# A bargain for mergesorts (functional pearl)

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

CYRIL COHEN, Université Côte d'Azur, Inria, France

KAZUHIKO SAKAGUCHI, Inria, France

We present a novel characterization of stable mergesort functions using relational parametricity, and show that it implies the 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 characterization property for each variation. To further motivate this work, we show a performance trade-off between tail-recursive and non-tail-recursive mergesorts that (1) the former in call-by-value evaluation avoids using up stack space and is efficient and (2) the latter in call-by-need evaluation is an optimal incremental sort, meaning that it performs only $O(n + k \log k)$ comparisons to compute the least (or greatest) $k$ items of a list of length $n$. 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 Coq proof assistant.

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

Additional Key Words and Phrases:  Interactive theorem proving, Relational parametricity, Mergesort, Stable sort, Evaluation strategies

## 1 INTRODUCTION

Sorting is one of the most well-studied problems in computer science. 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 functional 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 Coq [Appel 2023][Chlipala 2013, Section 7.1] and Isabelle [Nipkow et al. 2024; Sternagel 2013]. While there are some exceptions (Section 6), most such case studies do not prove the stability and consider only one implementation without any optimization or heuristic, because proving the correctness and stability of several sort functions with distinct sets of intricate optimization techniques is a complex and laborious task. In particular, a methodology to share a large part of the functional correctness proofs between several variants of a sort algorithm has not been studied to the best of our our knowledge. 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, we should be able to turn any stable mergesort function into the identity function. In other words, if it permutes 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. Formally, we characterize the mergesort

functions by the existence of such a corresponding abstract mergesort function satisfying the following properties (Sections 3.2 and 4.4):

- the binary relational parametricity [Reynolds 1983],
- the abstract mergesort function instantiated with merge is the mergesort function, and
- the abstract mergesort function instantiated with concatenation is the identity function,

Our correctness proof of mergesort is twofold. Firstly, we prove that each mergesort function we wish to verify satisfies the above properties (Sections 3.3, 4.4.1 and 4.4.2). Nonetheless, the actual work that has to be carried out by hand is the proof of the third item, because the first item should follow from an abstraction theorem [Reynolds 1983], and the second item should hold by definition at least for sufficiently simple implementations. Secondly, we deduce several correctness results solely from the characterization property (Sections 3.4 and 4.4.3). Our characterization 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 3.3 and 4.1), and the naturality of sort (Lemma 3.10). These two consequences of our characterization are sufficient to deduce several correctness results of mergesort, including stability.

To further motivate this work, we review some optimization techniques for mergesort (Section 4). In particular, we show that the optimal mergesort function differs depending on whether one wants to compute only a first few elements of the output incrementally by taking advantage of call-by-need evaluation or not; in other words, there are at least two kinds of optimized mergesort functions that cannot substitute each other for performance reasons. If one wants to compute the entire output in call-by-value evaluation, the optimal implementation is tail-recursive mergesort, *e.g.*, List.stable_sort of the OCaml [Leroy et al. 2022] standard library, which avoids using up stack space and thus is efficient (Section 4.1). If one wants to compute only a first few elements of the output by taking advantage of call-by-need evaluation, the optimal implementation is non-tail-recursive mergesort, *e.g.*, bottom-up non-tail-recursive mergesort in Section 3.1 and Data.List.sort of the GHC (Glasgow Haskell Compiler) [The GHC Team 2021] libraries. The former algorithm does not allow for incremental computation regardless of the evaluation strategy. Contrarily, the latter is slower than the former and crashes on longer inputs in the call-by-value evaluation. The advantage of our verification approach here is that it allows us to verify both implementations of mergesort modularly. In addition, we prove that computing the least $k$ items of a list of length $n$ costs $O(n + k \log k)$ time in the latter case, which is asymptotically optimal [Paredes and Navarro 2006] (Section 4.2). To put it another way, we show that laziness achieves the same asymptotics as the optimal algorithms known in the context of the so-called *incremental sorting* problem without making the process of saving and resuming computations explicit. There is another optimization technique called *smooth mergesorts* that take advantage of sorted slices in the input, which can be combined with any bottom-up mergesort (Section 4.3).

We formalized our functional correctness proofs in the Coq proof assistant [The Coq Development Team 2024a]. In Section 5, we discuss two technical aspects of our formalization. We first review a technique [Gonthier 2009] to convince Coq that bottom-up non-tail-recursive mergesort is terminating (Section 5.1.2), which does not require explicit termination proofs (the use of well-founded recursion), but is done by making the mergesort function structurally recursive [Giménez 1994] without the use of fuel. Avoiding the use of well-founded recursion has two practical advantages: (1) it makes execution of mergesort inside the Coq kernel easier and more efficient, and (2) it makes application of the parametricity translation [Bernardy et al. 2012] of the ParamCoq plugin [Keller and Lasson 2012; Keller et al. 2014] to the abstract mergesort function straightforward. Furthermore, this technique makes the balanced binary tree construction of mergesort tail-recursive (Section 5.1.2), thus allows us to implement bottom-up tail-recursive mergesort (Section 5.1.3), and

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 the given implementation, ✎ means that the classification can apply but such an implementation is not given in this paper, and ✗ means that the classification does not apply.

|  | top-down | bottom-up | non-tail-rec. | tail-rec. | smooth |
|---|---|---|---|---|---|
| Section 3.1 | ✓ | ✓ | ✓ | ✗ | - |
| Section 4.1 | ✓ | ✗ | ✗ | ✓ | ✗ |
| Section 4.2 | ✎ | ✓ | ✓ | ✗ | ✎ |
| Section 4.3 | ✗ | ✓ | ✓ | ✎ | ✓ |
| Section 5.1.2 | ✗ | ✓ | ✓ | ✗ | ✎ |
| Section 5.1.3 | ✗ | ✓ | ✗ | ✓ | ✎ |
| OCaml (`List.stable_sort`) | ✓ | ✗ | ✗ | ✓ | ✗ |
| GHC (`Data.List.sort`) | ✗ | ✓ | ✓ | ✗ | ✓ |
| Nipkow et al. [2024] | ✓ | ✓ | ✓ | ✗ | ✓ |
| Appel [2023]; Chlipala [2013] | ✓ | ✗ | ✓ | ✗ | ✗ |
| Leroy [[n. d.]] | ✗ | ✓ | ✓ | ✗ | ✗ |

both of these bottom-up non-tail-recursive and tail-recursive mergesorts can be easily made smooth in contrast to top-down mergesorts. We second 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). Constructing an instance of this interface for a concrete mergesort function is mostly automatic: the binary relational parametricity can be automatically proved using the Paramcoq plugin, and the equation about the instantiation with merge can often be proved by definition (Section 5.2.2). We stress that our functional correctness proofs in Coq are axiom-free thanks to the Paramcoq plugin, although the discussion regarding the abstraction theorem in Section 3 is briefly done.

To summarize, Table 1 classifies the mergesort functions and their optimization techniques presented in this paper and related work. Note that non-tail-recursive mergesort presented in Section 4.2 cannot be made top-down and smooth at the same time, and Leino and Lucio [2015]; Sternagel [2013] verified slightly modified versions of GHC's `Data.List.sort`.

The paper is organized as follows. 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 non-tail-recursive mergesort whose underlying traces have different shapes (Section 3.1), and describe how to prove the characterization property on them (Section 3.3). In Section 4, we review optimization techniques such as tail-recursive mergesort (Section 4.1), non-tail-recursive mergesort as optimal incremental sort (Section 4.2), and smooth mergesort (Section 4.3), and extend the characterization and proof technique presented in Section 3 to tail-recursive and smooth mergesorts (Section 4.4). In Section 5, we show how to make mergesort structurally recursive (Section 5.1) and discuss the design and organization of the library (Section 5.2). Section 6 concludes the paper.

