma 3.13 with $T_1 := T$, $T_2 := T'$, $x \sim_T y := (x = f\, y)$, ${\leq_1} := {\leq_T}$, and ${\leq_2} := {\leq_{T'}}$. Then, the first premise is equivalent to the definition of $\leq_{T'}$. The second premise $(xs_1, xs_2) \in [\![\, \mathrm{list} \,]\!]_{\sim_T}$ is equivalent to $xs_1 = [f\, x \mid x \leftarrow xs_2]$. By substituting this equation to the conclusion, we get $\texttt{sort}_{\leq_T} [f\, x \mid x \leftarrow xs_2] = [f\, x \mid x \leftarrow \texttt{sort}_{\leq_{T'}} xs_2]$. □

Remark 3.15. Lemma 3.14 is known as a free theorem [Wadler 1989, Section 3.3] for type $\forall(T : \mathcal{U}), (T \to T \to \text{bool}) \to \text{list}\, T \to \text{list}\, T$. It is also a case where parametricity implies naturality [Reddy 1997].

Theorem 3.16. *For any total preorder $\leq$ on T and predicate p on T, $\texttt{filter}_p$ commutes with $\texttt{sort}_\leq$ under function composition; that is, the following equation holds for any xs of type $\text{list}\, T$:*

$$\texttt{filter}_p\, (\texttt{sort}_\leq\, xs) = \texttt{sort}_\leq\, (\texttt{filter}_p\, xs).$$

Proof. We will rely on the fact that two lists are equal whenever they are sorted w.r.t. a transitive and irreflexive relation and contain the same set of elements (Lemma A.64).

Since $\leq$ is total and hence reflexive, we instead consider a relation on natural numbers (indices):

$$i \leq_I j \coloneqq \texttt{nth}\, x_0\, xs\, i \leq \texttt{nth}\, x_0\, xs\, j$$

where $\texttt{nth}\, x_0\, xs\, i$ is the ith element in the list xs (Definition A.66) and its default value x_0 is an arbitrary element from the list xs. We can turn this relation into an irreflexive relation by composing it lexicographically with the strict order on natural numbers $<_\mathbb{N}$, resulting in the relation

$$i <_I j \coloneqq i \leq_I j \land (j \not\leq_I i \lor i <_\mathbb{N} j).$$

Now we replace xs everywhere with $\texttt{map}_{\texttt{nth}\, x_0\, xs}\, is$ where $is \coloneqq [0, \ldots, |xs| - 1]$ (Lemma A.67). It thus remains to prove

$$\texttt{filter}_p\, (\texttt{sort}_\leq\, (\texttt{map}_{\texttt{nth}\, x_0\, xs}\, is)) = \texttt{sort}_\leq\, (\texttt{filter}_p\, (\texttt{map}_{\texttt{nth}\, x_0\, xs}\, is)).$$

Using the naturality of $\texttt{sort}$ (Lemma 3.14) and $\texttt{filter}$ (Lemma A.37), *i.e.*, $\texttt{map}_f\, (\texttt{filter}_{p \circ f}\, xs) = \texttt{filter}_p\, (\texttt{map}_f\, xs)$, it remains to prove

$$\texttt{map}_{\texttt{nth}\, x_0\, xs}\, (\texttt{filter}_{p_I}\, (\texttt{sort}_{\leq_I}\, is)) = \texttt{map}_{\texttt{nth}\, x_0\, xs}\, (\texttt{sort}_{\leq_I}\, (\texttt{filter}_{p_I}\, is))$$

where $p_I \coloneqq p \circ \texttt{nth}\, x_0\, xs$.

Now, we apply the congruence rule with respect to $\texttt{map}$ and the fact mentioned in the beginning of this proof (Lemma A.64) by checking that both sides of the above equation are sorted w.r.t. $<_I$ and they contain the same set of elements. The latter condition follows from Corollary 3.8 and the fact that $x \in \texttt{filter}_p\, xs$ iff $p\, x \land x \in xs$ for any p, xs, and x (Lemma A.38). Finally, both lists are sorted w.r.t. $<_I$ because $\texttt{sort}$ is stable (Corollary 3.12), $is \coloneqq [0, \ldots, |xs| - 1]$ is sorted w.r.t. $<_\mathbb{N}$ (Lemma A.69), and $\texttt{filter}$ turns a sorted list into a sorted list whenever the relation is transitive (Lemma A.62). □

3.4.3 A remark on the formulation of stability. The stability of a sort function is often [H. Cormen et al. 2022; Leino and Lucio 2015; Leroy [n. d.]; Sternagel 2013] formulated as follows.

Corollary 3.17 (The standard formulation of the stability). *For any total preorder $\leq$ on T, the equivalent elements always appear in the same order in the input and output of sorting; that is, the following equation holds for any x of type T and s of type $\text{list}\, T$:*

$$[y \leftarrow \texttt{sort}_\leq\, s \mid x \equiv y] = [y \leftarrow s \mid x \equiv y].$$

Proof.

$$[y \leftarrow \texttt{sort}_\leq\, s \mid x \equiv y] = \texttt{sort}_\leq\, [y \leftarrow s \mid x \equiv y] \qquad \text{(Theorem 3.16)}$$

$[y \leftarrow s \mid x \equiv y]$, whose elements are all equivalent, is pairwise sorted w.r.t. $\leq$. Thus,

$$= [y \leftarrow s \mid x \equiv y] \qquad \text{(Lemma 3.10)}$$

□

We argue that our stability results (Theorems 3.11 and 3.16) are more general than Corollary 3.17 for the following reasons:

- While Corollary 3.17 restricts the predicate to $(x \equiv \cdot)$, Theorem 3.16 applies to any predicate and opened up a natural way to prove other useful stability results (Lemmas B.23 to B.26).
- In the above proofs, Corollary 3.17 is an easy consequence of Theorem 3.16, which follows from Theorem 3.11.
- While we proved the converse implications, *i.e.*, Corollary 3.17 implies Theorems 3.11 and 3.16 under some assumptions on `sort`, their proofs are non-trivial, and Theorem 3.11 derived from Corollary 3.17 requires $\leq_1$ to be transitive (see Appendix C).

4 Optimizations

In this section, we review some optimization techniques for mergesort, and see how our proof technique presented in Section 3 extends to them (Section 4.3). In Section 4.1, we review tail-recursive mergesort in call-by-value evaluation, which does not use up stack space and thus is efficient. In Section 4.2, we review smooth mergesort, that reuses sorted slices in the input in the sorting process. In Section 4.3.1, we extend the characterization presented in Section 3.2 to support the optimized mergesorts. In Sections 4.3.2 and 4.3.3, we describe how the tail-recursive and smooth mergesorts satisfy the extended characteristic property, respectively. In Section 4.3.4, we demonstrate that the extended characteristic property implies the same correctness results as Section 3.4.

4.1 Tail-recursive mergesort†

Although the naive mergesort algorithms presented in Section 3.1 achieves optimal $O(n \log n)$ time complexity, the `merge` function (Figure 1a) in call-by-value evaluation consumes a linear amount of stack space and crashes on longer inputs, since it is not tail recursive. The commonly used technique to make it tail recursive is to add an *accumulator* argument `accu` that is initially the empty list and accumulates the result, as in `revmerge` in Figure 1b.[1] The tail-recursive merge function accumulates the first elements of the input lists as the last element of the output list, and produces its output in reverse order. That is to say, the following equation holds for any binary relation `(<=)` and any lists `xs` and `ys`:

```
revmerge (<=) xs ys [] = rev (merge (<=) xs ys).
```

Two lists sorted in descending order can be merged without reversing them using the converse relation `(>=) := (fun x y -> y <= x)`. In fact, the following equation holds for any total preorder `(<=)`, and any lists `s1`, `s2`, `s3`, and `s4` sorted w.r.t. `(<=)`:[2]

```
  merge (<=) (merge (<=) s1 s2) (merge (<=) s3 s4)
= revmerge (>=) (revmerge (<=) s3 s4 []) (revmerge (<=) s1 s2 []) [].
```

There are other non-tail-recursive functions in the naive mergesort algorithms. In Figure 2a, the splitting function `split_n` is not tail recursive and costs a linear time in its second argument `k`. In Figure 2b, the `merge_pairs` and `map` functions are not tail recursive. Note that the non tail-recursiveness of the top-down `sort` function (or `sort_rec` below) is not an actual issue since the depth of its recursive calls is logarithmic in the length of the input. Therefore, based on the top-down approach, all the major inefficiency issues explained here can be addressed as follows.

[1] Although `rev_append xs ys` is equal to `append (rev xs) ys` (Lemma A.18), the former is tail recursive but the latter is not. Therefore, the `revmerge` function has to use the former to avoid using up stack space.

[2] Note that swapping the arguments of the outer `revmerge` matters for maintaining the stability of the algorithm.

```
let sort (<=) xs =
  let (>=) = (fun x y -> y <= x) in
  let rec sort_rec xs b n =
    match n, xs with
    | 1, x :: xs' -> [x], xs'
    | _, _ ->
      let n1 = n / 2 in
      let s1, xs'  = sort_rec xs (not b) n1 in
      let s2, xs'' = sort_rec xs' (not b) (n - n1) in
      (if b then revmerge (>=) s2 s1 [] else revmerge (<=) s1 s2 []), xs''
  in
  if xs = [] then [] else fst (sort_rec xs true (length xs))
```

The auxiliary recursive function `sort_rec` takes three arguments: a list `xs`, a Boolean value `b`, a positive integer `n` that must be less than or equal to the length of `xs`. It returns the pair of the sorted list of the first `n` elements of `xs` and the rest of the input. The sorted list (first component) is in ascending order if `b` is `true`, otherwise in descending order.

`List.stable_sort` of the OCaml standard library follows the same approach as above. Additionally, its auxiliary function corresponding to `sort_rec` above is defined as two mutually-recursive functions corresponding to the cases that `b` is `true` or `false`, respectively. It stops the recursion when $n \leq 3$, which is an effective micro-optimization. We will present a bottom-up tail-recursive mergesort in Section 5.1.3 and make it smooth (Section 4.2) in Appendix D.2.

4.2 Smooth mergesort†

A mergesort algorithm that takes advantage of sorted slices in the input is called *natural* [Knuth 1973, Algorithm N in Section 5.2.4] mergesort. In this paper, we instead call it *smooth* [Dijkstra 1982][Paulson 1996, Subsection "Bottom-up merge sort" in Section 3.21] mergesort to avoid confusion with naturality (Lemma 3.14). Bottom-up non-tail-recursive mergesort (Figure 2b) can easily be made smooth by dividing the input into weakly increasing or strictly decreasing slices instead of singleton lists. Such a slice of the input that is already sorted is called a *run*. Note that we cannot use a non-strictly decreasing slice, because its reversal does not preserve the order of equivalent elements in the slice, and thus, it breaks the stability of the algorithm. An example of smooth bottom-up non-tail-recursive mergesort follows:

```
let sort (<=) xs =
  let rec merge_pairs = ... in (* These functions remain    *)
  let rec merge_all = ... in   (* unchanged from Figure 2b. *)
  let rec sequences = function
    | a :: b :: xs -> if a <= b then ascending b [a] xs else descending b [a] xs
    | [a] -> [[a]]
    | [] -> []
  and ascending a accu = function
    | b :: xs when a <= b -> ascending b (a :: accu) xs
    | xs -> rev (a :: accu) :: sequences xs
  and descending a accu = function
    | b :: xs when not (a <= b) -> descending b (a :: accu) xs
    | xs -> (a :: accu) :: sequences xs
  in
  merge_all (sequences xs)
```

where `sequences` splits the input into sorted slices, and `ascending` and `descending` process increasing and decreasing runs, respectively.

In fact, GHC's mergesort function `Data.List.sort` is smooth bottom-up non-tail-recursive mergesort. Its slightly modified versions have been formally verified in Isabelle/HOL [Sternagel 2013] and Dafny [Leino and Lucio 2015], which we compare to our formalization in Section 7.

4.3 Extended characterization and correctness proofs

4.3.1 Extended characterization of stable mergesort functions. In this section, we extend the characterization presented in Section 3.2 to support tail-recursive and smooth mergesorts. We first add more operators on T and R to the type of abstract sort functions `asort`, as follows:

$$\mathrm{T}'_{\texttt{asort}} \coloneqq \forall (T\, R : \mathcal{U}), \underbrace{(T \to T \to \mathrm{bool})}_{\text{relation } \leq} \to \underbrace{(R \to R \to R)}_{\text{merge by } \leq} \to \underbrace{(R \to R \to R)}_{\text{merge by } \geq} \to \underbrace{(T \to R)}_{\text{singleton}} \to \underbrace{R}_{\text{empty}} \to \underbrace{\mathrm{list}\, T}_{\text{input}} \to \underbrace{R}_{\text{output}}.$$

The first argument of type $T \to T \to \mathrm{bool}$ is there to give `asort` direct access to the relation $\leq$ without going through merge, which we will exploit to support smooth mergesorts. Since tail-recursive mergesorts merge sorted sequences both by $\leq$ and $\geq$, the second and third arguments of type $R \to R \to R$ now represent merge by $\leq$ and $\geq$, respectively. However, R still represents the type of lists sorted w.r.t. $\leq$. In order to merge them with $\geq$, we introduce the following operator $\mathbin{\rotatebox[origin=c]{180}{$\mathbb{M}$}}$:

$$xs \mathbin{\mathbb{W}}_{\leq} ys \coloneqq \texttt{rev}\,(\texttt{rev}\, ys \mathbin{\mathbb{M}}_{\geq} \texttt{rev}\, xs).$$

Therefore, we replace Equations (1) and (2) with the following equations, respectively:

$$\forall (T : \mathcal{U})\,(\leq\, : T \to T \to \mathrm{bool})\,(xs : \mathrm{list}\, T), \texttt{asort}\,(\leq)\,(\mathbb{M}_{\leq})\,(\mathbb{W}_{\leq})\,[\cdot]\,[]\, xs = \texttt{sort}_{\leq}\, xs, \tag{6}$$

$$\forall (T : \mathcal{U})\,(\leq\, : T \to T \to \mathrm{bool})\,(xs : \mathrm{list}\, T), \texttt{asort}\,(\leq)\,(+\!\!+)\,(+\!\!+)\,[\cdot]\,[]\, xs = xs. \tag{7}$$

We define the *extended characteristic property* of stable mergesort functions as the existence of an abstract mergesort function `asort` that is parametric (Section 2.2), *i.e.*, $(\texttt{asort}, \texttt{asort}) \in [\![\, \mathrm{T}'_{\texttt{asort}} \,]\!]$, and satisfies Equations (6) and (7).

4.3.2 Tail-recursive mergesort†*.* The abstract mergesort function for the tail-recursive mergesort function (Section 4.1) can be obtained just by abstracting out the tail-recursive merge function with `(<=)` and `(>=)` to the two abstract merge functions as follows:

```
let asort (<=) merge merge' singleton empty xs =
  let rec sort_rec xs b n =
    match n, xs with
    | 1, x :: xs' -> singleton x, xs'
    | _, _ ->
      let n1 = n / 2 in
      let s1, xs' = sort_rec xs (not b) n1 in
      let s2, xs'' = sort_rec xs' (not b) (n - n1) in
      (if b then merge' s1 s2 else merge s1 s2), xs''
  in
  if xs = [] then empty else fst (sort_rec xs true (length xs))
```

By instantiating the above `asort` as in Equation (6), we replace `revmerge (<=) s1 s2 []` and `revmerge (>=) s2 s1 []` in `sort` in Section 4.1 with `merge (<=) s1 s2` and `rev (merge (>=) (rev s2) (rev s1))`, respectively. While the sorted lists that appear in execution of `sort` in

Section 4.1 are a mix of increasing and decreasing lists, this replacement turns all of them in increasing order. Therefore, the proof of Equation (6) cannot be done just by definition, and involves some equational reasoning about merge and reversal of lists (see Section 6).

4.3.3 Smooth mergesort[†]*.* The major obstacle in defining the abstract mergesort function for the smooth mergesort (Section 4.2) is that it uses the cons (`::`) which does not directly correspond to any of the four abstract operators `merge`, `merge'`, `singleton`, and `empty` on sorted lists. For example, the recursive function `descending` uses `a :: accu` knowing that `a` is strictly smaller than the head of `accu` and `accu` is strictly increasing, and thus, ensures `a :: accu` is strictly increasing. Therefore, we can simulate this behavior with `merge accu (singleton a)`. Similarly, we can simulate the behavior of `a :: accu` in `ascending`, where `a` is greater than or equal to the head of `accu` and `accu` is weakly decreasing, with `merge' accu (singleton a)`. Again, the proof of Equation (6) cannot be done just by definition, and involves these arguments and reasoning about reversal of lists (see Section 6).

4.3.4 Correctness proofs. In this section, we adapt the correctness proofs presented in Section 3.4 to the extended characteristic property. We adapt the induction principle over traces (Lemma 3.5) as follows.

Lemma 4.1 (An induction principle over traces of `sort`). *Suppose $\leq$ and $\sim$ are binary relations on T and* list T*, respectively, and xs is a list of type* list T*. Then, $xs \sim \mathtt{sort}_{\leq}\ xs$ holds whenever the following four induction cases hold:*

- *for any lists xs, xs', ys, and ys' of type* list T*, $(xs +\!\!+\ ys) \sim (xs' \wedge\!\!\!\wedge_{\leq}\ ys')$ holds whenever $xs \sim xs'$ and $ys \sim ys'$ hold,*
- *for any lists xs, xs', ys, and ys' of type* list T*, $(xs +\!\!+\ ys) \sim \mathtt{rev}\,(\mathtt{rev}\ ys' \wedge\!\!\!\wedge_{\geq} \mathtt{rev}\ xs')$ holds whenever $xs \sim xs'$ and $ys \sim ys'$ hold,*
- *for any x of type T, $[x] \sim [x]$ holds, and*
- $[] \sim []$ *holds.*

The only difference between the old and new induction principles is the addition of the second induction case. Therefore, the proofs of Lemma 3.7 and Theorem 3.11 can be easily adapted to the new induction principle by adding the corresponding case, as follows. Again, we refer the readers to Lemma B.3 for the complete proof of Lemma 3.10.