## 2 PRELIMINARIES

The examples of mergesort functions are presented in the OCaml syntax in this paper. However, since we have to take different evaluation strategies into account in Section 4, we do not fix the

evaluation strategy to interpret our examples. As a reference, some functions in the `List` module[1] to manipulate lists are listed below.

```
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 Coq, including the Coq 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:

$$\mathsf{map}_f := \mathsf{map}\ \mathsf{f}, \qquad\qquad [f\ x \mid x \leftarrow xs] := \mathsf{map}_f\ xs,$$
$$\mathsf{filter}_p := \mathsf{filter}\ \mathsf{p}, \qquad\qquad [x \mid x \leftarrow xs, p\ x] := \mathsf{filter}_p\ xs,$$
$$xs \mathbin{+\mkern-8mu+} ys := \mathsf{append}\ xs\ ys, \qquad\qquad [f\ x \mid x \leftarrow xs, p\ x] := \mathsf{map}_f\ (\mathsf{filter}_p\ xs).$$

Throughout the paper, we use the comparison model to discuss the computational complexities of our examples. That is to say, we postulate that a comparison of two items is done in constant space and time, and no other operation on the items is allowed. For the sake of simplicity and without loss of generality, we fix the comparison function to "less than or equal" and think only about the case of sorting items in ascending order.

## 3  NON-TAIL-RECURSIVE MERGESORTS

In this section, we present a simplified version of the characterization (Section 3.2) and the correctness proofs (Sections 3.3 and 3.4) that work only for non-tail-recursive mergesort. The proofs are twofold. Firstly, we prove that a mergesort function satisfies the characterization property, which has to be done for each mergesort function (Section 3.3). Secondly, we deduce several correctness results solely from the characterization property, which can be done generically for any mergesort function (Section 3.4). We will later extend our approach (Section 4.4) to support tail-recursive

---

[1]See Leroy et al. [2022, Chapter 25] for the OCaml standard library. Note that `List.split_n` is not available in the standard library: see https://ocaml.org/p/core/v0.16.2/doc/Core/List/index.html#val-split_n instead for example.

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

|                  (a) Non-tail-recursive merge.                  |                  (b) Tail-recursive merge.                  |

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

```
let rec sort (<=) = function         let sort (<=) xs =
  | [] -> []                           let rec merge_pairs = function
  | [x] -> [x]                           | a :: b :: xs ->
  | xs ->                                  merge (<=) a b :: merge_pairs xs
    let k = length xs / 2 in             | xs -> xs in
    let (xs1, xs2) = split_n xs k in   let rec merge_all = function
    merge (<=)                           | [] -> []
      (sort (<=) xs1)                    | [x] -> x
      (sort (<=) xs2)                    | xs -> merge_all (merge_pairs xs)
                                       in
                                       merge_all (map (fun x -> [x]) xs)
```

|                  (a) Top-down mergesort.                  |                  (b) Bottom-up mergesort.                  |

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

mergesorts (Section 4.1) and smooth mergesorts (Section 4.3). As variations of mergesort covered by this section, we consider top-down and bottom-up non-tail-recursive mergesorts (Section 3.1).

## 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 by 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. Since the input is not necessarily a concatenation of two sorted lists, mergesort has to recursively apply the merge function to sort the entire input. There are broadly two approaches to perform such recursion: *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 both cases, any two lists being merged are of the same length if the length of the input is a power of two, but otherwise, they have to merge two lists of different lengths at some point. The difference of the lengths of two lists being merged is at most one in the top-down mergesort, and contrary, the length of the former list is always a power of two in the bottom-up mergesort. This

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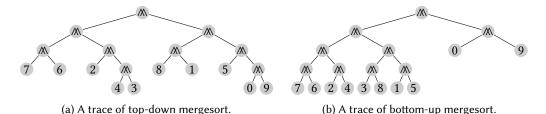
Fig. 3. The traces of the sorting processes for the input list [7; 6; 2; 4; 3; 8; 1; 5; 0; 9] in the naive top-down (a) and bottom-up (b) mergesort algorithms. Each number placed as a leaf denotes a singleton list consisting of it, and ⋀ placed as an internal node denotes the merging of its children.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

difference of associativity of merging 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 is always an AVL tree in the top-down mergesort, but it is relatively unbalanced in the bottom-up mergesort. Regardless of the approach, the height of any trace must be logarithmic in the length of the input $n$ 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

In this section, we present the characterization of stable non-tail-recursive mergesort functions. A sort algorithm is *stable* if equivalent elements, such that $x$ and $y$ satisfying $x \le y \le x$ where $\le$ is the order given to the sort algorithm, 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. Taking this observation into account, the intuition behind the characterization is that we should be able to turn the mergesort function into the identity function just 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:

$$\forall (T : \mathcal{U}), \quad \underbrace{(T \to T \to \text{bool})}_{\text{comparison function}} \to \qquad\qquad \underbrace{\text{list } T}_{\text{input}} \to \underbrace{\text{list } T}_{\text{output}}$$

to asort of type:

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

where $\mathcal{U}$ is a universe (type of types). 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 $⋏_\leq$ ($:=$ merge ($\leq$)):

$$\forall (T : \mathcal{U}) \ (\leq \ : T \to T \to \text{bool}) \ (xs : \text{list } T), \text{asort} \ (⋏_\leq) \ [\cdot] \ [] \ xs = \text{sort}_\leq \ xs \tag{1}$$

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

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

Although both Equations (1) and (2) instantiate the type parameter $R$ of the abstract sort function asort with 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.*, the following (binary) relational parametricity [Reynolds 1983] holds:

$\forall (T_1 \ T_2 : \mathcal{U}) \ (\sim_T \ \subseteq T_1 \times T_2),$

$\forall (R_1 \ R_2 : \mathcal{U}) \ (\sim_R \ \subseteq R_1 \times R_2),$

$\forall (\text{merge}_1 : R_1 \to R_1 \to R_1) \ (\text{merge}_2 : R_2 \to R_2 \to R_2),$ (merge)

$\quad (\forall (xs_1 : R_1) \ (xs_2 : R_2), xs_1 \sim_R xs_2 \to$

$\quad \ \forall (ys_1 : R_1) \ (ys_2 : R_2), ys_1 \sim_R ys_2 \to (\text{merge}_1 \ xs_1 \ ys_1) \sim_R (\text{merge}_2 \ xs_2 \ ys_2)) \to$

$\forall ([\cdot]_1 : T_1 \to R_1) \ ([\cdot]_2 : T_2 \to R_2),$ (singleton)

$\quad (\forall (x_1 : T_1) \ (x_2 : T_2), x_1 \sim_T x_2 \to [x_1]_1 \sim_R [x_2]_2) \to$

$\forall ([]_1 : R_1) \ ([]_2 : R_2), []_1 \sim_R []_2 \to$ (empty)

$\forall (xs_1 : \text{list } T_1) \ (xs_2 : \text{list } T_2), (xs_1, xs_2) \in [\![\text{list}]\!]_{\sim_T} \to$

$\quad (\text{asort merge}_1 \ [\cdot]_1 \ []_1 \ xs_1) \sim_R (\text{asort merge}_2 \ [\cdot]_2 \ []_2 \ xs_2)$

where $[\![\text{list}]\!]_\sim$ relates two lists $xs := [x_1, \ldots, x_n]$ and $ys := [y_1, \ldots, y_m]$ iff they have the same length $n = m$ and each corresponding pair of their elements is in relation with the other, *i.e.*, $x_i \sim y_i$ for any $1 \leq i \leq n$.

To sum up, we characterize stable mergesort functions by the existence of an abstract mergesort function asort that satisfies Equations (1) and (2) and relational parametricity above.

## 3.3 Proofs of the characterization property

In this section, we informally and briefly show that the two mergesort functions presented in Figure 2 satisfy the characterization property presented in Section 3.2. Figure 4 shows two abstract mergesort functions asort derived from the mergesort functions sort in Figure 2. For each variation, Equation (1) should hold by definition, *i.e.*, asort ($⋏_\leq$) $[\cdot]$ $[]$ is the same as sort$_\leq$ modulo renaming. The relational parametricity should follow from an abstraction theorem [Reynolds 1983]. Therefore, here we only prove Equation (2), *i.e.*, asort ($+\!\!+$) $[\cdot]$ $[]$ is the identity function, for each variation. The Coq counterpart of this section is Section 5.2.2.

LEMMA 3.1 (EQUATION (2) FOR THE TOP-DOWN MERGESORT IN FIGURE 2A). asort *in Figure 4a satisfies* asort ($+\!\!+$) $[\cdot]$ $[]$ $xs = xs$ *for any xs.*

PROOF. Let $\phi$ 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

```
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
```

```
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)
```

(a) Top-down abstract mergesort.                                    (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.

$0 < k < n$ 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 3.2 (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

$$\text{flatten (merge\_pairs } xs) = \text{flatten } xs \tag{3}$$

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

$$
\begin{aligned}
\text{flatten (merge\_pairs } xs) &= \text{flatten } ((a + b) :: \text{merge\_pairs } xs') &&\text{(Definition Figure 4b)} \\
&= (a + b) + \text{flatten (merge\_pairs } xs') &&\text{(Definition A.15)} \\
&= (a + b) + \text{flatten } xs' &&\text{(I.H.)} \\
&= a + (b + \text{flatten } xs') &&\text{(Lemma A.13)} \\
&= \text{flatten } xs. &&\text{(Definition A.15)}
\end{aligned}
$$

Second, we prove

$$\text{merge\_all } xs = \text{flatten } xs \tag{4}$$

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,

$$\text{merge\_all } xs = \text{merge\_all (merge\_pairs } xs) \qquad \text{(Definition Figure 4b)}$$

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

$$
\begin{aligned}
&= \text{flatten (merge\_pairs } xs) &&\text{(I.H.)} \\
&= \text{flatten } xs. &&\text{(Equation (3))}
\end{aligned}
$$

Finally, we show

$$\texttt{asort}\ (\texttt{+\!\!+})\ [\cdot]\ []\ xs = \texttt{merge\_all}\ [[x]\mid x \leftarrow xs] \qquad\qquad \text{(Definition Figure 4b)}$$
$$= \texttt{flatten}\ [[x]\mid x \leftarrow xs] \qquad\qquad\qquad \text{(Equation (4))}$$
$$= xs. \qquad\qquad\qquad\qquad\qquad \text{(Structural induction on } xs) \qquad \square$$

## 3.4 Correctness proofs

In this section, we assume that a function $\texttt{sort}$ satisfies the characterization of stable mergesort functions (Section 3.2), and deduce several correctness results of $\texttt{sort}$ solely from that. The use of the relational parametricity of the abstract sorting function $\texttt{asort}$ in our correctness proofs is twofold: deducing an induction principle over traces (Section 3.4.1), and deducing the naturality of $\texttt{sort}$ (Section 3.4.2).

### 3.4.1 Induction over traces.

LEMMA 3.3 (AN INDUCTION PRINCIPLE OVER TRACES OF $\texttt{sort}$). *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

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

We apply the relational parametricity of $\texttt{asort}$ where $\sim_T$ and $\sim_R$ are instantiated with the equality over $T$ and $\sim$. The premise $(xs_1, xs_2) \in [\![\text{list}]\!]_{\sim_T}$ holds because both $xs_1$ and $xs_2$ are instantiated with $xs$ and $[\![\text{list}]\!]_{\sim_T}$ is just the equality over list $T$. The other three premises exactly correspond to the three induction cases. $\qquad\square$

As applications of the induction principle above, we prove the permutation property (Lemma 3.4 and Corollary 3.5) and a stability result (Lemma 3.7) of $\texttt{sort}$ below.

LEMMA 3.4. *For any relation $\leq$ on type $T$ and $xs$ of type* list $T$, *$\texttt{sort}_\leq xs =_{\text{perm}} xs$ holds; that is, $\texttt{sort}_\leq xs$ is a permutation of $xs$.*

PROOF. We prove it by induction on $\texttt{sort}_\leq xs$ (Lemma 3.3). The last two cases are obvious since $=_{\text{perm}}$ is reflexive. Suffice it to show that $xs' \wedge\!\!\wedge_\leq ys' =_{\text{perm}} xs \!+\!\!+ ys$ holds whenever $xs' =_{\text{perm}} xs$ and $ys' =_{\text{perm}} ys$ hold (which follows from Lemmas A.54 and A.72). $\qquad\square$

COROLLARY 3.5. *For any relation $\leq$ on type $T$ and $xs$ of type* list $T$, *$\texttt{sort}_\leq xs$ has the same set of elements as $xs$, i.e., $x \in \texttt{sort}_\leq xs$ iff $x \in xs$ for any $x \in T$.*

We define two versions of sortedness of lists to state the stability in a general form.

*Definition 3.6 (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* by $R$ 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* by $R$ 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 3.7 (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,* $\text{sort}_{\leq_1} xs$ *is sorted by the following lexicographic order:*

$$x \leq_{\text{lex}} y := x \leq_1 y \wedge (y \nleq_1 x \vee x \leq_2 y)$$

*whenever $xs$ is pairwise sorted by $\leq_2$.*

PROOF. We prove a generalized proposition:

$$(\text{sort}_{\leq_1} xs \subseteq xs) \wedge (xs \text{ is pairwise sorted by } \leq_2 \rightarrow \text{sort}_{\leq_1} xs \text{ is sorted by } \leq_{\text{lex}})$$

by induction on $\text{sort}_{\leq_1} xs$ (Lemma 3.3). 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

$$(xs' \subseteq xs) \wedge (ys' \subseteq ys) \rightarrow (xs' \bbA_{\leq_1} ys' \subseteq xs + ys)$$

which is obvious since $xs' \bbA_{\leq_1} ys'$ is a permutation of $xs' + ys'$ (Lemma A.72).

For the second component of the conjunction, suffice it to show that $xs' \bbA_{\leq_1} ys'$ is sorted by $\leq_{\text{lex}}$ whenever:

  (i)  $xs'$ and $ys'$ are subsets of $xs$ and $ys$, respectively,
  (ii)  $xs'$ (resp. $ys'$) is sorted by $\leq_{\text{lex}}$ if $xs$ (resp. $ys$) is pairwise sorted by $\leq_2$, and
  (iii)  $xs + ys$ is pairwise sorted by $\leq_2$.

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

  (iv)  both $xs$ and $ys$ are pairwise sorted by $\leq_2$, 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 by $\leq_{\text{lex}}$. 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 $xs' \bbA_{\leq_1} ys'$ is sorted by $\leq_{\text{lex}}$ (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 3.8. *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,* $\text{sort}_{\leq_1} xs$ *is sorted by the lexicographic order $\leq_{\text{lex}}$ of $\leq_1$ and $\leq_2$ whenever $xs$ is sorted by $\leq_2$.*

*3.4.2 Naturality.* The induction principle over traces (Lemma 3.3) does not allow us to relate two sorting processes that behave parametrically. Instead, we obtain the parametricity (Lemma 3.9) and the naturality (Lemma 3.10) of sort from the parametricity of asort.

LEMMA 3.9 (THE PARAMETRICITY OF sort). *Any* sort *function satisfying the characterization property satisfies the following relational parametricity:*

$\forall (T_1\ T_2 : \mathcal{U})\ (\sim_T \subseteq T_1 \times T_2),$

$\forall (\leq_1 : T_1 \rightarrow T_1 \rightarrow \text{bool})\ (\leq_2 : T_2 \rightarrow T_2 \rightarrow \text{bool}),$

$(\forall (x_1 : T_1)\ (x_2 : T_2), x_1 \sim_T x_2 \rightarrow \forall (y_1 : T_1)\ (y_2 : T_2), y_1 \sim_T y_2 \rightarrow (x_1 \leq_1 y_1) = (x_2 \leq_2 y_2)) \rightarrow$

$\forall (xs_1 : \text{list } T_1)\ (xs_2 : \text{list } T_2), (xs_1, xs_2) \in [\![\text{list}]\!]_{\sim_T} \rightarrow$

$(\text{sort}_{\leq_1} xs_1, \text{sort}_{\leq_2} xs_2) \in [\![\text{list}]\!]_{\sim_T}.$

PROOF. $\text{sort}_{\leq_i}$ is extensionally equal to asort $(\bbA_{\leq_i})\ (\lambda(x : T), [x])\ []$ for each $i \in \{1, 2\}$ thanks to the characterization. Since asort and its arguments here respect parametricity, sort respects parametricity as well. □