Extension to the proof of Lemma 3.7. We prove $\mathtt{sort}_{\leq}\ xs =_{\mathtt{perm}} xs$ by induction on $\mathtt{sort}_{\leq}\ xs$ (Lemma 4.1). Suffice it to show that $xs' =_{\mathtt{perm}} xs$ and $ys' =_{\mathtt{perm}} ys$ imply:

$$
\begin{aligned}
\mathtt{rev}\,(\mathtt{rev}\ ys' \wedge\!\!\!\wedge_{\geq} \mathtt{rev}\ xs') &=_{\mathtt{perm}} \mathtt{rev}\ ys' \wedge\!\!\!\wedge_{\geq} \mathtt{rev}\ xs' && \text{(Lemma A.56)}\\
&=_{\mathtt{perm}} \mathtt{rev}\ ys' +\!\!+\ \mathtt{rev}\ xs' && \text{(Lemma A.72)}\\
&= \mathtt{rev}\,(xs' +\!\!+\ ys') && \text{(Lemma A.20)}\\
&=_{\mathtt{perm}} xs' +\!\!+\ ys' && \text{(Lemma A.56)}\\
&=_{\mathtt{perm}} xs +\!\!+\ ys. && \text{(Lemma A.54)}
\end{aligned}
$$

The other induction cases are done in the first proof of Lemma 3.7. □

Extension to the proof of Theorem 3.11. We prove a generalized proposition:

$$(\mathtt{sort}_{\leq_1}\ xs \subseteq xs) \wedge (xs \text{ is pairwise sorted w.r.t. } \leq_2 \implies \mathtt{sort}_{\leq_1}\ xs \text{ is sorted w.r.t. } \leq_{\mathrm{lex}})$$

by induction on $\mathtt{sort}_{\leq_1}\ xs$ (Lemma 4.1), where $x \leq_{\mathrm{lex}} y \coloneqq x \leq_1 y \wedge (y \not\leq_1 x \vee x \leq_2 y)$.

For the first component of the conjunction, suffice it to show

$$(xs' \subseteq xs) \wedge (ys' \subseteq ys) \implies (\mathtt{rev}\,(\mathtt{rev}\ ys' \wedge\!\!\!\wedge_{\geq_1} \mathtt{rev}\ xs') \subseteq xs +\!\!+\ ys)$$

which is obvious since rev (rev $ys' \mathbin{⩕}_{\geq_1}$ rev xs') is a permutation of $xs' +\!\!+\ ys'$.

For the second component of the conjunction, suffice it to show that rev (rev $ys' \mathbin{⩕}_{\geq_1}$ rev xs') is sorted w.r.t. $\leq_{\text{lex}}$, or equivalently, its reversal rev $ys' \mathbin{⩕}_{\geq_1}$ rev xs' is sorted w.r.t. the converse of $\leq_{\text{lex}}$ (Lemma A.61):

$$x \geq_{\text{lex}} y \coloneqq x \geq_1 y \wedge (y \not\geq_1 x \vee x \geq_2 y),$$

whenever:

(i) xs' (resp. ys') is a subset of xs (resp. ys),
(ii) xs' (resp. ys') is sorted w.r.t. $\leq_{\text{lex}}$ if xs (resp. ys) is pairwise sorted w.r.t. $\leq_2$, and
(iii) $xs +\!\!+\ ys$ is pairwise sorted w.r.t. $\leq_2$.

Among these, (iii) is equivalent to the following conjunction (Lemma A.58):

(iv) both xs and ys are pairwise sorted w.r.t. $\leq_2$, and
(v) $x \leq_2 y$ holds for any $x \in xs$ and $y \in ys$.

Hypotheses (ii) and (iv) imply that both xs' and ys' are sorted w.r.t. $\leq_{\text{lex}}$, or equivalently, rev xs' and rev ys' are sorted w.r.t. $\geq_{\text{lex}}$ (Lemma A.61). Hypotheses (i) and (v) imply that $y \geq_2 x$ holds for any $y \in$ rev ys' and $x \in$ rev xs' (Lemmas A.30 and A.33). These two facts suffice to show that rev $ys' \mathbin{⩕}_{\geq_1}$ rev xs' is sorted w.r.t. $\geq_{\text{lex}}$ (Lemma A.74).

The other induction cases are done in the first proof of Theorem 3.11. □

The naturality of `sort` (Lemma 3.14), which remains the same, can be deduced from the parametricity of `asort` as well. Therefore, the rest of the correctness proofs, which have been verified in Rocq, remains the same.

5 Formalization in Rocq

In this section, we discuss two technical aspects of our formalization of mergesort functions and their correctness proofs in Rocq. We first review a technique [Gonthier 2009][3] to make bottom-up mergesorts structurally recursive, so that their termination becomes trivial for Rocq (Section 5.1). Furthermore, this technique makes the balanced binary tree construction of bottom-up mergesort tail-recursive (Section 5.1.2), and thus allows us to implement bottom-up tail-recursive mergesort (Section 5.1.3). We second discuss the design and organization of the library, particularly, the interface for mergesort functions which allows us to state our correctness lemmas polymorphically for any stable mergesort function, and how to populate this interface with concrete mergesort functions (Section 5.2).

While Section 5.1 continues to use the OCaml syntax to present new mergesort functions, Appendix D presents our actual Rocq implementations of structurally-recursive mergesort functions, including some optimized implementations such as smooth variants (Section 4.2).

5.1 Structurally-recursive bottom-up mergesorts†

5.1.1 The syntactic guard condition. A fixpoint function f in Rocq [The Rocq Development Team 2025d] has the form of (`fix` f_{rec} $(\vec{x} : \vec{A})$ {`struct` x_k} : $B \coloneqq M$) where f_{rec} is the local name of the fixpoint function bound in M, $(\vec{x} : \vec{A})$ is the list of arguments, x_k is the kth element of $\vec{x}$ and the recursive (decreasing) argument, and M is the function body of type B. This fixpoint function f has type $(\forall\ (\vec{x} : \vec{A}),\ B)$. To ensure the termination of f, all recursive calls of f_{rec} in M must be *guarded by destructors* [Giménez 1994] in Rocq. That is to say, M must do recursive calls to f_{rec}

[3]The mergesort functions in the Mathematical Components (MathComp) library [Mahboubi and Tassi 2022] and the Rocq standard library [The Rocq Development Team 2025f], first appeared in commits respectively by Théry [2008] and Herbelin [2009], uses the same technique. Since its first appearance [Théry 2008] was a merge commit, we could not clearly identify who devised this technique. Nevertheless, we call it Gonthier's technique for convenience.

```
let rec merge (<=) xs ys (* struct xs *) =
  match xs with
  | [] -> ys
  | x :: xs' ->
    let rec merge' ys (* struct ys *) =
      match ys with
      | [] -> xs
      | y :: ys' -> if x <= y then x :: merge (<=) xs' ys else y :: merge' ys'
    in
    merge' ys

let sort (<=) =
  let rec push xs stack (* struct stack *) =
    match stack with
    | [] :: stack | ([] as stack) -> xs :: stack
    | ys :: stack -> [] :: push (merge (<=) ys xs) stack
  in
  let rec pop xs stack (* struct stack *) =
    match stack with
    | [] -> xs
    | ys :: stack -> pop (merge (<=) ys xs) stack
  in
  let rec sort_rec stack xs (* struct xs *) =
    match xs with
    | [] -> pop [] stack
    | x :: xs -> sort_rec (push [x] stack) xs
  in
  sort_rec []
```

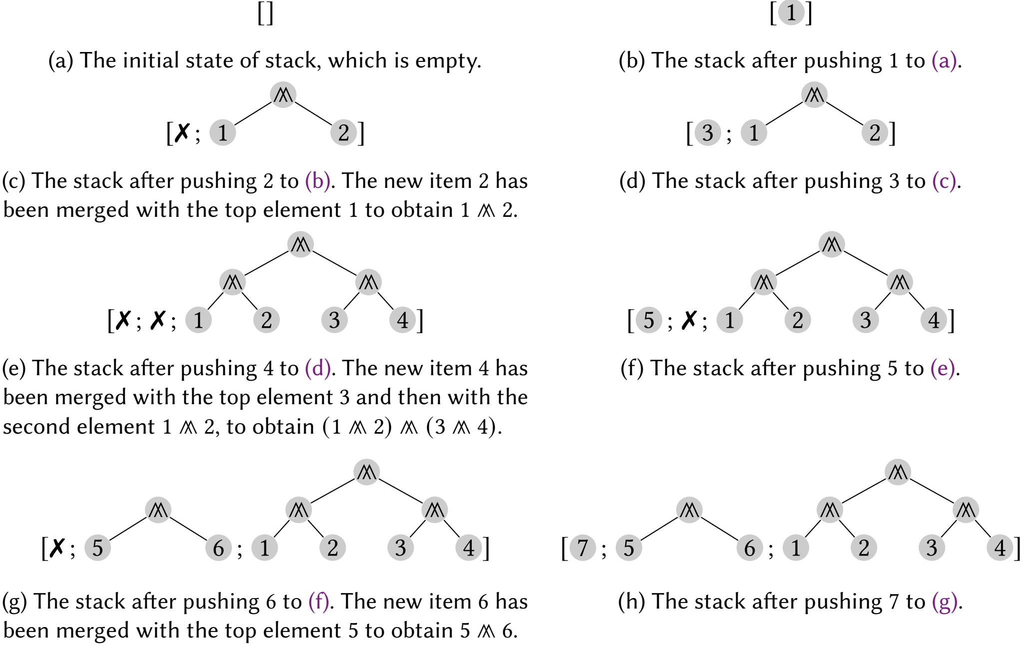
Fig. 5. Structurally-recursive non-tail-recursive merge and mergesort in OCaml.

only on strict subterms of x_k. In practice, the annotation of decreasing argument {`struct` x_k} may be left implicit, and Rocq can infer it automatically. Hereafter in this section, we explain how to make bottom-up mergesorts structurally recursive, but in the OCaml syntax with annotations of decreasing argument as comments, *e.g.*, `(* struct xs *)`.

As the first example of termination checking, we use the non-tail-recursive merge function. Its definition in Figure 1a already does not satisfy the syntactic guard condition, because the first and second recursive calls of `merge` are decreasing only on the first and the second lists, respectively, and the syntactic guard condition does not take a termination argument involving multiple parameters (*e.g.*, lexicographic termination) into account.

One way to work around this restriction is to use nested fixpoint as in `merge` in Figure 5. The outer recursive function `merge` performs recursion on `xs`, and the first case where `x <= y` holds calls it with `xs'`, which is obtained by destructing `xs` and hence a strict subterm of `xs`. Similarly, the inner recursive function `merge'` performs recursion on `ys`, and the second case where `x <= y` does not hold calls it with `ys'`, which is a strict subterm of `ys`. This way, the termination checker can confirm that the `merge` function terminates for any input.

5.1.2 Non-tail-recursive mergesort. In Figure 2b, `merge_all` does not satisfy the syntactic guard condition since `merge_pairs xs` is not a strict subterm of `xs`. To work around this issue, we



(a) The initial state of stack, which is empty.

(b) The stack after pushing 1 to (a).

(c) The stack after pushing 2 to (b). The new item 2 has been merged with the top element 1 to obtain 1 ⩕ 2.

(d) The stack after pushing 3 to (c).

(e) The stack after pushing 4 to (d). The new item 4 has been merged with the top element 3 and then with the second element 1 ⩕ 2, to obtain (1 ⩕ 2) ⩕ (3 ⩕ 4).

(f) The stack after pushing 5 to (e).

(g) The stack after pushing 6 to (f). The new item 6 has been merged with the top element 5 to obtain 5 ⩕ 6.

(h) The stack after pushing 7 to (g).

Fig. 6. An example of state transitions of the explicit stack of pending mergings in the sorting process explained in Section 5.1.2. The cross mark ✗ denotes an empty element in the stack. In general, pushing a new item xs to the stack is done by (1) replacing the longest consecutive subsequence of nonempty elements $xs_0, \dots, xs_n$ from the top of the stack, where xs_0 is the top element, with empty elements, and then (2) replacing the next empty element xs_{n+1} in the stack with xs_n ⩕ $(\cdots$ ⩕ $(xs_0$ ⩕ $xs) \dots)$.

manage pending mergings using an explicit stack [Gonthier 2009]. We represent the stack of pending mergings as a list of sorted lists whose head (0th) and tail elements respectively correspond to the top and bottom elements of the stack. The sorting process proceeds by repetitively pushing items to be sorted to the stack, as in `push` in Figure 5. An item xs pushed to the top of the stack is called a *merging of level 0*, and a result of merging two mergings of level n is called a *merging of level* $n + 1$. The nth element of the stack must be a merging of level n if any; otherwise, the empty list. Therefore, if the top of the stack (or the stack itself) is empty, pushing an item xs is done by replacing the top element with xs; otherwise—if the stack S has the form of $xs_0 :: S'$ where xs_0 is nonempty—, it is done by replacing the top element xs_0 with the empty list and pushing xs_0 ⩕$_{\leq}$ xs[4] to S'. In terms of trace, this procedure can be seen as a technique to construct balanced binary trees, and the nth element of the stack corresponds to a perfect binary tree of height n, as in Figure 6. The sorting process can be completed by pushing all the items in the input to the stack and then folding the stack by `merge`, as in `sort_rec` and `pop` in Figure 5, respectively.

Gonthier's mergesort (Figure 5), particularly its construction of balanced binary trees, can be seen as a variation of smooth bottom-up non-tail-recursive mergesort presented by O'Keefe [1982] and reviewed by Paulson [1996, Subsection "Bottom-up merge sort" in Section 3.21]. O'Keefe's mergesort does not keep empty lists in the stack, but recovers the information encoded by empty lists by counting the number of pushes performed on the stack, because one may know whether

[4]Note that the first item of the input list will be pushed first and placed as a part of the bottom element of the stack. The ordering of the arguments xs_0 and xs here matters for maintaining the stability of the algorithm.

the stack contains a merging of level n or not by testing the nth binary digit of the counter. To put it another way, the stack in our implementation can be seen as a binary natural number in the least significant bit first form representing the counter, by replacing the empty and non-empty elements with 0 and 1, respectively. Regardless of which approach we choose, bottom-up mergesort using the explicit stack of pending mergings does not use up stack space except for `merge`, because the depth of recursive calls of `push` is logarithmic in the length of the input, and `pop` and `sort_rec` are tail recursive. In Section 5.1.3, we implement bottom-up tail-recursive mergesort by relying on this observation.

Structurally-recursive non-tail-recursive mergesort can be made smooth (Section 4.2) by pushing sorted slices to the stack instead of singleton lists. See Appendix D.1 for its Rocq implementation.

Since the emptiness test for sorted lists is not supplied to the abstract mergesort function as defined in Section 4.3.1, we change the type of the stack in the abstract mergesort from `list R` to `list (option R)` and use the `None` constructor of the `option` type to represent empty lists in the stack. Therefore, the proof of Equation (6) for structurally-recursive mergesorts cannot be done just by definition (Section 6).

5.1.3 Tail-recursive mergesort. The nested fixpoint technique in defining a recursive function that have more than one recursive argument (Section 5.1.1) can be easily adapted to the tail-recursive merge function (Figure 1b), as in `revmerge` in Figure 7. As noted in Section 4.1, this function produces its output in reverse order.

If we take reversals done by `revmerge` into account in our construction scheme of balanced binary trees (Section 5.1.2), the `push` function can be adapted as in Figure 7. In this new `push` function, pending mergings of ascending and descending orders should appear alternately in the stack, and its top element and xs should always be ascending ones. Its one recursion step processes two elements of the stack to maintain this recursion invariant, and pushing an item xs to the stack proceeds as follows.

- If the top of the stack (or the stack itself) is empty, it replaces the top element with xs.
- If the top of the stack ys is nonempty and the next element zs is empty (or the length of the stack is 1), it replaces zs with $\texttt{rev}\,(ys \curlywedge_{\leq} xs)$ and ys with an empty element.
- Otherwise—if the stack has the form of $ys :: zs :: S'$ where both ys and zs are nonempty—, it has to push the result of merging xs, ys, and zs to S', where xs and ys are ascending but zs is descending. Therefore, it pushes $\texttt{rev}\,(zs \curlywedge_{\geq} \texttt{rev}\,(ys \curlywedge_{\leq} xs))$ to S' and replaces xs and ys with empty elements.

As in Section 5.1.2, the sorting process can be completed by pushing all the items in the input to the stack and then folding the stack by `revmerge` (`sort_rec` and `pop` in Figure 7, respectively). In a recursive call of `pop`, the head of the stack can either be ascending or descending in contrast to `push`. Its additional argument `mode` of type `bool` is `true` iff `xs` and the head of the stack are descending ones. If an element of the stack to be processed is empty, `pop` reverses `xs` (in the 4th case), because `xs` and the head of the stack have to be in the same order. In order to avoid reversing `xs` twice when the stack has a pair of two adjacent empty elements, the 3rd case skips such empty elements just for performance reasons.

Note again that the recursive functions presented in this section are tail recursive except for `push`, whose depth of recursive calls is logarithmic in the length of the input. Therefore, the `sort` function above does not use up stack space. Furthermore, it can be made smooth in the same way as the non-tail-recursive counterpart (Section 5.1.2). See Appendix D.2 for its Rocq implementation.

```
let rec revmerge (<=) xs ys accu (* struct xs *) =
  match xs with
  | [] -> rev_append ys accu
  | x :: xs' ->
    let rec revmerge' ys accu (* struct ys *) =
      match ys with
      | [] -> xs
      | y :: ys' ->
        if x <= y then
          revmerge (<=) xs' ys (x :: accu)
        else
          revmerge' ys' (y :: accu)
    in
    revmerge' ys accu

let sort (<=) =
  let (>=) = (fun x y -> y <= x) in
  let rec push xs stack (* struct stack *) =
    match stack with
    | [] :: stack | ([] as stack) -> xs :: stack
    | ys :: [] :: stack | ys :: ([] as stack) ->
      [] :: revmerge (<=) ys xs [] :: stack
    | ys :: zs :: stack ->
      [] :: [] :: push (revmerge (>=) (revmerge (<=) ys xs []) zs []) stack
  in
  let rec pop mode xs stack (* struct stack *) =
    match stack, mode with
    | [], true -> rev xs
    | [], false -> xs
    | [] :: [] :: stack, _ -> pop mode xs stack
    | [] :: stack, _ -> pop (not mode) (rev xs) stack
    | ys :: stack, true -> pop false (revmerge (>=) xs ys []) stack
    | ys :: stack, false -> pop true (revmerge (<=) ys xs []) stack
  in
  let rec sort_rec stack xs (* struct xs *) =
    match xs with
    | [] -> pop false [] stack
    | x :: xs -> sort_rec (push [x] stack) xs
  in
  sort_rec []
```

Fig. 7. Structurally-recursive tail-recursive merge and mergesort in OCaml.

5.2 Interface for stable mergesorts

In this section, we present the interface (Section 5.2.1) for mergesort functions bundling the extended characteristic property (Sections 3.2 and 4.3.1), so that we can state our correctness lemmas polymorphically for any stable mergesort function (Section 5.2.3), and explain how to populate (Section 5.2.2) this interface with concrete mergesort functions, in Rocq.