LEMMA 3.10 (THE NATURALITY OF sort [WADLER 1989, SECTION 3.3]). *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:*

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

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

PROOF. We instantiate Lemma 3.9 with $T_1 \coloneqq T$, $T_2 \coloneqq T'$, $x \sim_T y \coloneqq (x = f\ y)$, $\leq_1 \coloneqq \leq_T$, and $\leq_2 \coloneqq \leq_{T'}$. Then, the first premise is equivalent to the definition of $\leq_{T'}$. The second premise $(xs_1, xs_2) \in [\![\text{list}]\!]_{\sim_T}$ is equivalent to $xs_1 = [f\ x \mid x \leftarrow xs_2]$. By substituting this equation to the conclusion, we get $\text{sort}_{\leq_T} [f\ x \mid x \leftarrow xs_2] = [f\ x \mid x \leftarrow \text{sort}_{\leq_{T'}} xs_2]$.                                    □

LEMMA 3.11. *For any total preorder $\leq$ on $T$ and predicate $p$ on $T$, $\text{filter}_p$ commutes with $\text{sort}_{\leq}$ under function composition; that is, the following equation holds for any $xs$ of type* list $T$:

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

PROOF. We will rely on the fact that two lists are equal whenever they are sorted by 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):

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

where $\text{nth}\ x_0\ xs\ i$ is the $i^{\text{th}}$ element in the list $xs$ and its default value $x_0$ is an arbitrary element from the list $xs$ (Definition A.66). 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

$$i <_I j \coloneqq i \leq_I j \wedge (j \nleq_I i \vee i <_{\mathbb{N}} j).$$

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

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

Using the naturality of $\text{sort}$ (Lemma 3.10) and $\text{filter}$ (Lemma A.37), *i.e.*, $\text{map}_f (\text{filter}_{p \circ f} xs) = \text{filter}_p (\text{map}_f xs)$, it remains to prove

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

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

Now, we apply the congruence rule with respect to $\text{map}$ and the fact mentioned in the beginning of this proof (Lemma A.64) by checking that both sides of the above equation are sorted by $<_I$ and they contain the same set of elements. The latter condition follows from Corollary 3.5 and the fact that $x \in \text{filter}_p xs$ iff $p\ x \wedge x \in xs$ for any $p$, $xs$, and $x$ (Lemma A.38). Finally, both lists are sorted by $<_I$ because $\text{sort}$ is stable (Corollary 3.8), $is \coloneqq [0, \ldots, |xs| - 1]$ is sorted by $<_{\mathbb{N}}$ (Lemma A.69), and $\text{filter}$ turns a sorted list into a sorted list whenever the relation is transitive (Lemma A.62).          □

The stability of a sort function is often, *e.g.*, H. Cormen et al. [2022]; Leino and Lucio [2015]; Leroy [[n. d.]]; Sternagel [2013], formulated as follows.

COROLLARY 3.12 (THE STANDARD FORMULATION OF THE STABILITY). *For any total preorder $\leq$ on $T$, the equivalent elements, such that $x$ and $y$ satisfying $x \sim y \coloneqq x \leq y \wedge y \leq x$, 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 $T$:

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

PROOF. The left hand side is equal to $\text{sort}_\le$ $[y \leftarrow s \mid x \sim y]$ (Lemma 3.11). We use the fact that any list pairwise sorted by $\le$ is an invariant of $\text{sort}_\le$ (Lemma B.3). All elements of $[y \leftarrow s \mid x \sim y]$ are equivalent to $x$, and thus, it is pairwise sorted by $\le$.                                                    □

As it can be seen in the above proof, Corollary 3.12 is an immediate consequence of Lemmas 3.11 and B.3. We argue Lemma 3.11 has the advantage that it can be used in a place Corollary 3.12 does not directly apply, *i.e.*, where the predicate is not one collecting equivalent elements.

## 4 OPTIMIZATIONS

In this section, we review some optimization techniques for mergesort, and see how our approach presented in Section 3 extends to these optimized implementations. In Section 4.1, we review tail-recursive mergesort which avoids using up stack space and thus is efficient in call-by-value evaluation. In Section 4.2, we revisit non-tail-recursive mergesort in call-by-need evaluation, and show that it is an optimal *incremental sorting* algorithm, *i.e.*, it can compute the least $k$ items of a list of length $n$ in $O(n + k \log k)$ time. These two kinds of mergesort functions cannot substitute each other for performance reasons, because the former does not allow for incremental computation regardless of the evaluation strategy, and the latter crashes on longer inputs in the call-by-value evaluation. The existence of such a performance trade-off is also supported by a performance evaluation (Appendix F). In Section 4.3, we review another optimization technique called *smooth mergesorts*, that take advantage of sorted slices in the input. In Section 4.4, we extend our characterization and proof technique (Section 3) to tail-recursive mergesorts (Section 4.1) and smooth mergesorts (Section 4.3).

### 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) is not tail recursive since its recursive calls appear in non-tail positions. Each recursive call of such merge function consumes some stack space, and thus, it crashes on longer inputs in call-by-value evaluation. The commonly used technique to make it tail recursive is to add an *accumulator* argument accu that is initially the empty list and then accumulate the result on accu, as in revmerge in Figure 1b.[2] The tail-recursive merge function accumulates the first elements of the input lists as the last element of the resulting list. Thus, it finally produces its output in reverse order. That is to say, the following equation holds for any binary relation (<=) and any lists xs and ys:

$$\text{revmerge (<=) xs ys []} = \text{rev (merge (<=) xs ys).}$$

Two lists sorted in descending order can be merged properly 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 by (<=):[3]

$$\text{merge (<=) (merge (<=) s1 s2) (merge (<=) s3 s4)}$$
$$= \text{revmerge (>=) (revmerge (<=) s3 s4 []) (revmerge (<=) s1 s2 []) [].}$$

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 depth of recursive calls of the top-down sort function is logarithmic in the length of the input, although it is

---

[2]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.

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

not tail recursive. Thus, there is no actual issue there. Therefore, based on the top-down approach, all the major inefficiency issues explained here can be addressed as follows.
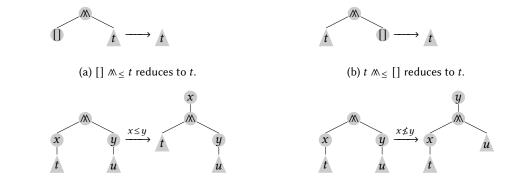
```
let sort (<=) xs =
  let (>=) = (fun x y -> y <= x) in
  let rec sort_rec xs b n =
    match n, xs with
    | 1, x :: xs' -> [x], xs'
    | _, _ ->
      let n1 = n / 2 in
      let s1, xs'  = sort_rec xs (not b) n1 in
      let s2, xs'' = sort_rec xs' (not b) (n - n1) in
      (if b then revmerge (>=) 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, which can be made smooth (Section 4.3).

## 4.2 Non-tail-recursive mergesort in call-by-need evaluation is an optimal incremental sorting algorithm

Although the tail-recursive merge function (Figure 1b) is efficient in call-by-value evaluation, it forces us to compute the entire output to obtain the first item in call-by-need evaluation. In contrast, the non-tail-recursive mergesort function (Figure 2b) can compute the output incrementally. As a result, there is a folklore fact that computing the least item of a list by (head . sort) in Haskell has a linear time complexity $O(n)$, e.g., Hinze [1997]. Besides that, the problem of obtaining the least $k$ items of the input list of length $n \geq k$ is called *partial sorting*, and its online version where $k$ is unknown to the algorithm is called *incremental sorting* [Paredes and Navarro 2006]. Some algorithms solve these problems in $O(n + k \log k)$ time, which is asymptotically optimal. In this section, we prove that the (bottom-up) non-tail-recursive mergesort algorithm (Figure 2b) in call-by-need evaluation is asymptotically optimal as an incremental sort algorithm. Note that the crucial part of our complexity analysis has also been discussed by Apfelmus [2009a,b], which concluded that (take k . sort) in Haskell costs $O(n + k \log n)$ time.

We extend the notion of trace to represent intermediate steps of the computation. Let [] and $t$ denote the empty list and an arbitrary (sub-)tree, respectively. An item to be sorted such as ①now denotes a head element of a list, and has one direct sub-tree representing its tail which can be omitted if it is empty. As a result, the reduction rules for the merge function can be seen as transformations of traces, as depicted in Figure 5. A trace is said to be in *head normal form* if it is either empty [] or a cons whose head is a value, and obtaining the head normal form of a nonempty trace $t$ is an equivalent operation to taking the least item of $t$. Obtaining the head normal form of $t \wedge_{\leq} u$ requires obtaining the head normal forms of both $t$ and $u$ and applying one of the reduction rules in Figure 5.

(a) $[] \curlywedge_{\leq} t$ reduces to $t$.

(b) $t \curlywedge_{\leq} []$ reduces to $t$.

(c) $(x :: t) \curlywedge_{\leq} (y :: u)$ reduces to $x :: (t \curlywedge_{\leq} (y :: u))$ if $x \leq y$ holds.

(d) $(x :: t) \curlywedge_{\leq} (y :: u)$ reduces to $y :: ((x :: t) \curlywedge_{\leq} u)$ if $x \leq y$ does not hold.

Fig. 5. The reduction rules for merge represented as transformations of traces.



(a) The initial state of trace to sort [5; 1; 2; 7; 0; 4; 3; 6].

(b) The trace after taking the least item 0 out from (a). Each merging has been reduced by one step.

(c) The trace after taking the second least item 1 out from (b). Each merging highlighted in (b) has been reduced by one step.

Fig. 6. An example transition of traces in call-by-need evaluation of non-tail-recursive mergesort.

The algorithm in question (Figure 2b) first divides the input into singleton lists in linear time, and then merges each adjacent pair to obtain a nested list of length $\frac{n}{2}$, then $\frac{n}{4}$, then $\frac{n}{8}$, and so on, to eventually construct the initial trace (*e.g.*, Figure 6a). This initialization phase excluding the computation of merging costs a linear time: $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots = O(n)$. After that, each merge has to be reduced by one step to obtain the least item (*e.g.*, Figure 6b). Since there is a linear number of occurrences of merge in the trace, taking the first item can be done in $O(n)$ time.

One of two direct sub-trees of each merge left after a reduction must be in head normal form according to the reduction rules of Figures 5c and 5d. If we exclude these sub-trees of head normal form from the entire trace, only one path remains (*e.g.*, as highlighted in Figure 6b). There are at most $\lceil \log_2 n \rceil$ occurrences of merge in that path, and obtaining the second least item forces the head normal form computation of them, which costs $O(\log n)$ time (*e.g.*, Figure 6c). Since the same condition will be maintained as the invariant of the trace after that (*e.g.*, as highlighted in Figure 6c), computing the second and later $k$ items costs $O(k \log n)$ time.

By summing the above two phases, we get $O(n + k \log n)$. Now, $n + k \log n \leq 2n + k \log k \Leftrightarrow \log \frac{n}{k} \leq \frac{n}{k}$, hence $n + k \log n = O(n + k \log k)$ [Paredes and Navarro 2006]. Therefore, the overall worst-case time complexity of computing the least $k$ items is $O(n + k \log k)$.

Note that the argument of this section is not specific to bottom-up mergesort, but the initialization phase has to be done in a linear time $O(n)$. For example, it does not apply to the naive top-down mergesort function of Figure 2a since its initialization phase costs a quasilinear time $O(n \log n)$, but the technique presented in Section 4.1 allows us to implement a top-down non-tail-recursive mergesort function that achieves optimal $O(n + k \log k)$ time complexity.

In Appendix E, we show that non-tail-recursive mergesort can actually be turned into a tail-recursive function in destination-passing style [Larus 1989; Minamide 1998] by the *Tail Modulo Cons* (TMC) transformation [Bour et al. 2021].

### 4.3 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 3.10). Bottom-up mergesort can easily be made smooth regardless of whether it is tail recursive or not, 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 mergesort based on bottom-up non-tail-recursive mergesort (Figure 2b) 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` (resp. `descending`) processes one weakly increasing (resp. strictly decreasing) slice.

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 work in Section 6.

### 4.4 New characterization of stable mergesort functions

In this section, we extend the characterization of stable mergesort functions presented in Section 3.2 to support tail-recursive (Section 4.1) and smooth (Section 4.3) mergesorts. We first add more

operators on $T$ and $R$ to the type of abstract sort functions asort, as follows:

$$\forall(T \; R : \mathcal{U}), \underbrace{(T \to T \to \text{bool})}_{\text{relation} \leq} \to \underbrace{(R \to R \to R)}_{\text{merge by} \leq} \to \underbrace{(R \to R \to R)}_{\text{merge by} \geq} \to$$

$$\underbrace{(T \to R)}_{\text{singleton}} \to \underbrace{R}_{\text{empty}} \to \underbrace{\text{list } T}_{\text{input}} \to \underbrace{R}_{\text{output}} .$$

The first argument of type $T \to T \to \text{bool}$ 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 (Section 4.3) 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 by $\leq$. In order to merge them with $\geq$, we introduce the following operator $\mathbb{W}$:

$$xs \; \mathbb{W}_{\leq} \; ys := \text{rev} \; (\text{rev} \; ys \; \mathbb{M}_{\geq} \; \text{rev} \; xs).$$

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

$$\forall(T : \mathcal{U}) \; (\leq \; : T \to T \to \text{bool}) \; (xs : \text{list } T), \text{asort} \; (\leq) \; (\mathbb{M}_{\leq}) \; (\mathbb{W}_{\leq}) \; [\cdot] \; [] \; xs = \text{sort}_{\leq} \; xs, \quad (5)$$

$$\forall(T : \mathcal{U}) \; (\leq \; : T \to T \to \text{bool}) \; (xs : \text{list } T), \text{asort} \; (\leq) \; (\text{+}) \; (\text{+}) \; [\cdot] \; [] \; xs = xs. \quad (6)$$

We characterize stable mergesort functions by the existence of an abstract mergesort function asort that satisfies Equations (5) and (6) and the relational parametricity of asort derived from the type above, which we omit for brevity.

In the following subsections, we describe how the new characterization applies to the tail-recursive mergesort (Section 4.4.1) and smooth mergesort (Section 4.4.2), and how the correctness proofs presented in Section 3.4 can be adapted to the new characterization (Section 4.4.3).