5.2.1 The interface. We first define the following constant `asort_ty` for the type of abstract sort functions.

```
Definition asort_ty :=
  ∀ (T R : Type),
    (T → T → bool) → (R → R → R) → (R → R → R) → (T → R) → R →
    list T → R.
```

To automatically generate parametricity statements and proofs, we use ParamCoq [Keller et al. 2014] that implements a version of the parametricity translation $[\![\cdot]\!]$ [Bernardy et al. 2012; Bernardy and Lasson 2011; Keller and Lasson 2012] for Rocq, extended to terms of the calculus. The abstraction theorem (Assumption 2.2) for this parametricity translation can be restated as follows.

Theorem 5.1 (Abstraction theorem). *If* $\vdash t : A$*, then* $\vdash [\![t]\!] : [\![A]\!]\, t\, t$.

Now, $[\![\cdot]\!]$ translates a Rocq term $t : A$ to a term of type $[\![A]\!]\, t\, t$, which is a Rocq internalization of the membership $(t, t) \in [\![A]\!]$ from Section 2.2, where the relation $[\![A]\!]$ is internalized as a function $A \to A \to \mathcal{U}$. Although we assumed the abstraction theorem in Sections 3 and 4 (Assumption 2.2), and Theorem 5.1 holds only for a fragment of the underlying calculus of Rocq, we stress that our functional correctness proofs are axiom free thanks to ParamCoq, which produces parametricity proofs for all the `asort` functions in our formalization.

The following `Parametricity` command from ParamCoq declares the parametricity relation $[\![$ `asort_ty` $]\!]$ as a new constant `asort_ty_R`.

```
Parametricity asort_ty.
```

We second define the interface for stable mergesort functions, that is, a dependent record type bundling a mergesort function and the characteristic property on it.

```
Structure stableSort := StableSort {
  apply : ∀ T : Type, (T → T → bool) → list T → list T;
  asort : asort_ty;
  asort_R : asort_ty_R asort asort;
  asort_mergeE : ∀ (T : Type) (leT : T → T → bool) (xs : list T),
    let geT x y := leT y x in
    let mergerev xs ys := rev (merge geT (rev ys) (rev xs)) in
    asort leT (merge leT) mergerev (fun x ⇒ [:: x]) [::] xs = apply leT xs;
  asort_catE : ∀ (T : Type) (leT : T → T → bool) (xs : list T),
    asort leT cat cat (fun x ⇒ [:: x]) [::] xs = xs;
}.
```

where `apply` is the mergesort function in question, `asort` is the abstract version of `apply`, `asort_R` is the parametricity of `asort`, `asort_mergeE` and `asort_catE` are respectively the two equational properties (6) and (7) on `asort`, `stableSort` is the record type bundling these five fields, and `StableSort` is the constructor of `stableSort`.

5.2.2 Populating the interface. Suppose `sort1` is a mergesort function and `asort1` is its abstract version. We state Equations (6) and (7) as follows:

```
Fact asort1_mergeE (T : Type) (leT : T → T → bool) (xs : list T) :
  let geT x y := leT y x in
  let mergerev xs ys := rev (merge geT (rev ys) (rev xs)) in
  asort1 leT (merge leT) mergerev (fun x ⇒ [:: x]) [::] xs = sort1 leT xs.

Fact asort1_catE (T : Type) (leT : T → T → bool) (xs : list T) :
  asort1 leT cat cat (fun x ⇒ [:: x]) [::] xs = xs.
```

and prove them by functional induction on `asort1` and equational reasoning, as we saw in Section 3.3. While Equation (6) can be proved by computation (by the `reflexivity` tactic) in the simplest cases, it is not the case for any structurally-recursive mergesort (Section 5.1). We discuss this limitation further in Section 6.

Proving that `asort1` is parametric can be done just by the parametricity translation (Section 5.2.1):

```
Parametricity asort1. (* generates a parametricity proof asort1_R. *)
```

Provided we have all these ingredients, we can construct an instance of the `stableSort` record type as follows.

```
Definition sort1_stable ≔
  StableSort sort1 asort1 asort1_R asort1_mergeE asort1_catE.
```

5.2.3 Correctness lemmas. The correctness lemmas of mergesort (Section 3.4) can be stated polymorphically for any `stableSort` instance. For example, the commutation of `filter` and mergesort functions (Theorem 3.16) can be stated as follows:

```
Lemma filter_sort (sort : stableSort) (T : Type) (leT : T → T → bool) :
  total leT → transitive leT →
  ∀ (p : T → bool) (xs : list T),
    filter p (apply sort T leT xs) = apply sort T leT (filter p xs).
```

where `apply` can be omitted by declaring it as an implicit coercion [The Rocq Development Team 2025e], so that a `stableSort` instance itself can be seen as a sort function in the user-facing syntax:

```
Coercion apply : stableSort ↣ Funclass.
```

Also, we provide for several lemmas a version where hypotheses are localized by a predicate. For example, the relation `leT` in the above lemma `filter_sort` (Theorem 3.16) only needs to be total and transitive on the domain delimited by a predicate `P` provided all elements of `s` are in `P`, as follows:

Corollary 5.2. *The following equation holds*

$$\mathtt{filter}_p\,(\mathtt{sort}_{\leq}\,xs) = \mathtt{sort}_{\leq}\,(\mathtt{filter}_p\,xs)$$

whenever $\leq$ *is total and transitive on a predicate* $P \subseteq T$*, meaning that* $x \leq y \lor y \leq x$ *and* $x \leq y \land y \leq z \Rightarrow x \leq z$ *hold for any* $x, y, z \in P$*, and all elements of* xs *satisfy* P*.*

This corollary corresponds to the following lemma in Rocq:

```
Lemma filter_sort_in
  (sort : stableSort) (T : Type) (P : T → bool) (leT : T → T → bool) :
  {in P &, total leT} → {in P & &, transitive leT} →
  ∀ (p : T → bool) (xs : list T), all P xs →
    filter p (sort T leT xs) = sort T leT (filter p xs).
```

where `{in P &, Q}` *reduces to* `∀ x : T, P x → ∀y : T, P y → R x y` *given that* `Q` *reduces to* `∀ x y : T, R x y` *and* `P` *has type* `T → bool`*,* `{in P & &, Q}` *is its ternary version, and* `all P s` *means that* `P` *holds for any element of* `xs`*.*

Proof. In dependent type theory, one may define a subtype `sig P` collecting inhabitants of `T` satisfying `P`. Using the canonical `val : sig P → T` function, the relation `leT` can be turned into a relation `leP x y ≔ leT (val x) (val y)` on `sig P`, which is a total preorder thanks to the assumptions on `leT` (Lemma A.76). Since `P` holds for any element of `xs`, we can replace `xs` everywhere with `map val xs'`, where `xs'` is a list of type `list (sig P)` (Lemma A.77). Thanks

to the naturality of `sort` (Lemma 3.14) and `filter` (Lemma A.37), and the congruence rule with respect to `map`, it remains to proves

```
filter p' (sort (sig P) leP xs') = sort (sig P) leP (filter p' xs')
```

where `p' x ≔ p (val x)`. We can then conclude by applying `filter_sort` (Theorem 3.16). □

Appendix B provides the list of all lemmas about stable sort functions solely derived from the characterization, their formal statements in Rocq that use the `stableSort` structure, and their informal proofs.

6 Limitations

As we mentioned in Sections 4.3.2, 4.3.3, 5.1.2 and 5.2.2, the present proof technique has the limitation that the proof of Equation (6) cannot be done by definition (by computation in Rocq) for elaborate variations of mergesort. This is because these mergesorts inspect sorted lists by operations not supplied to the abstract mergesort function, the abstract mergesort function has to simulate it in a non-trivial way, and thus, the proof of Equation (6) involves the simulation arguments. The examples of such operations appeared in the paper are tail-recursive merge (Sections 4.1 and 4.3.2), cons in smooth mergesorts (Sections 4.2 and 4.3.3), and emptiness test in structurally-recursive mergesorts (Section 5.1). Since we are primarily interested in structurally-recursive mergesorts in our formalization, the proof of Equation (6) had to be done by hand for most variations. Nevertheless, we argue that the proofs remain relatively short compared to related work (Section 7).

We might be able to relax the limitation by supplying more operators to the abstract mergesort function, or by using a type system that has more conversion rules. For example, we can consider splitting the abstract type R representing sorted lists into two abstract types R and R' respectively representing sorted lists in ascending and descending orders, and supplying tail-recursive merge functions of types $R \to R \to R'$ and $R' \to R' \to R$, to relax the limitation for tail-recursive mergesorts. However, this solution would come at a cost that the stack of pending mergings in structurally-recursive tail-recursive mergesort (Section 5.1.3) cannot be represented as a homogeneous list, since abstract sorted lists of types R and R' should appear alternately in the stack. Similarly, we can consider supplying an emptiness test function to relax the limitation for structurally-recursive mergesorts. Any solution of supplying more operators would come at a cost that the abstract correctness proofs have to be adapted as we extend the characterization with the new operators. Exploring such possibilities is left as future work.

Since GHC 9.12.1 [The GHC Team 2024b], `Data.List.sort` has been changed to use 3-way and 4-way merge functions `merge3 as bs cs` and `merge4 as bs cs ds` [cbrt-x 2023], which are documented as manually-fused versions of `merge (merge as bs) cs` and `merge (merge as bs) (merge cs ds)`, respectively. However, `merge4 [] as bs cs` and `merge4 as [] bs cs` reduce to `merge3 as bs cs`, and thus, it changes the way it nests the 2-way merge. Therefore, the 4-way merge actually cannot be expressed as a simple combinations of the 2-way merge; thus, applying our proof technique to this mergesort would require to either fix the 4-way merge function by adding a right-associative version of 3-way merge or extend the characteristic property with the 4-way merge.

7 Related work

Section 3.1 implies that mergesort conceptually consists of a construction of a balanced binary tree and folding of this binary tree with merge, which is an instance of the fact that many sorting algorithms can be explained as folds of unfolds, or, dually, as unfolds of folds [Hinze et al. 2012, 2013]. The traces we informally show in Figure 3 are reminiscent of the tree datatype of Hinze

et al. [2013, Section 6]. Our `asort` functions for non-tail-recursive mergesorts (Section 3.2) and their `makeTree` functions have the same type, except that binary trees are Church-encoded in the former (Remark 3.6), and expose the underlying binary tree construction of mergesort.

Using parametricity to test or verify sort functions is not a new idea. For example, Knuth's 0-1-Principle [Knuth 1973, Theorem Z of Section 5.3.4] can be deduced from parametricity [Day et al. 1999]. While such an idea has been advanced towards both program testing [Hou (Favonia) and Wang 2022; Voigtländer 2008] and formal verification [Bove and Coquand 2004], the present work is the first application of parametricity to prove the stability of sorting functions to the best of our knowledge.

The only prior work on formally proving the stability of mergesort is by Leino and Lucio [2015]; Leroy [[n. d.]]; Sternagel [2013]. Among these, Leroy [[n. d.]] proved a non-smooth bottom-up non-tail-recursive mergesort (similar to Figure 2b) correct in Rocq (in 360 LoC), Sternagel [2013] proved a slightly modified version of GHC's mergesort correct in Isabelle/HOL (in 177 LoC), and Leino and Lucio [2015] ported the latter result to Dafny (in 633 LoC). All of them proved the permutation property (Lemma 3.7), sortedness (Lemma B.17), and stability (Corollary 3.17). In addition, Sternagel [2013] proved extensional equality of the mergesort and an insertion sort (Lemma B.20). On the other hand, our formalization consists of:

- 121 LoC for implementations and functional correctness proofs of 4 merge functions (hence around 31 LoC per merge function on average),
- 16 LoC for implementation and characteristic property proof of the insertion sort (Definition 3.1 and Lemma 3.2), and
- 533 LoC for implementations and characteristic property proofs of 8 mergesort functions listed in Appendix D (hence around 67 LoC per sort function on average), and
- 472 LoC for generic functional correctness proofs and the infrastructure for 43 lemmas listed in Appendix B, but only 198 LoC to cover our main stability results (Theorems 3.11 and 3.16) and the results in related work (Lemmas B.17, B.20 and 3.7 and Corollary 3.17).

Hence, for:

- a single mergesort function, we would have on average 312 LoC ($= 31 + 16 + 67 + 198$) to get its implementation, permutation, sortedness, stability, extensional equality to the insertion sort, including all the infrastructure, and
- all eight mergesort functions, it amounts to an average of 109 LoC per sort function for theorems up to Corollary 3.17, and 143 LoC for all 43 lemmas of our theory of sort functions.

Therefore, although comparing LoC across different systems, standard libraries, coding styles, *etc.* only allows for very crude comparison, we conclude that the only prior work similarly concise to our proof is the one by Sternagel [2013]. Whether our proof is shorter depends on how we compare them: if we extract from our proof only one implementation it is 135 LoC longer than his, while if we divide the total LoC by the number of mergesort variants, it is 68 LoC shorter. Now, the key ingredient for achieving the conciseness of Sternagel [2013]'s proofs appear to be skillful functional induction schemes and a simple invariant for proving Corollary 3.17 for GHC's mergesort [Sternagel 2013, Section 4]. However, this approach does not fully scale to tail-recursive mergesorts and the proof of Theorem 3.11, which is slightly more general than Corollary 3.17 (Section 3.4.3 and Appendix C), and thus, does not allow us to easily prove a large part of our functional correctness results. Our smooth non-tail-recursive mergesort (an extension of Section 5.1.2, shown in Appendix D.1) does not follow GHC's mergesort as close as in Leino and Lucio [2015]; Sternagel [2013], since `mergeAll` function (`merge_all` in Figure 2b) as is cannot be defined in Rocq due to the syntactic guard condition. Nevertheless, this is a limitation of Rocq rather than a limitation of our approach.

Nipkow et al. [2025] presented several verified sort algorithms, *e.g.*, non-smooth top-down and bottom-up non-tail-recursive mergesorts, and smooth mergesort presented by Sternagel [2013]. However, a large part of their functional correctness proofs are done again for each implementation, and thus, their proofs are not modular as ours. In contrast to the present work, Nipkow et al. [2025] verifies asymptotic complexity bounds of mergesorts using automatically derived *running time functions* modelling a call-by-value semantics. However, this approach lacks a formal guarantee that the running time function represents the actual running time and would not allow us to formally prove that non-tail-recursive mergesort in call-by-need evaluation is an optimal incremental sorting [Paredes and Navarro 2006].

De Gouw et al. [2019] verified a smooth mergesort for arrays called *TimSort* taken from the OpenJDK core library using Java verification tool KeY. During their attempt of verification, they discovered and fixed a bug that may cause an array index out of bounds error, and proved that no such error occurs in the fixed version. While the bug fix has a significant impact, they had not proven the functional correctness (sortedness and permutation properties) of the fixed version.

As a concluding remark, we stress that our formalization has a few features that none of the related work above achieved: 1) a common interface (characterization) for stable mergesort functions, general induction principle for mergesort replacing functional induction, sharing of functional correctness proofs between several variations of mergesort, 2) formally verified tail-recursive mergesort (Sections 4.1 and 5.1.3), including the smooth one (Appendix D.2), and 3) extensive theory of stable sort functions (Appendix B) which cannot be easily derived from the results in the related work (Section 3.4.3 and Appendix C).

8 Conclusion

We characterized mergesort functions for lists using their abstract versions and parametricity (Sections 3.2 and 4.3.1). By abstracting out the type of sorted lists as a type parameter, we forced the abstract mergesort functions to use the only provided operators (such as the order relation, merge, singleton, and empty) to construct sorted lists, thus we ruled out behaviors incorrect as sorting functions, and the parametricity of the abstract mergesort function ensures such correct behavior. By instantiating the abstract mergesort functions in two ways, we should be able to obtain both input and output of the mergesort function (Equations (6) and (7)). By exploiting the fact that parametricity implies induction principles on Church-encoded datatypes (Remark 3.6) and a case where parametricity implies naturality (Remark 3.15), we deduced an induction principle (Lemmas 3.5 and 4.1) over traces (Figure 3) from Equations (6) and (7), and the naturality of mergesort (Lemma 3.14), respectively. These two properties were sufficient to deduce several correctness results of mergesort, including stability (Sections 3.4 and 4.3.4).

In order to verify a given mergesort function using our technique, one just has to define its abstract version, and then, prove its parametricity, and Equations (6) and (7). Among these properties, parametricity follows from the abstraction theorem [Bernardy et al. 2012; Bernardy and Lasson 2011; Keller and Lasson 2012; Reynolds 1983], which is implemented in Paramcoq [Keller et al. 2014]. The rest of the correctness proofs work generically for any mergesort satisfying our characteristic property. Therefore, the actual work that has to be carried out by hand is to prove Equations (6) and (7) by induction and equational reasoning, which justifies the title of this paper, claiming our functional correctness proofs are almost for free and at a bargain.

Data-Availability Statement

An artifact of this work is available for reproduction on Zenodo [Cohen and Sakaguchi 2025] and includes source code and dependencies. Source code is further available for reuse through the Git repository at https://github.com/pi8027/stablesort.

Acknowledgments

The authors gratefully thank anonymous reviewers of ICFP '22, '24, and '25, Kazuhiro Inaba, and Yannick Zakowski for their comments. The general structure of our correctness proofs of mergesorts presented in Section 3.4, except for the use of parametricity, is largely based on the correctness proofs of a non-smooth bottom-up non-tail-recursive mergesort (`path.sort`) in the MathComp library, to which the authors made significant contributions (in pull requests #328, #358, #601, #650, #680, #727, #1174, and #1186). The authors would like to thank other MathComp developers and contributors who contributed to the discussion: Yves Bertot, Christian Doczkal, Georges Gonthier, Assia Mahboubi, and Anton Trunov.

References

Thorsten Altenkirch, Yorgo Chamoun, Ambrus Kaposi, and Michael Shulman. 2024. Internal Parametricity, without an Interval. *Proc. ACM Program. Lang.* 8, POPL (2024), 2340–2369. doi:10.1145/3632920

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free - Parametricity for dependent types. *J. Funct. Program.* 22, 2 (2012), 107–152. doi:10.1017/S0956796812000056

Jean-Philippe Bernardy and Marc Lasson. 2011. Realizability and Parametricity in Pure Type Systems. In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6604)*. Springer, 108–122. doi:10.1007/978-3-642-19805-2_8

Ana Bove and Thierry Coquand. 2004. Formalising Bitonic Sort in Type Theory. In *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3839)*. Springer, 82–97. doi:10.1007/11617990_6

cbrt-x. 2023. haskell/core-libraries-committee#236: Improve performance of Data.List.sort. https://github.com/haskell/core-libraries-committee/issues/236 Accessed: 2025-02-21.

Adam Chlipala. 2013. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press. http://mitpress.mit.edu/books/certified-programming-dependent-types

Cyril Cohen and Kazuhiko Sakaguchi. 2025. Stable sort algorithms and their stability proofs in Rocq. doi:10.5281/zenodo.15656542

Nancy A. Day, John Launchbury, and Jeff Lewis. 1999. Logical Abstractions in Haskell. In *Proceedings of the 1999 Haskell Workshop*. https://www.haskell.org/haskell-symposium/1999/1999-28.pdf Technical Report UU-CS-1999-28, Utrecht University.

Stijn de Gouw, Frank S. de Boer, Richard Bubel, Reiner Hähnle, Jurriaan Rot, and Dominic Steinhöfel. 2019. Verifying OpenJDK's Sort Method for Generic Collections. *J. Autom. Reason.* 62, 1 (2019), 93–126. doi:10.1007/s10817-017-9426-4

Edsger W. Dijkstra. 1982. Smoothsort, an Alternative for Sorting In Situ. *Sci. Comput. Program.* 1, 3 (1982), 223–233. doi:10.1016/0167-6423(82)90016-8

Eduardo Giménez. 1994. Codifying Guarded Definitions with Recursive Schemes. In *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers (Lecture Notes in Computer Science, Vol. 996)*. Springer, 39–59. doi:10.1007/3-540-60579-7_3

Georges Gonthier. 2009. RE: [Coq-Club] Sorting. https://sympa.inria.fr/sympa/arc/coq-club/2009-04/msg00040.html Accessed: 2021-11-11.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms* (fourth ed.). MIT Press. https://mitpress.mit.edu/9780262046305/

Hugo Herbelin. 2009. Addition of mergesort + cleaning of the Sorting library. https://github.com/coq/coq/commit/f698148f6aee01a207ce5ddd4bebf19da2108bff Accessed: 2025-02-13.

Ralf Hinze, Daniel W. H. James, Thomas Harper, Nicolas Wu, and José Pedro Magalhães. 2012. Sorting with bialgebras and distributive laws. In *Proceedings of the 8th ACM SIGPLAN workshop on Generic programming, WGP@ICFP 2012, Copenhagen, Denmark, September 9-15, 2012*. ACM, 69–80. doi:10.1145/2364394.2364405

Ralf Hinze, José Pedro Magalhães, and Nicolas Wu. 2013. A Duality of Sorts. In *The Beauty of Functional Code - Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday (Lecture Notes in Computer Science, Vol. 8106)*. Springer, 151–167. doi:10.1007/978-3-642-40355-2_11

Kuen-Bang Hou (Favonia) and Zhuyang Wang. 2022. Logarithm and program testing. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–26. doi:10.1145/3498726

Jane Street Group, LLC. 2024. Core: Industrial strength alternative to OCaml's standard library. https://ocaml.org/p/core/v0.17.1/doc/Core/index.html Accessed: 2025-02-21, Version 0.17.1.

Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.* 3, POPL (2019), 2:1–2:24. doi:10.1145/3290315

Chantal Keller and Marc Lasson. 2012. Parametricity in an Impredicative Sort. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France (LIPIcs, Vol. 16)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 381–395. doi:10.4230/LIPIcs.CSL.2012.381

Chantal Keller, Marc Lasson, Abhishek Anand, Pierre Roux, Emilio Jesús Gallego Arias, Cyril Cohen, and Matthieu Sozeau. 2014. Paramcoq: Coq plugin for parametricity. https://github.com/coq-community/paramcoq Accessed: 2024-01-17.

Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.

K. Rustan M. Leino and Paqui Lucio. 2015. An Assertional Proof of the Stability and Correctness of Natural Mergesort. *ACM Trans. Comput. Log.* 17, 1 (2015), 6:1–6:22. doi:10.1145/2814571

Xavier Leroy. [n. d.]. Library Mergesort. https://xavierleroy.org/coq/Mergesort.html Accessed: 2024-02-10.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2024. The OCaml system. https://v2.ocaml.org/releases/4.14/htmlman/ Accessed: 2024-02-10, Version 4.14.

Assia Mahboubi and Enrico Tassi. 2022. *Mathematical Components*. Zenodo. doi:10.5281/zenodo.7118596

Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gomez-Londono, Peter Lammich, Christian Sternagel, Simon Wimmer, and Bohua Zhan. 2025. Functional Data Structures and Algorithms: A Proof Assistant Approach. https://functional-algorithms-verified.org/ Accessed: 2025-06-13.

Richard A. O'Keefe. 1982. *A smooth applicative merge sort*. Department of Artificial Intelligence, University of Edinburgh. This literature does not seem available at the point of writing the present paper. Therefore, when we refer to it, we actually rather refer to the part of Paulson [1996] explaining the ideas of this literatures.

Rodrigo Paredes and Gonzalo Navarro. 2006. Optimal Incremental Sorting. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments, ALENEX 2006, Miami, Florida, USA, January 21, 2006*. SIAM, 171–182. doi:10.1137/1.9781611972863.16

Lawrence C. Paulson. 1996. *ML for the working programmer* (second ed.). Cambridge University Press.

Uday S. Reddy. 1997. When Parametricity implies Naturality (Notes). (1997). https://citeseerx.ist.psu.edu/pdf/c54b8f6633e2ad8a55195ba24ffae1e3d377407a Accessed: 2025-02-12.

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. North-Holland/IFIP, 513–523.

Christian Sternagel. 2013. Proof Pearl - A Mechanized Proof of GHC's Mergesort. *J. Autom. Reason.* 51, 4 (2013), 357–370. doi:10.1007/s10817-012-9260-7

Enrico Tassi. 2019. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA (LIPIcs, Vol. 141)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:18. doi:10.4230/LIPICS.ITP.2019.29

The GHC Team. 2024a. Glasgow Haskell Compiler User's Guide. https://downloads.haskell.org/~ghc/9.8.4/docs/users_guide/ Accessed: 2025-02-21, Version 9.8.4.

The GHC Team. 2024b. Glasgow Haskell Compiler User's Guide. https://downloads.haskell.org/~ghc/9.12.1/docs/users_guide/ Accessed: 2025-02-21, Version 9.12.1.

The Rocq Development Team. 2025a. *The Rocq Reference Manual*. https://rocq-prover.org/doc/V9.0.0/refman/ the PDF version with numbered sections is available at https://github.com/coq/coq/releases/tag/V9.0.0.

The Rocq Development Team. 2025b. *Section 2.1.11 "Sections"*. In [The Rocq Development Team 2025a]. https://rocq-prover.org/doc/V9.0.0/refman/language/core/sections.html

The Rocq Development Team. 2025c. *Section 2.1.12 "The Module System"*. In [The Rocq Development Team 2025a]. https://rocq-prover.org/doc/V9.0.0/refman/language/core/modules.html

The Rocq Development Team. 2025d. *Section 2.1.9 "Inductive types and recursive functions"*. In [The Rocq Development Team 2025a]. https://rocq-prover.org/doc/V9.0.0/refman/language/core/inductive.html

The Rocq Development Team. 2025e. *Section 2.2.6 "Implicit Coercions"*. In [The Rocq Development Team 2025a]. https://rocq-prover.org/doc/V9.0.0/refman/addendum/implicit-coercions.html

The Rocq Development Team. 2025f. *Section 4.1.1 "The Coq libraries"*. In [The Rocq Development Team 2025a]. https://rocq-prover.org/doc/V9.0.0/refman/language/coq-library.html

Laurent Théry. 2008. merging topred with trunk. https://github.com/math-comp/mathcomp-history-before-github/commit/777eb0fa2ed9d4d157df8b7a90922ca0eebca65a Accessed: 2025-02-13.

Janis Voigtländer. 2008. Much ado about two (pearl): a pearl on parallel prefix computation. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 29–35. doi:10.1145/1328438.1328445

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. ACM, 347–359. doi:10.1145/99370.99404

Philip Wadler. 1990. Recursive types for free! https://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt Accessed: 2025-01-29.

A Basic definitions and facts used for proofs

This appendix provides a list of some definitions and lemmas in the MathComp library [Mahboubi and Tassi 2022] used for the proofs in Sections 3.3, 3.4 and 4.3.4 and Appendix B. Most informal definitions and lemmas are followed by the corresponding formal definitions and statements in Rocq. Some of these formal definitions and statements are modified to avoid introducing new definitions but still convertible with the original definitions and statements.

We use the following `Implicit Types` declaration to interpret the formal definitions and statements in Appendices A to C.

```
Implicit Types (sort : stableSort) (T R S : Type).
```

A.1 Predicates and relations

Definition A.1 (Unary predicates and binary relations). A unary *predicate* $p \subseteq T$ and a binary *relation* $R \subseteq T \times T$ on a type T are functions of types $T \to \mathrm{bool}$ and $T \to T \to \mathrm{bool}$, respectively:

```
Definition pred T : Type := T → bool.
Definition rel T : Type := T → pred T.
```

We sometimes generalize relations to ones between two types T and S; that is, a relation $R \subseteq T \times S$ between T and S is a function of type $T \to S \to \mathrm{bool}$, *e.g.*, Definition A.31. We also abuse this terminology to mean any subset of T and $T \times T$, that is not necessarily decidable and respectively corresponds to any function of type `T → Prop` and `T → T → Prop` in Rocq.

Definition A.2 (Totality of binary relations). A binary relation R on type T is *total* if $x\ R\ y$ or $y\ R\ x$ holds for any $x, y \in T$.

```
Definition total T (R : rel T) : Prop := ∀ x y : T, R x y || R y x.
```

Definition A.3 (Transitive relations). A binary relation R on type T is *transitive* if $x\ R\ y$ and $y\ R\ z$ imply $x\ R\ z$ for any $x, y, z \in T$.

```
Definition transitive T (R : rel T) : Prop :=
  ∀ y x z : T, R x y → R y z → R x z.
```

Definition A.4 (Antisymmetric relations). A binary relation R on type T is *antisymmetric* if $x\ R\ y$ and $y\ R\ x$ imply $x = y$ for any $x, y \in T$.

```
Definition antisymmetric T (R : rel T) : Prop :=
  ∀ x y : T, R x y && R y x → x = y.
```

Definition A.5 (Reflexive relations). A binary relation R on type T is *reflexive* if $x\ R\ x$ holds for any $x \in T$.

```
Definition reflexive T (R : rel T) : Prop := ∀ x : T, R x x.
```

Definition A.6 (Irreflexive relations). A binary relation R on type T is *irreflexive* if $x\ R\ x$ does not for any $x \in T$.

```
Definition irreflexive T (R : rel T) : Prop := ∀ x : T, R x x = false.
```

Definition A.7 (Preorders and orders). A binary relation $\leq$ on type T is said to be:

- a *total preorder* if $\leq$ is transitive and total,
- a *strict preorder* if $\leq$ is transitive and irreflexive, and
- a *total order* if $\leq$ is transitive, antisymmetric, and total.

Definition A.8 (Lexicographic relations). Given two binary relations $\leq_1$ and $\leq_2$ on type T, their lexicographic relation $\leq_{(1,2)}$ is defined as follows:

$$x \leq_{(1,2)} y \coloneqq x \leq_1 y \wedge (y \not\leq_1 x \vee x \leq_2 y).$$

We also write it as $\leq_{\text{lex}}$ when there is only one lexicographic relation in the context and thus there is no ambiguity.

```
Definition lexord T (leT leT' : rel T) ≔
  [rel x y | leT x y && (leT y x ==> leT' x y)].
```

Lemma A.9. *The lexicographic relation $\leq_{(1,2)}$ is total (resp. transitive) whenever both $\leq_1$ and $\leq_2$ are total (resp. transitive).*

```
Lemma lexord_total T (leT leT' : rel T) :
  total leT → total leT' → total (lexord leT leT').

Lemma lexord_trans T (leT leT' : rel T) :
  transitive leT → transitive leT' → transitive (lexord leT leT').
```

Lemma A.10. *The lexicographic composition of binary relations is associative; that is, the left associative composition $\leq_{((1,2),3)}$ is the same relation as the right associative one $\leq_{(1,(2,3))}$.*

```
Lemma lexordA T (leT leT' leT'' : rel T) :
  lexord leT (lexord leT' leT'') =2 lexord (lexord leT leT') leT''.
```

A.2 Lists and list surgery operators

Definition A.11 (Lists). A list of type list T is a finite sequence of values of type T; that is, the set of lists is defined as the least fixed point of the following rules:

- the empty sequence [] (nil) is a list of type list T for any type T, and
- a cons cell $x :: s$ is a list of type list T if x and s have type T and list T, respectively.

```
Inductive list (A : Type) : Type ≔ nil : list A | cons : A → list A → list A.
```

We write list literals of the form $(x_1 :: x_2 :: \cdots :: x_n :: [])$ as $[x_1, x_2, \ldots, x_n]$, or `[:: x1; x2; ...; xn]` in Rocq.

Definition A.12 (Concatenation). Given two lists $s_1 \coloneqq [x_1, \ldots, x_n]$ and $s_2 \coloneqq [y_1, \ldots, y_m]$, cat s_1 s_2, also written as $s_1 +\!\!+ s_2$, concatenates s_1 and s_2, *i.e.*, $[x_1, \ldots, x_n, y_1, \ldots, y_m]$. This is a Rocq equivalent of `List.append` in OCaml.

```
Definition cat T : list T → list T → list T ≔
  fix cat (s1 s2 : list T) {struct s1} : list T ≔
    if s1 is x :: s1' then x :: cat s1' s2 else s2.

Notation "s1 ++ s2" ≔ (cat s1 s2).
```

Lemma A.13. *The concatenation operator $+\!\!+$ is associative.*

```
Lemma catA T (x y z : list T) : x ++ (y ++ z) = (x ++ y) ++ z.
```

Definition A.14 (Right fold). Given a function f, an initial value z, and a list $s \coloneqq [x_1, x_2, \ldots, x_n]$, foldr f z s is the right fold of s with f and z, *i.e.*, $f\, x_1\, (f\, x_2\, (\ldots (f\, x_n\, z) \ldots))$.

```
Definition foldr T R (f : T → R → R) (z0 : R) : list T → R ≔
  fix foldr (s : list T) {struct s} : R ≔
    if s is x :: s' then f x (foldr s') else z0.
```

Definition A.15 (Flattening). Given a list of lists $ss = [s_1, s_2, \ldots, s_n]$, `flatten` ss is the list obtained by folding ss with concatenation, *i.e.*, $s_1 +\!\!+ s_2 +\!\!+ \ldots +\!\!+ s_n$. This is a Rocq equivalent of `List.flatten` in OCaml.

```
Definition flatten T : list (list T) → list T ≔ foldr (@cat T) [::].
```

Definition A.16. Given two lists $s_1 \coloneqq [x_1, \ldots, x_n]$ and $s_2 \coloneqq [y_1, \ldots, y_m]$, `catrev` $s_1\ s_2$ reverses s_1 and concatenates it with s_2, *i.e.*, $[x_n, \ldots, x_1, y_1, \ldots, y_m]$. This is a Rocq equivalent of `List.rev_append` in OCaml.

```
Definition catrev T : list T → list T → list T ≔
  fix catrev (s1 s2 : list T) {struct s1} : list T ≔
    if s1 is x :: s1' then catrev s1' (x :: s2) else s2.
```

Definition A.17 (List reversal). Given a list $s = [x_1, x_2, \ldots, x_n]$, `rev` s is the reversal of s, *i.e.*, $[x_n, \ldots, x_2, x_1]$. This is a Rocq equivalent of `List.rev` in OCaml.

```
Definition rev T (s : list T) : list T ≔ catrev s [::].
```

Lemma A.18. *For any lists s and t,* `catrev` $s\ t$ = `rev` $s +\!\!+ t$ *holds.*

```
Lemma catrevE T (s t : list T) : catrev s t = rev s ++ t.
```

Lemma A.19. `rev` *is involutive; that is,* `rev` (`rev` s) $= s$ *holds for any list s.*

```
Lemma revK T (s : list T) : rev (rev s) = s.
```

Lemma A.20. *For any lists s and t,* `rev` $(s +\!\!+ t)$ = `rev` $t +\!\!+$ `rev` s *holds.*

```
Lemma rev_cat T (s t : list T) : rev (s ++ t) = rev t ++ rev s.
```

Definition A.21. Given a natural number n and a list s, `take` $n\ s$ and `drop` $n\ s$ are respectively

- the list collecting the first n elements of s, or s if the length of s is less than n, and
- the list obtained by dropping the first n elements of s, or [] if the length of s is less than n.