*4.4.1 Tail-recursive mergesort.* The abstract mergesort function for the tail-recursive mergesort function (Section 4.1) can be obtained by simply abstracting out the tail-recursive merge function with (<=) and (>=) to the two abstract merge functions, respectively, 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))
```

Note that the order of the arguments of merge by (>=) alternates between Section 4.1 and asort above for the same reason as Footnote 3, and that sorted lists that appear in execution of asort instantiated as in Equation (5) are always increasing while they are a mix of increasing and decreasing lists in the tail-recursive mergesort presented in Section 4.1. Therefore, the proof of Equation (5) should now be done by induction involving some equational reasoning about merge and reversal of lists, rather than just by definition.

*4.4.2 Smooth mergesort.* The major obstacle in defining the abstract mergesort function for the smooth mergesort (Section 4.3) 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 reproduce this behavior with merge accu (singleton a). Similarly, we can reproduce 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). As in Section 4.4.1, the proof of Equation (5) cannot be done just by definition.

*4.4.3 Correctness proofs.* The induction principle over traces (Lemma 3.3) can be adapted to the new characterization 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, $xs \sim$ sort$_\leq xs$ holds whenever the following four induction cases hold:*

- *for any lists $xs$, $xs'$, $ys$, and $ys'$ of type* list $T$, $(xs \mathbin{+\!\!+} ys) \sim (xs' \mathbin{\wedge\!\!\wedge}_\leq ys')$ *holds whenever $xs \sim xs'$ and $ys \sim ys'$ hold,*
- *for any lists $xs$, $xs'$, $ys$, and $ys'$ of type* list $T$, $(xs \mathbin{+\!\!+} ys) \sim$ rev (rev $ys' \mathbin{\wedge\!\!\wedge}_\geq$ 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 Lemmas 3.4 and 3.7 can be easily adapted to the new induction principle by adding the corresponding case, as follows.

EXTENSION TO THE PROOF OF LEMMA 3.4. We prove sort$_\leq xs =_{\mathrm{perm}} xs$ by induction on sort$_\leq xs$ (Lemma 4.1). Suffice it to show that rev (rev $ys' \mathbin{\wedge\!\!\wedge}_\geq$ rev $xs'$) $=_{\mathrm{perm}} xs \mathbin{+\!\!+} ys$ holds whenever $xs' =_{\mathrm{perm}} xs$ and $ys' =_{\mathrm{perm}} ys$ hold (which follows from Lemmas A.54 to A.56 and A.72). The other induction cases are done in the first proof of Lemma 3.4.                                                                □

EXTENSION TO THE PROOF OF LEMMA 3.7. We prove a generalized proposition:

$$(\mathrm{sort}_{\leq_1} xs \subseteq xs) \wedge (xs \text{ is pairwise sorted by } \leq_2 \rightarrow \mathrm{sort}_{\leq_1} xs \text{ is sorted by } \leq_{\mathrm{lex}})$$

by induction on sort$_{\leq_1} xs$ (Lemma 4.1), where $x \leq_{\mathrm{lex}} y := x \leq_1 y \wedge (y \not\leq_1 x \vee x \leq_2 y)$.
    For the first component of the conjunction, suffice it to show

$$(xs' \subseteq xs) \wedge (ys' \subseteq ys) \rightarrow (\mathrm{rev}\,(\mathrm{rev}\,ys' \mathbin{\wedge\!\!\wedge}_{\geq_1} \mathrm{rev}\,xs') \subseteq xs \mathbin{+\!\!+} ys)$$

which is obvious since rev (rev $ys' \mathbin{\wedge\!\!\wedge}_{\geq_1}$ rev $xs'$) is a permutation of $xs' \mathbin{+\!\!+} ys'$.
    For the second component of the conjunction, suffice it to show that rev (rev $ys' \mathbin{\wedge\!\!\wedge}_{\geq_1}$ rev $xs'$) is sorted by $\leq_{\mathrm{lex}}$, or equivalently, its reversal rev $ys' \mathbin{\wedge\!\!\wedge}_{\geq_1}$ rev $xs'$ is sorted by the converse of $\leq_{\mathrm{lex}}$ (Lemma A.61):

$$x \geq_{\mathrm{lex}} y := x \geq_1 y \wedge (y \not\geq_1 x \vee x \geq_2 y),$$

whenever:

- (i) $xs'$ and $ys'$ are subsets of $xs$ and $ys$, respectively,
- (ii) $xs'$ (resp. $ys'$) is sorted by $\leq_{\mathrm{lex}}$ if $xs$ (resp. $ys$) is pairwise sorted by $\leq_2$, and
- (iii) $xs \mathbin{+\!\!+} ys$ is pairwise sorted by $\leq_2$.

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

- (iv) both $xs$ and $ys$ are pairwise sorted by $\leq_2$, 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 by $\leq_{\text{lex}}$, or equivalently, rev $xs'$ and rev $ys'$ are sorted by $\geq_{\text{lex}}$ (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 $ys'$ $\wedge\!\!\wedge_{\geq_1}$ rev $xs'$ is sorted by $\geq_{\text{lex}}$ (Lemma A.74).

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

The naturality of sort (Lemma 3.10), 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 Coq, remains the same.

## 5 FORMALIZATION IN COQ

In this section, we discuss two technical aspects of our formalization of mergesort functions and their correctness proofs in Coq. We first review a technique [Gonthier 2009] to make bottom-up mergesorts structurally recursive, so that their termination becomes trivial for Coq (Section 5.1). Furthermore, this technique makes the balanced binary tree construction of 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 Coq implementations of structurally-recursive mergesort functions, including some optimized implementations such as smooth variants (Section 4.3). Appendix F evaluates the performance of OCaml and Haskell code extracted [Letouzey 2004] from our Coq implementations of mergesort, as well as ones in the standard libraries, *i.e.*, List.stable_sort of OCaml and Data.List.sort of Haskell.

### 5.1 Structurally-recursive bottom-up mergesorts

*5.1.1 The syntactic guard condition.* A fixpoint function $f$ in Coq [The Coq Development Team 2024d] has the form of (fix $f_{\text{rec}}$ ($\vec{x} : \vec{A}$) {struct $x_k$} : $B$ := $M$) where $f_{\text{rec}}$ is the local name of the fixpoint function bound in $M$, ($\vec{x} : \vec{A}$) is the list of arguments, $x_k$ is the $k^{\text{th}}$ 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 $f_{\text{rec}}$ in $M$ must be *guarded by destructors* [Giménez 1994] in Coq. That is to say, $M$ must do recursive calls to $f_{\text{rec}}$ only on strict subterms of $x_k$. In practice, the annotation of decreasing argument {struct $x_k$} may be left implicit, and the Coq system 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 7. 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

```
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. 7. Structurally-recursive non-tail-recursive merge and mergesort in OCaml.

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 manage pending mergings using an explicit stack [Gonthier 2009]. We represent the stack of pending mergings as a list of sorted lists whose head ($0^{th}$) 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 7. 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 $n^{th}$ 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 $xs_0 \mathbin{\wedge\!\!\!\wedge}_{\le} xs$[4] to $S'$. In terms of trace, this procedure can be seen as a technique to construct balanced binary trees, and the $n^{th}$ element of the stack corresponds to a perfect binary tree of height $n$, as in Figure 8.

---

[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.
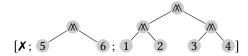
[]

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

[ 1 ]

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

$[ \textbf{✗} \, ; \, 1 \qquad \qquad 2 \, ]$

(c) The stack after pushing 2 to (b). The new item 2 has been merged with the top element 1 to obtain $1 \mathbb{M} 2$.

$[ \, 3 \, ; \, 1 \qquad \qquad 2 \, ]$

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

$[ \textbf{✗} \, ; \, \textbf{✗} \, ; \, 1 \qquad 2 \qquad 3 \qquad 4 \, ]$

(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 $1 \mathbb{M} 2$, to obtain $(1 \mathbb{M} 2) \mathbb{M} (3 \mathbb{M} 4)$.

$[ \, 5 \, ; \, \textbf{✗} \, ; \, 1 \qquad 2 \qquad 3 \qquad 4 \, ]$

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

$[ \textbf{✗} \, ; \, 5 \qquad \qquad 6 \, ; \, 1 \qquad 2 \qquad 3 \qquad 4 \, ]$

(g) The stack after pushing 6 to (f). The new item 6 has been merged with the top element 5 to obtain $5 \mathbb{M} 6$.