Their pair (`take` $n\ s$, `drop` $n\ s$) is a Rocq equivalent of `List.split_n` in OCaml.

```
Definition take T : nat → list T → list T ≔
  fix take (n : nat) (s : list T) {struct s} : list T ≔
    match s, n with
    | [::], _ | _, 0 ⇒ nil
    | x :: s', S n' ⇒ x :: take n' s'
    end.

Definition drop T : nat → list T → list T ≔
  fix drop (n : nat) (s : list T) {struct s} : list T ≔
    match s, n with
    | [::], _ | _ :: _, 0 ⇒ s
    | _ :: s', S n' ⇒ drop n' s'
    end.
```

Lemma A.22. *For any natural number n and list s,* `take` $n\ s +\!\!+$ `drop` $n\ s = s$ *holds.*

```
Lemma cat_take_drop (n : nat) T (s : list T) : take n s ++ drop n s = s.
```

A.3 Counting and size

Definition A.23. Given a predicate $p \subseteq T$ and a list s of type list T, $|s|_a$ is the number of elements in s satisfying p.

```
Definition count T (p : pred T) : list T → nat :=
  foldr (fun x n ⇒ p x + n) 0.
```

Definition A.24. Given a list s, $|s|$ is the length of s. This is a Rocq equivalent of `List.length` in OCaml.

```
Definition size T : list T → nat := foldr (fun _ n ⇒ S n) 0.
```

Lemma A.25. *For any s of type* list T, $|s|_T = |s|$ *holds.*

```
Lemma count_predT T (s : list T) : count (fun _ ⇒ true) s = size s.
```

Lemma A.26. *For any $p \subseteq T$ and s_1 and s_2 of type* list T, $|s_1 +\!\!+\, s_2|_p = |s_1|_p + |s_2|_p$ *holds.*

```
Lemma count_cat T (p : pred T) (s1 s2 : list T) :
  count p (s1 ++ s2) = count p s1 + count p s2.
```

Lemma A.27. *For any $p \subseteq T$ and s of type* list T, $|\mathsf{rev}\, s|_p = |s|_p$ *holds.*

```
Lemma count_rev T (p : pred T) (s : list T) : count a (rev s) = count a s.
```

A.4 Predicates on lists

Definition A.28. Given a predicate $p \subseteq T$ and a list s of type list T, $\mathtt{all}_p\, s$ holds if all elements of s satisfy p.

```
Definition all T (p : pred T) : list T → bool :=
  foldr (fun x b ⇒ p x && b) true.
```

Lemma A.29. *For any $p \subseteq T$ and s of type* list T, $\mathtt{all}_p\, s$ *holds iff* $|s|_p = |s|$.

```
Lemma all_count T (p : pred T) (s : list T) : all p s = (count p s == size s).
```

Lemma A.30. *For any $p_1 \subseteq p_2 \subseteq T$ and s of type* list T, $\mathtt{all}_{p_2}\, s$ *holds whenever* $\mathtt{all}_{p_1}\, s$ *holds.*

```
Lemma sub_all T (p1 p2 : pred T) :
  (∀ x : T, p1 x → p2 x) → ∀ s : list T, all p1 s → all p2 s.
```

Definition A.31. Given a relation $R \subseteq T \times S$ and two lists s_1 and s_2 of respectively types list T and list S, $\mathtt{allrel}_R\, s_1\, s_2$ holds if any elements x of s_1 and y of s_2 satisfy $x\; R\; y$.

```
Definition allrel T S (r : T → S → bool) (xs : list T) (ys : list S) : bool :=
  all (fun x ⇒ all (r x) ys) xs.
```

Lemma A.32. *For any $R \subseteq T \times S$ and s_1 and s_2 of respectively types* list T *and* list S, $\mathtt{allrel}_R\, s_1\, s_2$ *holds iff* $\mathtt{allrel}_{R^C}\, s_2\, s_1$ *holds, where $R^C \subseteq S \times T$ is the converse relation of R,* i.e., $y\; R^C\; x := x\; R\; y$.

```
Lemma allrelC T S (r : T → S → bool) (xs : list T) (ys : list S) :
  allrel r xs ys = allrel (fun x y ⇒ r y x) ys xs.
```

Lemma A.33. *For any relation $R \subseteq T \times S$ and s_1 and s_2 of respectively types* list T *and* list S, $\mathtt{allrel}_R\, (\mathsf{rev}\, s_1)\, (\mathsf{rev}\, s_2)$ *holds iff* $\mathtt{allrel}_R\, s_1\, s_2$ *holds.*

```
Lemma allrel_rev2 T S (r : T → S → bool) (s1 : list T) (s2 : list S) :
  allrel r (rev s1) (rev s2) = allrel r s1 s2.
```

Definition A.34. Given a list s, `uniq` s holds if all the elements of s are pairwise different, *i.e.*, s is duplication free.

```
Definition uniq (T : eqType) : list T → bool :=
  fix uniq (s : list T) {struct s} : bool :=
    if s is x :: s' then (x ∉ s') && uniq s' else true.
```

where `x ∉ s'` means that `x` is not an element of `s'`, which requires `T` to be an `eqType`, a type with a decidable equality.

A.5 Map and filter

Definition A.35 (Map). Given a function f of type $T_1 \to T_2$ and a list $s := [x_1, x_2, \ldots, x_n]$ of type list T_1, $\mathtt{map}_f\ s$ is the list of type list T_2 mapping f to the elements of s, *i.e.*, $[f\ x_1, f\ x_2, \ldots, f\ x_n]$. This is a Rocq equivalent of `List.map` in OCaml.

```
Definition map T1 T2 (f : T1 → T2) : list T1 → list T2 :=
  foldr (fun x s ⇒ f x :: s) [::].
```

Definition A.36 (Filter). Given a predicate $p \subseteq T$ and a list s of type list T, $\mathtt{filter}_p\ s$ is the list collecting all the elements of s that satisfy p, and preserves the order of the elements in the input. This is a Rocq equivalent of `List.filter` in OCaml.

```
Definition filter T (p : pred T) : list T → list T :=
  foldr (fun x s ⇒ if p x then x :: s else s) [::].
```

Lemma A.37 (The naturality of `filter`). *For any f of type $T_1 \to T_2$, $p \subseteq T_2$, and s of type* list T_1, $\mathtt{filter}_p\ (\mathtt{map}_f\ s) = \mathtt{map}_f\ (\mathtt{filter}_{p \circ f}\ s)$ *holds.*

```
Lemma filter_map T1 T2 (f : T1 → T2) (p : pred T2) (s : list T1) :
  filter p (map f s) = map f (filter (fun x ⇒ p (f x)) s).
```

Lemma A.38. *For any $p \subseteq T$, x of type T, and s of type* list T, *x is an element of* $\mathtt{filter}_p\ s$ *iff x is an element of s that satisfies p.*

```
Lemma mem_filter (T : eqType) (p : pred T) (x : T) (s : list T) :
  (x ∈ filter p s) = p x && (x ∈ s).
```

A.6 Subsequences

Definition A.39 (Mask). Given two lists m and s of respectively types list bool and list T, $\mathtt{mask}_m\ s$ is the subsequence of s selected by m; that is, the ith element of s is selected if the ith element of m is `true`.

```
Definition mask T : list bool → list T → list T :=
  fix mask (m : list bool) (s : list T) {struct m} : list T :=
    match m, s with
    | [::], _ | _, [::] ⇒ [::]
    | b :: m', x :: s' ⇒ if b then x :: mask m' s' else mask m' s'
    end.
```

Lemma A.40 (The naturality of `mask`). *For any f of type $T_1 \to T_2$, and m and s of respectively types* list bool *and* list T_1, $\mathtt{map}_f\ (\mathtt{mask}_m\ s) = \mathtt{mask}_m\ (\mathtt{map}_f\ s)$ *holds.*

```
Lemma map_mask T1 T2 (f : T1 → T2) (m : list bool) (s : list T1) :
  map f (mask m s) = mask m (map f s).
```

Lemma A.41. *For any $p \subseteq T$ and s of type* list T, $\texttt{filter}_p\, s = \texttt{mask}_{m'}\, s$ *where* $m' \coloneqq \texttt{map}_p\, s$ *holds.*

```
Lemma filter_mask T (p : pred T) (s : list T) : filter p s = mask (map p s) s.
```

Lemma A.42. *For any s and m of respectively types* list T *and* list bool, $\texttt{mask}_m\, s = \texttt{filter}_p\, s$ *where* $p\, x \coloneqq x \in \texttt{mask}_m\, s$ *holds whenever s is duplication free.*

```
Lemma mask_filter (T : eqType) (s : list T) (m : list bool) :
  uniq s → mask m s = filter [in mask m s] s.
```

Definition A.43 (Subsequence relation). Given two lists s_1 and s_2, subseq $s_1\, s_2$ means that s_1 is a subsequence of s_2.

```
Definition subseq (T : eqType) : list T → list T → bool :=
  fix subseq (s1 s2 : list T) {struct s2} : bool :=
    match s2, s1 with
      | _, [::] ⇒ true
      | [::], _ :: _ ⇒ false
      | y :: s2', x :: s1' ⇒ subseq (if x == y then s1' else s1) s2'
    end.
```

Lemma A.44. *For any lists s_1 and s_2, s_1 is a subsequence of s_2 iff there exists a list m of type* list bool *such that $|m| = |s_2|$ and* $s_1 = \texttt{mask}_m\, s_2$.

```
Lemma subseqP (T : eqType) (s1 s2 : list T) :
  reflect
    (exists2 m : list bool, size m = size s2 & s1 = mask m s2) (subseq s1 s2).
```

Lemma A.45. *For any m and s of respectively types* list bool *and* list T, $\texttt{mask}_m\, s$ *is a subsequence of s.*

```
Lemma mask_subseq (T : eqType) (m : list bool) (s : list T) :
  subseq (mask m s) s.
```

Definition A.46. For any x of type T and s of type list T, $\texttt{index}_x\, s$ is the index of the first occurrence of x in s.

```
Definition index (T : eqType) (x : T) : list T → nat :=
  fix find (s : list T) {struct s} : nat :=
    if s is y :: s' then
      if y == x then 0 else S (find s')
    else
      0.
```

Definition A.47. For any s of type list T, and x and y of type T, we say that x and y occur in order in s, or write mem2 $s\, x\, y$, when y occurs in s (non-strictly) after the first occurrence of x. Here, "non-strictly" means that mem2 $s\, x\, x$ holds even if x occurs in s only once.

```
Definition mem2 (T : eqType) (s : list T) (x y : T) : bool :=
  y ∈ drop (index x s) s.
```

Lemma A.48. *For any s of type* list T*, and x and y of type T, x and y occur in order in s iff the following xy is a subsequence of s:*

$$xy \coloneqq \begin{cases} [x] & (x = y) \\ [x, y]. & (otherwise) \end{cases}$$

```
Lemma mem2E (T : eqType) (s : list T) (x y : T) :
  mem2 s x y = subseq (if x == y then [:: x] else [:: x; y]) s.
```

A.7 Permutation

Definition A.49 (Permutation relation). Two lists s_1 and s_2 are *permutation* of each other iff $|s_1|_{\{x\}} = |s_2|_{\{x\}}$ for any element x of s_1 or s_2, then we write $s_1 =_{\mathtt{perm}} s_2$.

```
Definition perm_eq (T : eqType) (s1 s2 : list T) : bool ≔
  all [pred x | count (pred1 x) s1 == count (pred1 x) s2] (s1 ++ s2).
```

In order to use the fact $s_1 =_{\mathtt{perm}} s_2$ to rewrite a goal of the form $s_1 =_{\mathtt{perm}} s_3$ to $s_2 =_{\mathtt{perm}} s_3$, some lemmas, *e.g.*, Lemmas A.55 and A.56, are stated in the form of $\forall s_3, (s_1 =_{\mathtt{perm}} s_3) = (s_2 =_{\mathtt{perm}} s_3)$, using the following notation (see also Lemma A.51).

```
Notation perm_eql s1 s2 ≔ (perm_eq s1 =1 perm_eq s2).
```

Lemma A.50. *For any lists s_1 and s_2 of type* list T*, $s_1 =_{\mathtt{perm}} s_2$ holds iff $|s_1|_p = |s_2|_p$ holds for any predicate $p \subseteq T$.*

```
Lemma permP (T : eqType) (s1 s2 : list T) :
  reflect (∀ p : pred T, count p s1 = count p s2) (perm_eq s1 s2).
```

Lemma A.51. *For any lists s_1 and s_2, $s_1 =_{\mathtt{perm}} s_2$ holds iff $(s_1 =_{\mathtt{perm}} s_3) = (s_2 =_{\mathtt{perm}} s_3)$ holds for any list s_3.*

```
Lemma permPl (T : eqType) (s1 s2 : list T) :
  reflect (perm_eql s1 s2) (perm_eq s1 s2).
```

Lemma A.52. *For any lists s_1 and s_2 of type* list T *such that $s_1 =_{\mathtt{perm}} s_2$ and x of type T, $x \in s_1$ iff $x \in s_2$.*

```
Lemma perm_mem (T : eqType) (s1 s2 : list T) : perm_eq s1 s2 → s1 =i s2.
```

where `s1 =i s2 ≔ (∀ x, x ∈ s1 = x ∈ s2)` *means that* `s1` *and* `s2` *have the same set of elements.*

Lemma A.53. *For any lists s_1 and s_2 such that $s_1 =_{\mathtt{perm}} s_2$, s_1 is duplication free iff s_2 is duplication free.*

```
Lemma perm_uniq (T : eqType) (s1 s2 : list T) :
  perm_eq s1 s2 → uniq s1 = uniq s2.
```

Lemma A.54. *The permutation relation $=_{\mathtt{perm}}$ is a congruence relation with respect to concatenation $+\!\!+$; that is, $s_1 =_{\mathtt{perm}} s_2$ and $t_1 =_{\mathtt{perm}} t_2$ imply $s_1 +\!\!+ t_1 =_{\mathtt{perm}} s_2 +\!\!+ t_2$ for any lists s_1, s_2, t_1, and t_2.*

```
Lemma perm_cat (T : eqType) (s1 s2 t1 t2 : list T) :
  perm_eq s1 s2 → perm_eq t1 t2 → perm_eq (s1 ++ t1) (s2 ++ t2).
```

Lemma A.55. *For any lists s_1 and s_2, $s_1 +\!\!+ s_2$ is a permutation of $s_2 +\!\!+ s_1$.*

```
Lemma perm_catC (T : eqType) (s1 s2 : list T) : perm_eql (s1 ++ s2) (s2 ++ s1).
```

Lemma A.56. *For any list s,* `rev` *s is a permutation of s.*

```
Lemma perm_rev (T : eqType) (s : list T) : perm_eql (rev s) s.
```

A.8 Sortedness

Definition A.57 (Pairwise sortedness, Definition 3.9). Given a relation $R \subseteq T \times T$, a list $s \coloneqq [x_0, \dots, x_n]$ of type list T is said to be *pairwise sorted* w.r.t. R if the relation R holds any x_i and x_j such that $i < j \le n$, *i.e.*,

$$x_0 \; R \; x_1 \wedge \cdots \wedge x_0 \; R \; x_n \wedge x_1 \; R \; x_2 \wedge \cdots \wedge x_1 \; R \; x_n \wedge \cdots \wedge x_{n-1} \; R \; x_n.$$

```
Definition pairwise T (r : T → T → bool) : list T → bool ≔
  fix pairwise (xs : list T) {struct xs} : bool ≔
    if xs is x :: xs0 then all (r x) xs0 && pairwise xs0 else true.
```

Lemma A.58. *For any relation $R \subseteq T \times T$ and lists s_1 and s_2 of type* list T, *s_1 ++ s_2 is pairwise sorted w.r.t. R iff the following conjunction holds:*

- *$x \; R \; y$ holds for any $x \in s_1$ and $y \in s_2$, and*
- *both s_1 and s_2 are pairwise sorted w.r.t. R.*

```
Lemma pairwise_cat T (r : T → T → bool) (s1 s2 : list T) :
  pairwise r (s1 ++ s2) = allrel r s1 s2 && (pairwise r s1 && pairwise r s2).
```

Definition A.59 (Path and sortedness, Definition 3.9). Given a relation $R \subseteq T \times T$, a list $s \coloneqq [x_0, \dots, x_n]$ of type list T is said to be *sorted* if R holds for each adjacent pair in s, *i.e.*, $x_0 \; R \; x_1 \wedge \cdots \wedge x_{n-1} \; R \; x_n$. A cons cell $x :: s$ is said to be an R-path if it is sorted w.r.t. R.

In Rocq, the former is defined using the latter:

```
Definition path T (e : rel T) : T → list T → bool ≔
  fix path (x : T) (p : list T) {struct p} : bool ≔
    if p is y :: p' then e x y && path y p' else true.

Definition sorted T (e : rel T) (s : list T) : bool ≔
  if s is x :: s' then path e x s' else true.
```

Lemma A.60. *The sortedness and the pairwise sortedness are equivalent for transitive relations.*

```
Lemma sorted_pairwise T (r : rel T) :
  transitive r → ∀ s : list T, sorted r s = pairwise r s.
```

Lemma A.61. *For any relation $R \subseteq T \times T$ and list s of type* list T, `rev` *s is sorted w.r.t. R iff s is sorted w.r.t. its converse relation R^C.*

```
Lemma rev_sorted T (r : rel T) (s : list T) :
  sorted r (rev s) = sorted (fun x y : T ⇒ r y x) s.
```

Lemma A.62. *For any transitive relation $R \subseteq T \times T$, predicate $p \subseteq T$, and list s of type* list T, `filter`$_p$ *s is sorted w.r.t. R whenever s is sorted w.r.t. R.*

```
Lemma sorted_filter T (r : rel T) : transitive r →
  ∀ (p : pred T) (s : list T), sorted r s → sorted r (filter p s).
```

Lemma A.63. *For any transitive and antisymmetric relation* $\leq\, \subseteq T \times T$ *and lists* s_1 *and* s_2 *of type* list T, $s_1 = s_2$ *holds whenever* $s_1 =_{\mathtt{perm}} s_2$ *and both* s_1 *and* s_2 *are sorted w.r.t.* $\leq$.

```
Lemma sorted_eq (T : eqType) (leT : rel T) :
  transitive leT → antisymmetric leT →
  ∀ s1 s2 : list T,
    sorted leT s1 → sorted leT s2 → perm_eq s1 s2 → s1 = s2.
```

Lemma A.64. *For any strict preorder* $\leq\, \subseteq T \times T$ *and lists* s_1 *and* s_2 *of type* list T, $s_1 = s_2$ *holds whenever* s_1 *and* s_2 *contain the same set of elements and both of them are sorted w.r.t.* $\leq$.

```
Lemma irr_sorted_eq (T : eqType) (leT : rel T) :
  transitive leT → irreflexive leT →
  ∀ s1 s2 : list T, sorted leT s1 → sorted leT s2 → s1 =i s2 → s1 = s2.
```

A.9 Indexing

Definition A.65. Given two natural numbers m and n, `iota` $m\,n$ is the list of natural numbers $[m, m+1, \ldots, m+n-1]$.

```
Fixpoint iota (m n : nat) {struct n} : list nat :=
  if n is S n' then m :: iota (S m) n' else [::].
```

Definition A.66. Given x_0 of type T, a list s of type list T, a natural number n, `nth` $x_0\,s\,n$ is the ith element of s, except that it is x_0 when $n \leq |s|$.

```
Definition nth T (x0 : T) : list T → nat → T :=
  fix nth (s : list T) (n : nat) {struct n} : T :=
    match s, n with
    | [::], _ ⇒ x0
    | x :: _, 0 ⇒ x
    | _ :: s', S n' ⇒ nth s' n'
    end.
```

Lemma A.67. *For any* x_0 *of type* T *and* s *of type* list T, s *is equal to* $[\mathtt{nth}\, x_0\, s\, i \mid i \leftarrow \mathtt{iota}\, 0\, |s|]$.

```
Lemma mkseq_nth T (x0 : T) (s : list T) : map (nth x0 s) (iota 0 (size s)) = s.
```

Lemma A.68. *Any* `iota` *sequence is duplication free.*

```
Lemma iota_uniq (m n : nat) : uniq (iota m n).
```

Lemma A.69. *Any* `iota` *sequence is sorted w.r.t. the strict less than relation of natural numbers.*

```
Lemma iota_ltn_sorted (i n : nat) : sorted ltn (iota i n).
```

A.10 Merge

Definition A.70 (Merge). Given a relation $\leq\, \subseteq T \times T$ and two lists s_1 and s_2 of type list T, their merge $s_1 \curlywedge_{\leq} s_2$ is a list of type list T defined as follows:

$$
\begin{aligned}
[] \curlywedge_{\leq} s_2 &\coloneqq s_2 \\
s_1 \curlywedge_{\leq} [] &\coloneqq s_1 \\
(x :: s_1) \curlywedge_{\leq} (y :: s_2) &\coloneqq \begin{cases} x :: (s_1 \curlywedge_{\leq} (y :: s_2)) & (x \leq y) \\ y :: ((x :: s_1) \curlywedge_{\leq} s_2). & (\text{otherwise}) \end{cases}
\end{aligned}
$$

```
Definition merge T (leT : rel T) ≔
  fix merge (s1 : list T) {struct s1} : list T → list T ≔
    match s1 with
    | [::] ⇒ id
    | x1 :: s1' ⇒
      fix merge_s1 (s2 : list T) {struct s2} : list T ≔
        match s2 with
        | [::] ⇒ s1
        | x2 :: s2' ⇒
          if leT x1 x2 then x1 :: merge s1' s2 else x2 :: merge_s1 s2'
        end
    end.
```

Lemma A.71. *For any* $\leq\ \subseteq T \times T$, $p \subseteq T$, *and* s_1 *and* s_2 *of type* list T, $|s_1 \mathbin{⩕_{\leq}} s_2|_p$ *is equal to* $|s_1 +\!\!+\, s_2|_p$.

```
Lemma count_merge T (leT : rel T) (p : pred T) (s1 s2 : list T) :
  count p (merge leT s1 s2) = count p (s1 ++ s2).
```

Lemma A.72. *For any* $\leq\ \subseteq T \times T$, *and* s_1 *and* s_2 *of type* list T, $s_1 \mathbin{⩕_{\leq}} s_2$ *is a permutation of* $s_1 +\!\!+\, s_2$.

```
Lemma perm_merge (T : eqType) (r : rel T) (s1 s2 : list T) :
  perm_eql (merge r s1 s2) (s1 ++ s2).
```

Lemma A.73. *For any* $\leq\ \subseteq T \times T$, *and* s_1 *and* s_2 *of type* list T, $s_1 \mathbin{⩕_{\leq}} s_2$ *is equal to* $s_1 +\!\!+\, s_2$ *whenever* $x \leq y$ *holds for any* $x \in s_1$ *and* $y \in s_2$.

```
Lemma allrel_merge T (leT : rel T) (s1 s2 : list T) :
  allrel leT s1 s2 → merge leT s1 s2 = s1 ++ s2.
```

Lemma A.74. *For any* $\leq_1, \leq_2\ \subseteq T \times T$, *and* s_1 *and* s_2 *of type* list T, $s_1 \mathbin{⩕_{\leq_1}} s_2$ *is sorted w.r.t. the lexicographic order* $\leq_{\text{lex}} \coloneqq \leq_{(1,2)}$ *whenever* $\leq_1$ *is total,* $x \leq_2 y$ *holds for any* $x \in s_1$ *and* $y \in s_2$, *and both* s_1 *and* s_2 *are sorted w.r.t.* $\leq_{\text{lex}}$.

```
Lemma merge_stable_sorted T (r r' : rel T) :
  total r → ∀ s1 s2 : list T,
  allrel r' s1 s2 → sorted (lexord r r') s1 → sorted (lexord r r') s2 →
  sorted (lexord r r') (merge r s1 s2).
```

Lemma A.75. *For any total preorder* $\leq\ \subseteq T \times T$, $⩕_{\leq}$ *is associative.*

```
Lemma mergeA T (leT : rel T) :
  total leT → transitive leT → associative (merge leT).
```

A.11 Sigma types

Lemma A.76. *Suppose* T_1 *and* T_2 *are types,* $D_1 \subset T_1$ *and* $D_2 \subseteq T_2$ *are predicates on them,* $P \subseteq T_1 \times T_2$ *is a relation, and* $\mathtt{sig}\, D_1$ *and* $\mathtt{sig}\, D_1$ *are subtypes of* T_1 *and* T_2 *collecting inhabitants satisfying* D_1 *and* D_2, *respectively. Then,* $(\mathtt{val}\, x, \mathtt{val}\, y) \in P$ *holds for any* $x \in \mathtt{sig}\, D_1$ *and* $y \in \mathtt{sig}\, D_2$, *whenever* $(x, y) \in P$ *holds for any* $x \in T_1$ *and* $y \in T_2$ *such that* $x \in D_1$ *and* $y \in D_2$.

Its ternary version also holds.

```
Lemma in2_sig T1 T2 (D1 : pred T1) (D2 : pred T2) (P2 : T1 → T2 → Prop) :
  {in D1 & D2, ∀ (x : T1) (y : T2), P2 x y} →
  ∀ (x : sig D1) (y : sig D2), P2 (val x) (val y).

Lemma in3_sig
    T1 T2 T3 (D1 : pred T1) (D2 : pred T2) (D3 : pred T3)
    (P3 : T1 → T2 → T3 → Prop) :
  {in D1 & D2 & D3, ∀ (x : T1) (y : T2) (z : T3), P3 x y z} →
  ∀ (x : sig D1) (y : sig D2) (z : sig D3), P3 (val x) (val y) (val z).
```

Lemma A.77. *For any $p \subseteq T$ and a list s of type* list T, *there exists a list s' of type* list (sig p) *such that s is equal to* $\mathtt{map}_{\mathtt{val}}\ s'$, *whenever* $\mathtt{all}_p\ s$ *holds.*

```
Lemma all_sigP T (p : pred T) (s : list T) :
  all p s → {s' : list (sig p) | s = map val s'}.
```

B The theory of stable sort functions

This appendix provides the list of all lemmas and their proofs about stable sort functions we stated and proved using the `stableSort` structure (Section 5.2.1). Each informal statement is followed by the corresponding formal statements in Rocq, which include corollaries such as ones delimiting the domain by a predicate, *e.g.*, Corollary 5.2.

Lemma B.1 (An induction principle over traces of sort, Lemma 4.1). *Suppose $\leq$ and $\sim$ are binary relations on T and* list T, *respectively, and xs is a list of type* list T. *Then, $xs \sim \mathtt{sort}_{\leq}\ xs$ holds whenever the following four induction cases hold:*

- *for any lists xs, xs', ys, and ys' of type* list T, $(xs +\!\!+\ ys) \sim (xs' \curlywedge_{\leq} ys')$ *holds whenever $xs \sim xs'$ and $ys \sim ys'$ hold,*
- *for any lists xs, xs', ys, and ys' of type* list T, $(xs +\!\!+\ ys) \sim \mathtt{rev}\,(\mathtt{rev}\ ys' \curlywedge_{\geq} \mathtt{rev}\ xs')$ *holds whenever $xs \sim xs'$ and $ys \sim ys'$ hold,*
- *for any x of type T, $[x] \sim [x]$ holds, and*
- $[] \sim []$ *holds.*

```
Lemma sort_ind sort T (leT : rel T) (R : list T → list T → Prop) :
  (∀ xs xs' : list T, R xs xs' → ∀ ys ys' : list T, R ys ys' →
    R (xs ++ ys) (merge leT xs' ys')) →
  (∀ xs xs' : list T, R xs xs' → ∀ ys ys' : list T, R ys ys' →
    R (xs ++ ys) (rev (merge (fun x y ⇒ leT y x) (rev ys') (rev xs')))) →
  (∀ x : T, R [:: x] [:: x]) →
  R nil nil →
  ∀ s : list T, R s (sort T leT s).
```

Proof. See Lemmas 3.5 and 4.1. □

Lemma B.2 (The naturality of sort, Lemma 3.14). *Suppose $\leq_T$ is a relation on type T, f is a function from T' to T, and s is a list of type* list T'. *Then, the following equation holds:*

$$\mathtt{sort}_{\leq_T}\ [f\,x \mid x \leftarrow s] = [f\,x \mid x \leftarrow \mathtt{sort}_{\leq_{T'}}\ s]$$

where $x \leq_{T'} y \coloneqq f\,x \leq_T f\,y$.

```
Lemma map_sort sort T T' (f : T' → T) (leT' : rel T') (leT : rel T) :
  (∀ x y : T', leT (f x) (f y) = leT' x y) →
  ∀ s : list T', map f (sort T' leT' s) = sort T leT (map f s).
```

```
Lemma sort_map sort T T' (f : T' → T) (leT : rel T) (s : list T') :
  sort T leT (map f s) = map f (sort T' (relpre f leT) s).
```

Proof. See Lemma 3.14. □

Lemma B.3 (Lemma 3.10). *For any $\leq \subseteq T \times T$ and s of type* list T *pairwise sorted w.r.t.* $\leq$, $\mathsf{sort}_{\leq}\, s$ *is equal to* s.

```
Lemma pairwise_sort sort T (leT : rel T) (s : list T) :
  pairwise leT s → sort T leT s = s.
```

Proof. We prove it by induction on $\mathsf{sort}_{\leq}\, s$ (Lemma B.1). For the first case, we show that $s'_1 \mathbin{⩕}_{\leq} s'_2 = s_1 +\!\!+\, s_2$ holds whenever $s_1 +\!\!+\, s_2$ is pairwise sorted w.r.t. $\leq$, which is equivalent to the following conjunction (Lemma A.58):

(i) both s_1 and s_2 are pairwise sorted w.r.t. $\leq$, and
(ii) $x \leq y$ holds for any $x \in s_1$ and $y \in s_2$.

Then,

$$\begin{aligned} s'_1 \mathbin{⩕}_{\leq} s'_2 &= s_1 \mathbin{⩕}_{\leq} s_2 && \text{((i) and I.H.)} \\ &= s_1 +\!\!+\, s_2. && \text{((ii) and Lemma A.73)} \end{aligned}$$

For the second case, we show that $\mathsf{rev}\,(\mathsf{rev}\, s'_2 \mathbin{⩕}_{\geq} \mathsf{rev}\, s'_1) = s_1 +\!\!+\, s_2$ holds whenever $s_1 +\!\!+\, s_2$ is pairwise sorted w.r.t. $\leq$, which is equivalent to the following conjunction (Lemma A.58):

(iii) both s_1 and s_2 are pairwise sorted w.r.t. $\leq$, and
(iv) $x \leq y$ holds for any $x \in s_1$ and $y \in s_2$, or equivalently (Lemmas A.32 and A.33), $y \geq x$ holds for any $y \in \mathsf{rev}\, s_2$ and $x \in \mathsf{rev}\, s_1$.

Then,

$$\begin{aligned} \mathsf{rev}\,(\mathsf{rev}\, s'_2 \mathbin{⩕}_{\geq} \mathsf{rev}\, s'_1) &= \mathsf{rev}\,(\mathsf{rev}\, s_2 \mathbin{⩕}_{\geq} \mathsf{rev}\, s_1) && \text{((iii) and I.H.)} \\ &= \mathsf{rev}\,(\mathsf{rev}\, s_2 +\!\!+\, \mathsf{rev}\, s_1) && \text{((iv) and Lemma A.73)} \\ &= s_1 +\!\!+\, s_2. && \text{(Lemmas A.19 and A.20)} \end{aligned}$$

The last two cases are trivial. □

Lemma B.4. *For any transitive relation $\leq \subseteq T \times T$ and s of type* list T *sorted w.r.t.* $\leq$, $\mathsf{sort}_{\leq}\, s$ *is equal to* s.

```
Lemma sorted_sort sort T (leT : rel T) :
  transitive leT → ∀ s : list T, sorted leT s → sort T leT s = s.

Lemma sorted_sort_in sort T (P : pred T) (leT : rel T) :
  {in P & &, transitive leT} →
  ∀ s : list T, all P s → sorted leT s → sort T leT s = s.
```

Proof. This follows from Lemmas A.60 and B.3. □

Theorem B.5 (Theorem 3.11). *For any $\leq_1, \leq_2 \subseteq T \times T$ and xs of type* list T, $\mathsf{sort}_{\leq_1}\, xs$ *is sorted w.r.t. the lexicographic order $\leq_{\mathrm{lex}} \coloneqq \leq_{(1,2)}$ whenever $\leq_1$ is total and xs is pairwise sorted w.r.t. $\leq_2$.*

```
Lemma sort_pairwise_stable sort T (leT leT' : rel T) : total leT →
  ∀ s : list T, pairwise leT' s → sorted (lexord leT leT') (sort T leT s).

Lemma sort_pairwise_stable_in sort T (P : pred T) (leT leT' : rel T) :
  {in P &, total leT} → ∀ s : list T, all P s → pairwise leT' s →
  sorted (lexord leT leT') (sort T leT s).
```

Proof. We prove it by induction on $\mathtt{sort}_{\leq}\ xs$ (Lemma B.1). See Sections 3.4.1 and 4.3.4 for details. □

Lemma B.6 (Corollary 3.12). *For any* $\leq_1, \leq_2 \subseteq T \times T$ *and* xs *of type* list T, $\mathtt{sort}_{\leq_1}\ xs$ *is sorted w.r.t. the lexicographic order* $\leq_{\text{lex}} \coloneqq \leq_{(1,2)}$ *whenever* $\leq_1$ *is total,* $\leq_2$ *is transitive, and* xs *is sorted w.r.t.* $\leq_2$.

```
Lemma sort_stable sort T (leT leT' : rel T) : total leT → transitive leT' →
  ∀ s : list T, sorted leT' s → sorted (lexord leT leT') (sort T leT s).

Lemma sort_stable_in sort T (P : pred T) (leT leT' : rel T) :
  {in P &, total leT} → {in P & &, transitive leT'} →
  ∀ s : list T, all P s → sorted leT' s →
  sorted (lexord leT leT') (sort T leT s).
```

Proof. This follows from Lemma A.60 and Theorem B.5. □

Lemma B.7. *For any* $\leq \subseteq T \times T$, $p \subseteq T$, *and* s *of type* list T, *the numbers of the elements of* $\mathtt{sort}_{\leq}\ s$ *and* s *satisfying* p *are the same,* i.e., $|\mathtt{sort}_{\leq}\ s|_p = |s|_p$.

```
Lemma count_sort sort T (leT : rel T) (p : pred T) (s : list T) :
  count p (sort T leT s) = count p s.
```

Proof. We prove it by induction on $\mathtt{sort}_{\leq}\ s$ (Lemma B.1). For the first case,

$$\begin{aligned} |s'_1 \mathbin{\wedge\!\!\wedge}_{\leq} s'_2|_p &= |s'_1|_p + |s'_2|_p && \text{(Lemmas A.26 and A.71)} \\ &= |s_1|_p + |s_2|_p && \text{(I.H.)} \\ &= |s_1 \mathbin{+\!\!+} s_2|_p. && \text{(Lemma A.26)} \end{aligned}$$

For the second case,

$$\begin{aligned} |\mathsf{rev}\,(\mathsf{rev}\, s'_2 \mathbin{\wedge\!\!\wedge}_{\geq} \mathsf{rev}\, s'_1)|_p &= |s'_2|_p + |s'_1|_p && \text{(Lemmas A.26, A.27 and A.71)} \\ &= |s'_1|_p + |s'_2|_p \\ &= |s_1|_p + |s_2|_p && \text{(I.H.)} \\ &= |s_1 \mathbin{+\!\!+} s_2|_p. && \text{(Lemma A.26)} \end{aligned}$$

The last two cases are trivial. □

Lemma B.8. *For any* $\leq \subseteq T \times T$ *and* s *of type* list T, *the lengths of* $\mathtt{sort}_{\leq}\ s$ *and* s *are equal,* i.e., $|\mathtt{sort}_{\leq}\ s| = |s|$.

```
Lemma size_sort sort T (leT : rel T) (s : seq T) : size (sort T leT s) = size s.
```

Proof. This follows from Lemmas A.25 and B.7. □

Lemma B.9. *Sorting the empty list gives the empty list.*

```
Lemma sort_nil sort T (leT : rel T) : sort T leT [::] = [::].
```

Proof. The length of $\texttt{sort}_\leq$ [] is 0 (Lemma B.8), and thus, it must be the empty list. □

Lemma B.10. *For any $p \subseteq T$, $\leq \subseteq T \times T$, and s of type* list T*, all elements of* $\texttt{sort}_\leq s$ *satisfy p iff all elements of s satisfy p.*

```
Lemma all_sort sort T (p : pred T) (leT : rel T) (s : list T) :
  all p (sort T leT s) = all p s.
```

Proof. This follows from Lemmas A.29, B.7 and B.8. □

Lemma B.11 (Lemma 3.7). *For any $\leq \subseteq T \times T$ and s of type* list T, $\texttt{sort}_\leq s$ *is a permutation of s.*

```
Lemma perm_sort sort (T : eqType) (leT : rel T) (s : list T) :
  perm_eql (sort T leT s) s.
```

Proof. This follows from Lemmas A.50 and B.7. □

Lemma B.12 (Corollary 3.8). *For any $\leq \subseteq T \times T$ and s of type* list T, $\texttt{sort}_\leq s$ *has the same set of elements as s.*

```
Lemma mem_sort sort (T : eqType) (leT : rel T) (s : list T) : sort T leT s =i s.
```

Proof. This follows from Lemmas A.52 and B.11. □

Lemma B.13. *For any $\leq \subseteq T \times T$ and s of type* list T, $\texttt{sort}_\leq s$ *is duplication free iff s is duplication free.*

```
Lemma sort_uniq sort (T : eqType) (leT : rel T) (s : list T) :
  uniq (sort T leT s) = uniq s.
```

Proof. This follows from Lemmas A.53 and B.11. □

Theorem B.14 (Theorem 3.16 and Corollary 5.2). *For any total preorder $\leq \subseteq T \times T$ and predicate $p \subseteq T$,* $\texttt{filter}_p$ *commutes with* $\texttt{sort}_\leq$ *under function composition; that is, the following equation holds for any s of type* list T*:*

$$\texttt{filter}_p\,(\texttt{sort}_\leq s) = \texttt{sort}_\leq\,(\texttt{filter}_p\, s).$$

```
Lemma filter_sort sort T (leT : rel T) : total leT → transitive leT →
  ∀ (p : pred T) (s : seq T), filter p (sort T leT s) = sort T leT (filter p s).

Lemma filter_sort_in sort T (P : pred T) (leT : rel T) :
  {in P &, total leT} → {in P & &, transitive leT} →
  ∀ (p : pred T) (s : seq T),
  all P s → filter p (sort T leT s) = sort T leT (filter p s).
```

Proof. This follows mainly from Lemmas A.37, A.64, A.67, B.2 and B.6. See the proof of Theorem 3.16 for details. □

Lemma B.15. *For any total preorder $\leq \subseteq T \times T$, predicate $p \subseteq T$, and s of type* list T*,*

$$\texttt{filter}_p\,(\texttt{sort}_\leq s) = \texttt{filter}_p\, s$$

holds whenever $\texttt{filter}_p\, s$ *sorted w.r.t. $\leq$.*

```
Lemma sorted_filter_sort sort T (leT : rel T) :
  total leT → transitive leT →
  ∀ (p : pred T) (s : list T),
  sorted leT (filter p s) → filter p (sort _ leT s) = filter p s.

Lemma sorted_filter_sort_in sort T (P : {pred T}) (leT : rel T) :
  {in P &, total leT} → {in P & &, transitive leT} →
  ∀ (p : pred T) (s : list T),
  all P s → sorted leT (filter p s) → filter p (sort _ leT s) = filter p s.
```

Proof. This follows from Lemma B.4 and Theorem B.14. □

Lemma B.16. *For any total preorders $\leq_1, \leq_2 \subseteq T \times T$, sorting a list with $\leq_2$ and then $\leq_1$ gives the same result as sorting the same list with the lexicographic order $\leq_{\text{lex}} \coloneqq \leq_{(1,2)}$; that is, the following equation holds for any s of type* list T*:*

$$\mathtt{sort}_{\leq_1} (\mathtt{sort}_{\leq_2}\, s) = \mathtt{sort}_{\leq_{\text{lex}}}\, s.$$

```
Lemma sort_sort sort T (leT leT' : rel T) :
  total leT → transitive leT → total leT' → transitive leT' →
  ∀ s : list T, sort T leT (sort T leT' s) = sort T (lexord leT leT') s.

Lemma sort_sort_in sort T (P : pred T) (leT leT' : rel T) :
  {in P &, total leT} → {in P & &, transitive leT} →
  {in P &, total leT'} → {in P & &, transitive leT'} →
  ∀ s : list T,
  all P s → sort T leT (sort T leT' s) = sort T (lexord leT leT') s.
```

Proof. As in the proof of Theorem B.14, we replace s everywhere with $\mathtt{map}_{\mathtt{nth}\, x_0\, s}\, [0, \dots, |s| - 1]$ (Lemma A.67), use the naturality of `sort` (Lemma B.2), and apply the congruence rule with respect to `map`. It remains to prove

$$\mathtt{sort}_{\leq'_1} (\mathtt{sort}_{\leq'_2}\, [0, \dots, |s| - 1]) = \mathtt{sort}_{\leq'_{\text{lex}}}\, [0, \dots, |s| - 1]$$

where $\leq'_1$, $\leq'_2$, and $\leq'_{\text{lex}}$ are the preimages of $\leq_1$, $\leq_2$, and $\leq_{\text{lex}}$ under $\mathtt{nth}\, x_0\, s$, respectively. Thanks to Lemma B.6, each side of the above equation is respectively sorted w.r.t.