$[ \, 7 \, ; \, 5 \qquad \qquad 6 \, ; \, 1 \qquad 2 \qquad 3 \qquad 4 \, ]$

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

Fig. 8. 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 $xs_0, \ldots, xs_n$ 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 $xs_n \mathbb{M} (\cdots \mathbb{M} (xs_0 \mathbb{M} xs) \ldots)$.

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 7, respectively. The above mergesort function, 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]. This algorithm 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 the stack contains a merging of level $n$ or not by testing the $n^{\text{th}}$ 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.

Non-tail-recursive mergesort presented in this section can be made smooth (Section 4.3) by pushing sorted slices to the stack instead of singleton lists. See Appendix D.1 for its Coq implementation.

Since the abstract mergesort function as defined in Section 4.4 cannot test the emptiness of sorted lists, we have to use the option type to encode empty lists. Therefore, the type of the stack becomes list (option R), and the proof of Equation (5) cannot be done just by definition.

```
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. 9. Structurally-recursive tail-recursive merge and mergesort in OCaml.

*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 9. 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 9. 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 rev ($ys \wedge_{\leq} xs$) 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 rev ($zs \wedge_{\geq}$ rev ($ys \wedge_{\leq} xs$)) 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 9, 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 Coq implementation.

## 5.2 Interface for stable mergesorts

In this section, we present the interface for mergesort functions bundling our characterization property presented in Sections 3.2 and 4.4, so that we can state our correctness lemmas polymorphically for any stable mergesort function, and explain how to populate this interface with instances for concrete mergesort functions in practice, in Coq.

*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 the PARAMCOQ plugin [Keller and Lasson 2012; Keller et al. 2014] that implements a parametricity translation $\llbracket \cdot \rrbracket$ [Bernardy et al. 2012] from Coq terms to their relatedness expressed as Coq terms. The abstraction theorem [Reynolds 1983] for a binary parametricity translation can be stated as follows.

THEOREM 5.1 (ABSTRACTION THEOREM [BERNARDY ET AL. 2012]). *If* $\vdash t : A$, *then* $\vdash \llbracket t \rrbracket : \llbracket A \rrbracket\ t\ t$.

As it can be seen in the above statement, $\llbracket \cdot \rrbracket$ translates types to their parametricity statements (*e.g.*, Section 3.2 and Lemma 3.9, but abstract on the term *t*) and terms to their parametricity proofs. The following Parametricity command from the PARAMCOQ plugin applies the parametricity translation to the constant asort_ty, and declares its result $\llbracket$asort_ty$\rrbracket$ 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 characterization 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 binary relational parametricity of `asort`, `asort_mergeE` and `asort_catE` are respectively the two equational properties (5) and (6) 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. Proving the binary relational parametricity of `asort1` can be done just by applying the parametricity translation:

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

As we claimed in Section 3.3, the proof of Equation (5) (`asort_mergeE` above) can be done just *by definition* for naive top-down and bottom-up non-tail-recursive mergesort (Section 3.1). More precisely, it means that we can make `asort1` $(\le)$ $(\mathbb{M}_{\le})$ $(\mathbb{W}_{\le})$ $[\cdot]$ $[]$ definitionally equal to $\text{sort1}_{\le}$, and thus, the proof can be done just by reflexivity in Coq.

```
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.
Proof. reflexivity. Qed.
```

However, if the mergesort function performs an operation on sorted lists that has no definitionally equal combination of the provided operations, *e.g.*, tail-recursive merge (Sections 4.1 and 4.4.1), cons in smooth mergesorts (Sections 4.3 and 4.4.2), and emptiness test in structurally-recursive mergesorts (Section 5.1), the above lemma has to be proved by induction. In Section 6, we discuss potential ways to make Equation (5) definitionally holds for more variants of mergesort.

Equation (6) has to be proved by induction, as demonstrated in Section 3.3.

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

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 generically for any `stableSort` instance. For example, the commutation of `filter` and mergesort functions (Lemma 3.11) 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 Coq Development Team 2024e], 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` (Lemma 3.11) 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*

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

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

*This corollary corresponds to the following lemma in* COQ*:*

```
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 `filter` and `sort`, 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` (Lemma 3.11).    □

Appendix B provides the list of all lemmas and their proofs about stable sort functions we stated and proved using the `stableSort` structure.

# 6 CONCLUSION AND RELATED WORK

We characterized mergesort functions for lists using their abstract versions and the binary relational parametricity (Sections 3.2 and 4.4). 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 binary relational 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 (5) and (6)). By combining the binary relational parametricity with these two equational properties, we deduced an induction principle over traces—binary trees reflecting the underlying divide-and-conquer structure—to reason about the relation between input and output of mergesort (Lemma 3.3), and the naturality of mergesort (Lemma 3.10). These two properties were sufficient to deduce several correctness results of mergesort, including stability (Sections 3.4 and 4.4.3).

The traces we informally show in Figure 3 are reminiscent of the tree datatype appearing in the work of Hinze et al. [2013], however we do not expose the `makeTree` functions that could be obtained by calling our asort functions on Fork, Leaf and Tip, that would respectively denote the binary node, singleton leaf and empty tree of the tree datatype.

In order to verify a given mergesort function using our technique, one just has to define its abstract version and then prove its binary relational parametricity and Equations (5) and (6). Among these properties, the proof of binary relational parametricity follows from the abstraction theorem [Reynolds 1983], or formally, the parametricity translation [Bernardy et al. 2012] of the Paramcoq plugin [Keller and Lasson 2012; Keller et al. 2014], and Equation (5) holds by definition, or formally, can be proved by reflexivity, at least for sufficiently simple implementations. The rest of the correctness proofs work generically for any mergesort satisfying our characterization property. Therefore, the actual work that has to be carried out by hand is the proof of Equation (6), which justifies the title of this paper, claiming our functional correctness proofs are almost for free and at a bargain.

The proof of Equation (5) has actually not been done by reflexivity in our Coq proofs. In general, if the mergesort function in question performs operations on sorted lists other than ones provided to the abstract mergesort function, Equation (5) cannot be proved just by definition. For example, the tail-recursive merge function used in tail-recursive mergesorts is not exactly the same as merge functions provided to the abstract mergesort function (Section 4.4.1), smooth mergesorts have to construct sorted lists directly using cons (Section 4.4.2), and structurally-recursive mergesorts have to test the emptiness of sorted lists (Section 5.1). Equation (5) has to be proved by induction and some equational reasoning for these mergesort functions. Nevertheless, we might be able to mitigate the situation by supplying more operators to the abstract mergesort function. For example, we can consider splitting the abstract type representing sorted lists $R$ into two abstract types $R$ and $R'$ respectively representing sorted lists in ascending and descending orders, and supplying tail-recursive merge functions of type $R \to R \to R'$ and $R' \to R' \to R$, so that Equation (5) holds by definition for tail-recursive mergesorts. Similarly, we can consider supplying an emptiness test function, so that Equation (5) holds by definition for structurally-recursive mergesorts. Exploring such possibilities is left as future work.

Using relational 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.

As we claimed in Section 1, most existing formal correctness proofs of mergesort do not provide the stability results and consider only one implementation without any optimization or heuristic. For example, two textbooks of formal verification in Coq [Appel 2023; Chlipala 2013] verified the functional correctness, except stability, of top-down non-tail-recursive mergesort, and Leroy [[n. d.]] verified the functional correctness, including stability, of non-smooth bottom-up non-tail-recursive mergesort in Coq. The functional correctness and stability of slightly modified versions of GHC's mergesort have been proved in Isabelle/HOL [Sternagel 2013] and Dafny [Leino and Lucio 2015]. 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 they do, since mergeAll function (merge_all in Figure 2b) as is cannot be defined in Coq due to the syntactic guard condition. Nipkow et al. [2024] presented several verified sort algorithms, *e.g.*, non-smooth top-down and bottom-up non-tail-recursive mergesorts, and smooth mergesort taken from Sternagel [2013]. However, a large part of their functional correctness proofs are redone for each implementation, and thus, their proofs are not modular as ours. In contrast to the present work, none of the above related work verifies tail-recursive mergesort. The stability results by Leino and Lucio [2015]; Leroy [[n. d.]]; Sternagel [2013] are formulated as in Corollary 3.12 or its equivalent statement. These results can be deduced from Lemma 3.11, which directly applies to filter with any predicate in contrast to Corollary 3.12. We compare these statements further in Appendix C.