- the (right-associative) lexicographic composition of $\leq'_1$, $\leq'_2$, and $<_{\mathbb{N}}$, and
- the lexicographic composition of $\leq'_{\text{lex}}$ and $<_{\mathbb{N}}$, which is by definition equal to the (left-associative) lexicographic composition of $\leq'_1$, $\leq'_2$, and $<_{\mathbb{N}}$

since

- $\leq'_1$, $\leq'_2$, and their lexicographic composition $\leq'_{\text{lex}}$ are total (Lemma A.9),
- $\leq'_2$, $<_{\mathbb{N}}$, and their lexicographic composition are transitive (Lemma A.9), and
- $[0, \dots, |s| - 1]$ is sorted w.r.t. $<_{\mathbb{N}}$ (Lemma A.69).

Two lexicographic compositions of $\leq'_1$, $\leq'_2$, and $<_{\mathbb{N}}$ are equivalent regardless of the associativity (Lemma A.10), and transitive (Lemma A.9) and irreflexive. Both sides of the equation have the same set of elements (Lemma B.12). Therefore, these two lists are equal (Lemma A.64). □

Lemma B.17. *For any $\leq \subseteq T \times T$ and s of type* list T, $\mathtt{sort}_{\leq}\, s$ *is sorted w.r.t. $\leq$ whenever $\leq$ is total.*

```
Lemma sort_sorted sort T (leT : rel T) :
  total leT → ∀ s : list T, sorted leT (sort T leT s).

Lemma sort_sorted_in sort T (P : pred T) (leT : rel T) :
  {in P &, total leT} →
  ∀ s : list T, all P s → sorted leT (sort T leT s).
```

Proof. Since s is sorted w.r.t. the trivial relation R such that $x \ R \ y$ holds for any $x, y \in T$, Corollary 3.12 implies that $\mathtt{sort}_{\leq} s$ is sorted w.r.t. the lexicographic order of $\leq$ and R, which is equivalent to $\leq$. □

Lemma B.18. *For any total order $\leq \subseteq T \times T$, and s_1 and s_2 of type list T, $\mathtt{sort}_{\leq} s_1 = \mathtt{sort}_{\leq} s_2$ holds iff s_1 is a permutation of s_2.*

```
Lemma perm_sortP sort (T : eqType) (leT : rel T) :
  total leT → transitive leT → antisymmetric leT →
  ∀ s1 s2 : list T,
  reflect (sort T leT s1 = sort T leT s2) (perm_eq s1 s2).

Lemma perm_sort_inP sort (T : eqType) (leT : rel T) (s1 s2 : list T) :
  {in s1 &, total leT} → {in s1 & &, transitive leT} →
  {in s1 &, antisymmetric leT} →
  reflect (sort T leT s1 = sort T leT s2) (perm_eq s1 s2).
```

Proof. Since $\mathtt{sort}_{\leq} s_1$ and $\mathtt{sort}_{\leq} s_2$ are respectively permutations of s_1 and s_2, $\mathtt{sort}_{\leq} s_1 =_{\mathtt{perm}} \mathtt{sort}_{\leq} s_2$ holds iff $s_1 =_{\mathtt{perm}} s_2$ holds. Therefore, ($\Rightarrow$) is trivial.

($\Leftarrow$) $\mathtt{sort}_{\leq} s_1$ and $\mathtt{sort}_{\leq} s_2$ are sorted w.r.t. $\leq$ (Lemma B.17) and permutation of each other. Thanks to Lemma A.63, these two sorted lists are equal. □

Lemma B.19 (Nipkow et al. [2025, Theorem 2.9 (Uniqueness of sorting)]). *For any two stable sort functions* sort *and* sort′ *and total preorder $\leq \subseteq T \times T$, $\mathtt{sort}_{\leq}$ and $\mathtt{sort}'_{\leq}$ are extensionally equal; that is, $\mathtt{sort}_{\leq} s = \mathtt{sort}'_{\leq} s$ holds for any s of type* list T.

```
Lemma eq_sort sort1 sort2 T (leT : rel T) :
  total leT → transitive leT → sort1 T leT =1 sort2 T leT.

Lemma eq_in_sort sort1 sort2 T (P : pred T) (leT : rel T) :
  {in P &, total leT} → {in P & &, transitive leT} →
  ∀ s : list T, all P s → sort1 T leT s = sort2 T leT s.
```

Proof. We replace s everywhere with $\mathtt{map}_{\mathtt{nth}\, x_0\, s} [0, \dots, |s| - 1]$ (Lemma A.67), use the naturality of sort and sort′ (Lemma B.2), and apply the congruence rule with respect to map. It remains to prove

$$\mathtt{sort}_{\leq_I} [0, \dots, |s| - 1] = \mathtt{sort}'_{\leq_I} [0, \dots, |s| - 1]$$

where $x \leq_I y \coloneqq \mathtt{nth}\, x_0\, s\, x \leq \mathtt{nth}\, x_0\, s\, y$.

Since $[0, \dots, |s| - 1]$ is sorted w.r.t. $<_{\mathbb{N}}$ (Lemma A.69), both sides of the above equation are sorted w.r.t. $<_I$, the lexicographic composition of $\leq_I$ and $<_{\mathbb{N}}$ (Lemma B.6). These two lists have the same set of elements (Lemma B.12). Therefore, they are equal (Lemma A.64). □

Lemma B.20. *For any total preorder* $\leq \subseteq T \times T$ *and* s *of type* list T, $\texttt{sort}_\leq\ s = \texttt{isort}_\leq\ s$ *holds for the insertion sort* `isort` *defined as follows.*

$$\begin{aligned} \texttt{isort}_\leq\ [] &\coloneqq [] \\ \texttt{isort}_\leq\ (x :: s) &\coloneqq [x] \mathbin{⩕}_\leq \texttt{isort}_\leq\ s. \end{aligned}$$

```
Lemma eq_sort_insert sort T (leT : rel T) : total leT → transitive leT →
  sort T leT =1 foldr (fun x : T ⇒ merge leT [:: x]) [::].

Lemma eq_in_sort_insert sort T (P : pred T) (leT : rel T) :
  {in P &, total leT} → {in P & &, transitive leT} →
  ∀ s : list T, all P s →
  sort T leT s = foldr (fun x : T ⇒ merge leT [:: x]) [::] s.
```

Proof. This follows from Lemma B.19 and the fact that `isort` satisfies the characteristic property (Lemma 3.2). □

Lemma B.21. *For any total preorder* $\leq \subseteq T \times T$, *and* s_1 *and* s_2 *of type* list T, $\texttt{sort}_\leq\ (s_1 \mathbin{+\!\!+} s_2) = \texttt{sort}_\leq\ s_1 \mathbin{⩕}_\leq \texttt{sort}_\leq\ s_2$ *holds.*

```
Lemma sort_cat sort T (leT : rel T) : total leT → transitive leT →
  ∀ s1 s2 : seq T,
  sort T leT (s1 ++ s2) = merge leT (sort T leT s1) (sort T leT s2).

Lemma sort_cat_in sort T (P : pred T) (leT : rel T) :
  {in P &, total leT} → {in P & &, transitive leT} →
  ∀ s1 s2 : seq T, all P s1 → all P s2 →
  sort T leT (s1 ++ s2) = merge leT (sort T leT s1) (sort T leT s2).
```

Proof. We replace all the occurrences of `sort` with the insertion sort `isort` (Lemma B.20), and prove the following equation by induction on s_1:

$$\texttt{isort}_\leq\ (s_1 \mathbin{+\!\!+} s_2) = \texttt{isort}_\leq\ s_1 \mathbin{⩕}_\leq \texttt{isort}_\leq\ s_2.$$

If $s_1 = []$, the equation holds by definition. Otherwise, $s_1 = x :: s_1'$ and

$$\begin{aligned} \texttt{isort}_\leq\ (s_1 \mathbin{+\!\!+} s_2) &= [x] \mathbin{⩕}_\leq \texttt{isort}_\leq\ (s_1' \mathbin{+\!\!+} s_2) && \text{(Definition)} \\ &= [x] \mathbin{⩕}_\leq (\texttt{isort}_\leq\ s_1' \mathbin{⩕}_\leq \texttt{isort}_\leq\ s_2) && \text{(I.H.)} \\ &= ([x] \mathbin{⩕}_\leq \texttt{isort}_\leq\ s_1') \mathbin{⩕}_\leq \texttt{isort}_\leq\ s_2 && \text{(Lemma A.75)} \\ &= \texttt{isort}_\leq\ s_1 \mathbin{⩕}_\leq \texttt{isort}_\leq\ s_2. && \text{(Definition)} \end{aligned}$$ □

Lemma B.22. *For any total preorder* $\leq \subseteq T \times T$, s *of type* list T, *and* m *of type* list bool, *there exists* m' *of type* list bool *that satisfies* $\texttt{mask}_{m'}\ (\texttt{sort}_\leq\ s) = \texttt{sort}_\leq\ (\texttt{mask}_m\ s)$.

```
Lemma mask_sort sort T (leT : rel T) : total leT → transitive leT →
  ∀ (s : list T) (m : list bool),
  {m_s : list bool | mask m_s (sort T leT s) = sort T leT (mask m s)}.

Lemma mask_sort_in sort T (P : pred T) (leT : rel T) :
  {in P &, total leT} → {in P & &, transitive leT} →
  ∀ (s : list T) (m : list bool), all P s →
  {m_s : list bool | mask m_s (sort T leT s) = sort T leT (mask m s)}.
```

Proof. We replace s everywhere with $\mathtt{map}_{\mathtt{nth}\, x_0\, s}\ [0, \ldots, |s| - 1]$ (Lemma A.67), use the naturality of sort (Lemma B.2) and mask (Lemma A.40), and apply the congruence rule with respect to map. It suffices to find m' such that

$$\mathtt{mask}_{m'}\ (\mathtt{sort}_{\leq_I}\ [0, \ldots, |s| - 1]) = \mathtt{sort}_{\leq_I}\ (\mathtt{mask}_m\ [0, \ldots, |s| - 1])$$

where $i \leq_I j \coloneqq \mathtt{nth}\, x_0\, s\, i \leq \mathtt{nth}\, x_0\, s\, j$. We show that $m' \coloneqq \mathtt{map}_p\ (\mathtt{sort}_{\leq_I}\ [0, \ldots, |s| - 1])$ where $p\, i \coloneqq i \in \mathtt{mask}_m\ [0, \ldots, |s| - 1]$ satisfy the above equation:

$$\begin{aligned}
& \mathtt{mask}_{m'}\ (\mathtt{sort}_{\leq_I}\ [0, \ldots, |s| - 1]) \\
={}& \mathtt{filter}_p\ (\mathtt{sort}_{\leq_I}\ [0, \ldots, |s| - 1]) && \text{(Lemma A.41)} \\
={}& \mathtt{sort}_{\leq_I}\ (\mathtt{filter}_p\ [0, \ldots, |s| - 1]) && \text{(Theorem B.14)} \\
={}& \mathtt{sort}_{\leq_I}\ (\mathtt{mask}_m\ [0, \ldots, |s| - 1]). && \text{(Lemmas A.42 and A.68)}
\end{aligned}$$

□

Lemma B.23. *For any total preorder $\leq \subseteq T \times T$, s of type* list T, *and m of type* list bool, *there exists m' of type* list bool *that satisfies* $\mathtt{mask}_{m'}\ (\mathtt{sort}_{\leq}\ s) = \mathtt{mask}_m\ s$, *whenever* $\mathtt{mask}_m\ s$ *is sorted w.r.t.* $\leq$.

```
Lemma sorted_mask_sort sort T (leT : rel T) : total leT → transitive leT →
  ∀ (s : list T) (m : list bool), sorted leT (mask m s) →
  {m_s : list bool | mask m_s (sort T leT s) = mask m s}.

Lemma sorted_mask_sort_in sort T (P : pred T) (leT : rel T) :
  {in P &, total leT} → {in P & &, transitive leT} →
  ∀ (s : list T) (m : list bool), all P s → sorted leT (mask m s) →
  {m_s : list bool | mask m_s (sort T leT s) = mask m s}.
```

Proof. This follows from Lemmas B.4 and B.22. □

Lemma B.24. *For any total preorder $\leq \subseteq T \times T$,* $\mathtt{sort}_{\leq}$ *preserves the subsequence relation; that is, for any t and s of type* list T, $\mathtt{sort}_{\leq}\ t$ *is a subsequence of* $\mathtt{sort}_{\leq}\ s$ *whenever t is a subsequence of s.*

```
Lemma subseq_sort sort (T : eqType) (leT : rel T) :
  total leT → transitive leT →
  ∀ t s : list T, subseq t s → subseq (sort T leT t) (sort T leT s).

Lemma subseq_sort_in sort (T : eqType) (leT : rel T) (t s : list T) :
  {in s &, total leT} → {in s & &, transitive leT} →
  subseq t s → subseq (sort T leT t) (sort T leT s).
```

Proof. Using Lemma A.44 that t is a subsequence of s iff there exists m of the same size as s such that $t = \mathtt{mask}_m\ s$, we are left to show that $\mathtt{sort}_{\leq}\ (\mathtt{mask}_m\ s)$ is a subsequence of $\mathtt{sort}_{\leq}\ s$. Now, by Lemma B.22, there is a mask m' such that $\mathtt{sort}_{\leq}\ (\mathtt{mask}_m\ s) = \mathtt{mask}_{m'}\ (\mathtt{sort}_{\leq}\ s)$ which is indeed a subsequence of $\mathtt{sort}_{\leq}\ s$ (Lemma A.45). □

Lemma B.25. *For any total preorder $\leq \subseteq T \times T$, and t and s of type* list T, *t is a subsequence of* $\mathtt{sort}_{\leq}\ s$ *whenever t is a subsequence of s and sorted w.r.t.* $\leq$.

```
Lemma sorted_subseq_sort sort (T : eqType) (leT : rel T) :
  total leT → transitive leT →
  ∀ t s : list T, subseq t s → sorted leT t → subseq t (sort T leT s).

Lemma sorted_subseq_sort_in sort (T : eqType) (leT : rel T) (t s : list T) :
  {in s &, total leT} → {in s & &, transitive leT} →
  subseq t s → sorted leT t → subseq t (sort T leT s).
```

Proof. This follows from Lemmas B.4 and B.24. □

Lemma B.26. *For any total preorder $\leq \subseteq T \times T$, s of type* list T*, and x and y of type T, x and y occur in order in* $\texttt{sort}_{\leq}\ s$ *whenever $x \leq y$ holds and x and y occur in order in s.*

```
Lemma mem2_sort sort (T : eqType) (leT : rel T) :
  total leT → transitive leT →
  ∀ (s : list T) (x y : T),
  leT x y → mem2 s x y → mem2 (sort T leT s) x y.

Lemma mem2_sort_in sort (T : eqType) (leT : rel T) (s : list T) :
  {in s &, total leT} → {in s & &, transitive leT} →
  ∀ x y : T, leT x y → mem2 s x y → mem2 (sort T leT s) x y.
```

Proof. If $x = y$, the statement means that all the elements of s are in $\texttt{sort}_{\leq}\ s$, which is a consequence of Lemma B.12. Otherwise we remark that x and y occur in order in s iff the sequence $[x, y]$ is a subsequence of s (Lemma A.48). Using Lemma B.24, we thus have that $\texttt{sort}_{\leq}\ [x, y]$ is a subsequence of $\texttt{sort}_{\leq}\ s$. Since $x \leq y$, we have that $\texttt{sort}_{\leq}\ [x, y] = [x, y]$ (*e.g.*, by computation through Lemma B.20), hence $[x, y]$ a subsequence of $\texttt{sort}_{\leq}\ s$. □

C Comparison of formulations of the stability

In this appendix, we compare our stability results (Theorems B.5 and B.14 and Lemmas B.6 and B.15) with the standard formulation (Corollary 3.17) and its versions formally proved in related work [Leino and Lucio 2015; Leroy [n. d.]; Sternagel 2013].

Leroy [[n. d.]] proved the stability of a mergesort function in the following form.

Lemma C.1 (Leroy [[n. d.]], Corollary 3.17). *For any total preorder $\leq$ on T, the equivalent elements always appear in the same order in the input and output of sorting; that is, the following equation holds for any x of type T and s of type* list T*:*

$$[y \leftarrow \texttt{sort}_{\leq}\ s \mid x \equiv y] = [y \leftarrow s \mid x \equiv y].$$

```
Lemma sort_stable_leroy sort T (leT : rel T) :
  total leT → transitive leT →
  ∀ (x : T) (s : list T),
    [seq y ← sort T leT s | leT x y && leT y x] =
      [seq y ← s | leT x y && leT y x].
```

Proof. This follows from Theorem B.14 or Lemma B.15. See Corollary 3.17. □

Leino and Lucio [2015]; Sternagel [2013] proved the stability of GHC's mergesort in the following form, where the total preorder on T is decomposed into a totally ordered type T' and a keying function k of type $T \to T'$.

Lemma C.2 (Leino and Lucio [2015]; Sternagel [2013]). *Let $k : T \to T'$ be a keying function whose codomain T' is totally ordered by $\leq_{T'}$, and $\leq_T \subseteq T \times T$ be the total preorder induced by k,* i.e., *$x \leq_T y \coloneqq k\,x \leq_{T'} k\,y$. For any s of type* list T*, the relative order of the elements of s having the same key is preserved by $\texttt{sort}_{\leq_T}\ s$; that is, the following equation holds for any x of type T:*

$$[y \leftarrow \texttt{sort}_{\leq_T}\ s \mid k\,x = k\,y] = [y \leftarrow s \mid k\,x = k\,y].$$

```
Lemma sort_stable_sternagel
    sort T (d : unit) (T' : orderType d) (key : T → T') (x : T) (s : list T) :
  [seq y ← sort T (relpre key ≤%O) s | key x == key y] =
    [seq y ← s | key x == key y].
```

where `orderType` *is the interface of totally ordered types and* ≤%O *is the order attached to a totally ordered type.*

Proof. Since $\leq_{T'}$ is total order hence antisymmetric, x and y of type T have the same key ($k\,x = k\,y$) iff they have the same order ($x \leq_T y \land y \leq_T x$). Since $\leq_T$ is total preorder, this follows from Lemma C.1. □

As we argued in Section 3.4.3, both Lemmas C.1 and C.2 are easy consequences of Theorem B.14 or Lemma B.15, which follow from Theorem B.5 or Lemma B.6. We formally proved its converse: Lemma C.1 implies Theorems B.5 and B.14 and Lemma B.6, assuming some extra conditions (mostly on `sort`) detailed below. However, their proofs are not as straightforward as the other direction, and Theorem B.5 and Lemma B.6 proved in this way (Lemmas C.3 and C.4 below) require the extra condition that $\leq_1$ is transitive.

Lemma C.3 (Lemma C.1 implies Theorem B.5). *Let* `sort` *be a function of type* $\forall T, (T \to T \to \mathrm{bool}) \to \mathrm{list}\,T \to \mathrm{list}\,T$ *such that*

- $\texttt{sort}_{\leq}\ [] = []$ *for any* $\leq$,
- $\texttt{sort}_{\leq}\ s$ *is sorted w.r.t.* $\leq$ *whenever* $\leq$ *is total, and*
- $[y \leftarrow \texttt{sort}_{\leq}\ s \mid x \equiv y] = [y \leftarrow s \mid x \equiv y]$ *for any type* T, *total preorder* $\leq$ *on* T, s *of type* $\mathrm{list}\,T$, *and* $x \in T$.

Then, for any type T, *relations* $\leq_1, \leq_2\ \subseteq\ T \times T$, *and* s *of type* $\mathrm{list}\,T$, $\texttt{sort}_{\leq_1}\ s$ *is sorted w.r.t. the lexicographic order* $\leq_{(1,2)}$ *whenever* $\leq_1$ *is total preorder and* s *is pairwise sorted w.r.t.* $\leq_2$.

Lemma C.4 (Lemma C.1 implies Lemma B.6). *Let* `sort` *be a function of type* $\forall T, (T \to T \to \mathrm{bool}) \to \mathrm{list}\,T \to \mathrm{list}\,T$ *satisfying the same conditions as Lemma C.3. Then, for any type* T, *relations* $\leq_1, \leq_2 \subseteq T \times T$, *and* s *of type* $\mathrm{list}\,T$, $\texttt{sort}_{\leq_1}\ s$ *is sorted w.r.t. the lexicographic order* $\leq_{(1,2)}$ *whenever* $\leq_1$ *is total preorder,* $\leq_2$ *is transitive, and* s *is sorted w.r.t.* $\leq_2$.

Lemma C.5 (Lemma C.1 implies Theorem B.14). *Let* `sort` *be a function of type* $\forall T, (T \to T \to \mathrm{bool}) \to \mathrm{list}\,T \to \mathrm{list}\,T$ *that satisfies the following conditions in addition to ones in Lemma C.3:*

- $\texttt{sort}_{\leq}\ s$ *has the same set of elements as* s *for any* $\leq\ \subseteq T \times T$ *and* s *of type* $\mathrm{list}\,T$, *and*
- `sort` *is natural, that is,* $\texttt{sort}_{\leq_T}\ [f\,x \mid x \leftarrow s] = [f\,x \mid x \leftarrow \texttt{sort}_{\leq_{T'}}\ s]$ *for any types* T *and* T', *function* f *from* T' *to* T, *relation* $\leq_T\ \subseteq T \times T$, *list* s *of type* $\mathrm{list}\,T'$, *where* $x \leq_{T'} y \coloneqq f\,x \leq_T f\,y$,

Then, for any type T, *total preorder* $\leq\ \subseteq T \times T$, *and predicate* $p \subseteq T$, $\texttt{filter}_p$ *commutes with* $\texttt{sort}_{\leq}$ *under function composition; that is, the following equation holds for any* s *of type* $\mathrm{list}\,T$*:*

$$\texttt{filter}_p\,(\texttt{sort}_{\leq}\ s) = \texttt{sort}_{\leq}\,(\texttt{filter}_p\ s).$$

D Definitions of mergesorts in Rocq

In Section 5.1, we presented structurally-recursive non-tail-recursive and tail-recursive mergesorts in OCaml. In this appendix, we present their Rocq reimplementations, including some optimized ones such as smooth variants (Section 4.2). The formal correctness proofs of these implementations are omitted in this appendix and available only in the supplementary material.

```
Module Type MergeSig.
Parameter merge : ∀ (T : Type) (leT : rel T), list T → list T → list T.
Parameter mergeE : ∀ (T : Type) (leT : rel T), merge leT =2 path.merge leT.
End MergeSig.

Module Merge <: MergeSig.

Fixpoint merge (T : Type) (leT : rel T) (xs ys : list T) : list T ≔
  if xs is x :: xs' then
    (fix merge' (ys : list T) : list T ≔
       if ys is y :: ys' then
         if leT x y then x :: merge leT xs' ys else y :: merge' ys'
       else xs) ys
  else ys.

Lemma mergeE (T : Type) (leT : rel T) : merge leT =2 path.merge leT.

End Merge.
```

Fig. 8. Structurally-recursive non-tail-recursive merge in Rocq, *i.e.*, the Rocq counterpart of merge in Figure 5.

D.1 Non-tail-recursive mergesorts in Rocq

In Figure 8, we define a module type [The Rocq Development Team 2025c] `MergeSig` abstracting out a non-tail-recursive merge function, and then, define one of its instances `Merge` using the nested fixpoint technique (Section 5.1.1). The purpose of this module type is to allow replacement of the implementation of merge function. Therefore, the non-tail-recursive mergesort functions in Figure 9 are provided as a functor module `CBN_` that takes an instance `M` of `MergeSig` as a parameter. In order to prove the mergesort functions in the `CBN_` module correct, `MergeSig` and `Merge` are equipped with the proof `mergeE` that the merge function implemented in the module is extensionally equal to the one in the MathComp library (`path.merge` in Figure 8, or Definition A.70).

Inside the `CBN_` module, we introduce a type `T` and a relation `leT` on `T` as section variables [The Rocq Development Team 2025b]. After closing the `CBN` section, these variables become inaccessible and declarations in the section will be parameterized by them. The functions `push`, `pop`, `sort1rec`, and `sort1` inside this module correspond to `push`, `pop`, `sort_rec`, and `sort` in Figure 5, respectively.

The `sort2` function is an optimized variants of `sort1` that perform sorting by repetitively taking taking two elements from the input, locally sorting them, and pushing them to the stack. Similarly, the `sort3` function in Figure 10 takes three elements from the input at a time which is analogous to the optimization technique employed by `List.stable_sort` of OCaml (see the end of Section 4.1).

The `sortN` function in Figure 11 is a smooth structurally-recursive non-tail-recursive mergesort that takes the longest sorted prefix from the input instead of a fixed number of elements. The `sortNrec` function determines whether the prefix is increasing or decreasing one, and calls either `incr` or `decr` depending on that. The `incr` and `decr` respectively takes the longest weakly increasing and strictly decreasing slice from the input, push it to the stack, and call `sortNrec` to process the next sorted slice. These three functions are made mutually recursive to pass the termination checker of Rocq.

```
Module CBN_ (M : MergeSig).
Section CBN.
Context (T : Type) (leT : rel T).

Fixpoint push (xs : list T) (stack : list (list T)) {struct stack} :
    list (list T) :=
  match stack with
  | [::] :: stack | [::] as stack => xs :: stack
  | ys :: stack => [::] :: push (M.merge leT ys xs) stack
  end.

Fixpoint pop (xs : list T) (stack : list (list T)) {struct stack} : list T :=
  if stack is ys :: stack then pop (M.merge leT ys xs) stack else xs.

Fixpoint sort1rec (stack : list (list T)) (xs : list T) {struct xs} : list T :=
  if xs is x :: xs then sort1rec (push [:: x] stack) xs else pop [::] stack.

Definition sort1 : list T -> list T := sort1rec [::].

Fixpoint sort2rec (stack : list (list T)) (xs : list T) {struct xs} : list T :=
  if xs is x1 :: x2 :: xs then
    let t := if leT x1 x2 then [:: x1; x2] else [:: x2; x1] in
    sort2rec (push t stack) xs
  else pop xs stack.

Definition sort2 : list T -> list T := sort2rec [::].

[..]

End CBN.
End CBN_.
```

Fig. 9. Structurally-recursive non-tail-recursive mergesorts in Rocq. The push, pop, sort1rec, and sort1 functions in this figure are the Rocq counterparts of push, pop, sort_rec, and sort in Figure 5. Some optimized implementations are omitted here as marked by [..] and shown in Figures 10 and 11.

D.2 Tail-recursive mergesorts in Rocq

Similarly to the non-tail-recursive counterpart (Figures 8 and 9 in Appendix D.1), we define a module type RevmergeSig and its instance Revmerge implementing a tail-recursive merge function (Figure 12), and the tail-recursive mergesort functions are provided as a functor module CBV_ that takes an instance M of RevmergeSig as a parameter (Figure 13).

The sort2 and sort3 functions in Figures 13 and 14 take two and three elements from the input at a time, respectively. Again, the latter is analogous to the optimization technique employed by List.stable_sort of OCaml (Section 4.1). The sortN function in Figure 15 is a smooth structurally-recursive tail-recursive mergesort, implemented in the same way as the non-tail-recursive counterpart (Figure 11).

We stress that the recursive functions defined and used in Figures 12 to 15 are tail recursive except for push in Figure 13, whose depth of recursive calls is logarithmic in the length of the input.

```
Fixpoint sort3rec (stack : list (list T)) (xs : list T) {struct xs} : list T ≔
  match xs with
  | x1 :: x2 :: x3 :: xs ⇒
    let t ≔
        if leT x1 x2 then
          if leT x2 x3 then [:: x1; x2; x3]
          else if leT x1 x3 then [:: x1; x3; x2] else [:: x3; x1; x2]
        else
          if leT x1 x3 then [:: x2; x1; x3]
          else if leT x2 x3 then [:: x2; x3; x1] else [:: x3; x2; x1]
    in
    sort3rec (push t stack) xs
  | [:: x1; x2] ⇒ pop (if leT x1 x2 then xs else [:: x2; x1]) stack
  | _ ⇒ pop xs stack
  end.

Definition sort3 : list T → list T ≔ sort3rec [::].
```

Fig. 10. A structurally-recursive non-tail-recursive mergesorts in Rocq, that takes three elements from the input at a time. This implementation is omitted in the place marked by [..] in Figure 9.

```
Fixpoint sortNrec (stack : list (list T)) (x : T) (xs : list T) {struct xs} :
    list T ≔
  if xs is y :: xs then
    if leT x y then incr stack y xs [:: x] else decr stack y xs [:: x]
  else
    pop [:: x] stack
with incr (stack : list (list T)) (x : T) (xs accu : list T) {struct xs} :
    list T ≔
  if xs is y :: xs then
    if leT x y then
      incr stack y xs (x :: accu)
    else
      sortNrec (push (catrev accu [:: x]) stack) y xs
  else
    pop (catrev accu [:: x]) stack
with decr (stack : list (list T)) (x : T) (xs accu : list T) {struct xs} :
    list T ≔
  if xs is y :: xs then
    if leT x y then
      sortNrec (push (x :: accu) stack) y xs
    else
      decr stack y xs (x :: accu)
  else
    pop (x :: accu) stack.

Definition sortN (xs : list T) : list T ≔
  if xs is x :: xs then sortNrec [::] x xs else [::].
```

Fig. 11. A smooth structurally-recursive non-tail-recursive mergesort in Rocq. This implementation is omitted in the place marked by [..] in Figure 9.

```
Module Type RevmergeSig.
Parameter revmerge :
  ∀ (T : Type) (leT : rel T), list T → list T → list T.
Parameter revmergeE : ∀ (T : Type) (leT : rel T) (xs ys : list T),
    revmerge leT xs ys = rev (path.merge leT xs ys).
End RevmergeSig.

Module Revmerge <: RevmergeSig.

Fixpoint merge_rec (T : Type) (leT : rel T) (xs ys accu : list T) {struct xs} :
    list T :=
  if xs is x :: xs' then
    (fix merge_rec' (ys accu : list T) {struct ys} :=
       if ys is y :: ys' then
         if leT x y then
           merge_rec leT xs' ys (x :: accu) else merge_rec' ys' (y :: accu)
       else
         catrev xs accu) ys accu
  else catrev ys accu.

Definition revmerge (T : Type) (leT : rel T) (xs ys : list T) : list T :=
  merge_rec leT xs ys [::].

Lemma revmergeE (T : Type) (leT : rel T) (xs ys : list T) :
  revmerge leT xs ys = rev (path.merge leT xs ys).

End Revmerge.
```

Fig. 12. Structurally-recursive tail-recursive merge in Rocq. The `merge_rec` function in this figure is the Rocq counterpart of `revmerge` in Figure 7, and `revmerge` in this figure instantiate it with `[]` as the accumulator.

Therefore, a linear consumption of stack space does not occur in any of the mergesort functions presented in this section.

```
Module CBV_ (M : RevmergeSig).
Section CBV.
Context (T : Type) (leT : rel T).
Let geT x y ≔ leT y x.

Fixpoint push (xs : list T) (stack : list (list T)) {struct stack} :
    list (list T) ≔
  match stack with
  | [::] :: stack | [::] as stack ⇒ xs :: stack
  | ys :: [::] :: stack | ys :: ([::] as stack) ⇒
    [::] :: M.revmerge leT ys xs :: stack
  | ys :: zs :: stack ⇒
    [::] :: [::] :: push (M.revmerge geT (M.revmerge leT ys xs) zs) stack
  end.

Fixpoint pop (mode : bool) (xs : list T) (stack : list (list T)) {struct stack}
    : list T ≔
  match stack, mode with
  | [::], true ⇒ rev xs
  | [::], false ⇒ xs
  | [::] :: [::] :: stack, _ ⇒ pop mode xs stack
  | [::] :: stack, _ ⇒ pop (~~ mode) (rev xs) stack
  | ys :: stack, true ⇒ pop false (M.revmerge geT xs ys) stack
  | ys :: stack, false ⇒ pop true (M.revmerge leT ys xs) stack
  end.

Fixpoint sort1rec (stack : list (list T)) (xs : list T) {struct xs} : list T ≔
  if xs is x :: xs then
    sort1rec (push [:: x] stack) xs
  else
    pop false [::] stack.

Definition sort1 : list T → list T ≔ sort1rec [::].

Fixpoint sort2rec (stack : list (list T)) (xs : list T) {struct xs} : list T ≔
  if xs is x1 :: x2 :: xs then
    let t ≔ if leT x1 x2 then [:: x1; x2] else [:: x2; x1] in
    sort2rec (push t stack) xs
  else pop false xs stack.

Definition sort2 : list T → list T ≔ sort2rec [::].

[..]

End CBV.
End CBV_.
```

Fig. 13. Structurally-recursive tail-recursive mergesorts in Rocq. The push, pop, sort1rec, and sort1 functions in this figure are the Rocq counterparts of push, pop, sort_rec, and sort in Figure 7. Some optimized implementations are omitted here as marked by [..] and shown in Figures 14 and 15.

```
Fixpoint sort3rec (stack : list (list T)) (xs : list T) {struct xs} : list T ≔
  match xs with
  | x1 :: x2 :: x3 :: xs ⇒
    let t ≔
        if leT x1 x2 then
          if leT x2 x3 then [:: x1; x2; x3]
          else if leT x1 x3 then [:: x1; x3; x2] else [:: x3; x1; x2]
        else
          if leT x1 x3 then [:: x2; x1; x3]
          else if leT x2 x3 then [:: x2; x3; x1] else [:: x3; x2; x1]
    in
    sort3rec (push t stack) xs
  | [:: x1; x2] ⇒ pop false (if leT x1 x2 then xs else [:: x2; x1]) stack
  | _ ⇒ pop false xs stack
  end.

Definition sort3 : list T → list T ≔ sort3rec [::].
```

Fig. 14. A structurally-recursive tail-recursive mergesorts in Rocq, that takes three elements from the input at a time. This implementation is omitted in the place marked by [..] in Figure 13.

```
Fixpoint sortNrec (stack : list (list T)) (x : T) (xs : list T) {struct xs} :
    list T ≔
  if xs is y :: xs then
    if leT x y then incr stack y xs [:: x] else decr stack y xs [:: x]
  else
    pop false [:: x] stack
with incr (stack : list (list T)) (x : T) (xs accu : list T) {struct xs} :
    list T ≔
  if xs is y :: xs then
    if leT x y then
      incr stack y xs (x :: accu)
    else
      sortNrec (push (catrev accu [:: x]) stack) y xs
  else
    pop false (catrev accu [:: x]) stack
with decr (stack : list (list T)) (x : T) (xs accu : list T) {struct xs} :
    list T ≔
  if xs is y :: xs then
    if leT x y then
      sortNrec (push (x :: accu) stack) y xs
    else
      decr stack y xs (x :: accu)
  else
    pop false (x :: accu) stack.

Definition sortN (xs : list T) : list T ≔
  if xs is x :: xs then sortNrec [::] x xs else [::].
```

Fig. 15. A smooth structurally-recursive tail-recursive mergesort in Rocq. This implementation is omitted in the place marked by [..] in Figure 13.