De Gouw et al. [2019] verified a widely-used variant of smooth mergesort for arrays called *TimSort* taken from the OpenJDK core library using the semi-automatic 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.

## REFERENCES

Heinrich Apfelmus. 2009a. Implicit Heaps. https://apfelmus.nfshost.com/articles/implicit-heaps.html Accessed: 2021-11-11.

Heinrich Apfelmus. 2009b. Quicksort and *k*-th smallest elements. https://apfelmus.nfshost.com/articles/quicksearch.html Accessed: 2021-11-11.

Andrew W. Appel. 2023. Software foundations, VOLUME 3, Verified Functional Algorithms. https://softwarefoundations.cis.upenn.edu/vfa-1.5.4/ Accessed: 2024-02-10, Version 1.5.4.

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

Frédéric Bour, Basile Clément, and Gabriel Scherer. 2021. Tail Modulo Cons. (2021). arXiv:2102.09823 [cs.PL] Accessed: 2021-12-23.

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. https://doi.org/10.1007/11617990_6

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

Nancy A. Day, John Launchbury, and Jeff Lewis. 1999. Logical Abstractions in Haskell. In *Proceedings of the 1999 Haskell Workshop.* http://www.cs.uu.nl/research/techreps/repo/CS-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. https://doi.org/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. https://doi.org/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. https://doi.org/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/

Ralf Hinze. 1997. Re: heap sort or the wonder of abstraction. https://www.mail-archive.com/haskell@haskell.org/msg01820.html Accessed: 2021-01-25.

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. https://doi.org/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. https://doi.org/10.1145/3498726

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. https://doi.org/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.

James Richard Larus. 1989. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. Ph. D. Dissertation. University of California, Berkeley. https://apps.dtic.mil/sti/citations/ADA631680

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. https://doi.org/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. 2022. The OCaml system. https://v2.ocaml.org/releases/4.14/htmlman/ Accessed: 2024-02-10, Version 4.14.

Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming : Program extraction within Coq proof assistant)*. Ph. D. Dissertation. University of Paris-Sud, Orsay, France. https://tel.archives-ouvertes.fr/tel-00150912

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

Yasuhiko Minamide. 1998. A Functional Representation of Data Structures with a Hole. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. ACM, 75–84. https://doi.org/10.1145/268946.268953

Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gomez-Londono, Peter Lammich, Christian Sternagel, Simon Wimmer, and Bohua Zhan. 2024. Functional Data Structures and Algorithms: A Proof Assistant Approach. https://functional-algorithms-verified.org/ Accessed: 2024-02-10.

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. https://doi.org/10.1137/1.9781611972863.16

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

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. https://doi.org/10.1007/s10817-012-9260-7

The Coq Development Team. 2024a. *The Coq Proof Assistant Reference Manual*. https://coq.inria.fr/doc/V8.19.0/refman/ the PDF version with numbered sections is available at https://github.com/coq/coq/releases/tag/V8.19.0.

The Coq Development Team. 2024b. *Section 2.1.11 "Sections"*. In [The Coq Development Team 2024a]. https://coq.inria.fr/doc/V8.19.0/refman/language/core/sections

The Coq Development Team. 2024c. *Section 2.1.12 "The Module System"*. In [The Coq Development Team 2024a]. https://coq.inria.fr/doc/V8.19.0/refman/language/core/modules

The Coq Development Team. 2024d. *Section 2.1.9 "Inductive types and recursive functions"*. In [The Coq Development Team 2024a]. https://coq.inria.fr/doc/V8.19.0/refman/language/core/inductive

The Coq Development Team. 2024e. *Section 2.2.6 "Implicit Coercions"*. In [The Coq Development Team 2024a]. https://coq.inria.fr/doc/V8.19.0/refman/addendum/implicit-coercions

The GHC Team. 2021. Glasgow Haskell Compiler User's Guide. https://downloads.haskell.org/~ghc/8.10.7/docs/html/users_guide/ Accessed: 2021-01-24, Version 8.10.7.

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. https://doi.org/10.1145/1328438.1328445

Philip Wadler. 1984. Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile Time. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*. ACM, 45–52. https://doi.org/10.1145/800055.802020

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. https://doi.org/10.1145/99370.99404

# A  BASIC DEFINITIONS AND FACTS USED FOR PROOFS

This appendix provides a list of some definitions and lemmas in the Mathematical Components (MathComp) library [Mahboubi and Tassi 2022] used for the proofs in Sections 3.3, 3.4 and 4.4.3 and Appendix B. Most informal definitions and lemmas are followed by the corresponding formal definitions and statements in Coq. 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 type $T \to \text{bool}$ and $T \to T \to \text{bool}$, 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 $T \to S \to \text{bool}$, *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 $T \to \text{Prop}$ and $T \to T \to \text{Prop}$ in Coq.

*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:

$$x \leq_{(1,2)} y := x \leq_1 y \land (y \not\leq_1 x \lor x \leq_2 y).$$

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 Coq.

*Definition A.12 (Concatenation).* Given two lists $s_1 := [x_1, \ldots, x_n]$ and $s_2 := [y_1, \ldots, y_m]$, 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 Coq 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 $s := [x_1, x_2, \ldots, x_n]$, 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 Coq 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 Coq 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 Coq 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 Coq 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 Coq 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 $T$, $|s_1 \mathbin{+\mkern-10mu+} s_2|_p = |s_1|_p + |s_2|_p$ *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 $T$, $|\mathrm{rev}\ s|_p = |s|_p$ *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 $T$, $\mathrm{all}_p\ s$ 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 $T$, $\mathrm{all}_p\ s$ *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 $T$, $\mathrm{all}_{p_2}\ s$ *holds whenever* $\mathrm{all}_{p_1}\ s$ *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 $S$, $\mathrm{allrel}_R\ s_1\ s_2$ 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 $S$, $\mathrm{allrel}_R\ s_1\ s_2$ *holds iff* $\mathrm{allrel}_{R^C}\ s_2\ s_1$ *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 $S$, $\mathrm{allrel}_R\ (\mathrm{rev}\ s_1)\ (\mathrm{rev}\ s_2)$ *holds iff* $\mathrm{allrel}_R\ s_1\ s_2$ *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 $T_1 \rightarrow T_2$ and a list $s := [x_1, x_2, \ldots, x_n]$ of type list $T_1$, $\mathrm{map}_f\ s$ 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 Coq 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 $T$, $\mathrm{filter}_p\ s$ 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 Coq 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 $T_1 \rightarrow T_2$, $p \subseteq T_2$, and $s$ of type* list $T_1$, $\mathrm{filter}_p\ (\mathrm{map}_f\ s) = \mathrm{map}_f\ (\mathrm{filter}_{p \circ f}\ s)$ *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* $\mathrm{filter}_p\ s$ *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 $T$, $\mathrm{mask}_m\ s$ is the subsequence of $s$ selected by $m$; that is, the $i^{\mathrm{th}}$ element of $s$ is selected if the $i^{\mathrm{th}}$ 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 $T_1 \rightarrow T_2$, and $m$ and $s$ of respectively types* list bool *and* list $T_1$, $\mathrm{map}_f\ (\mathrm{mask}_m\ s) = \mathrm{mask}_m\ (\mathrm{map}_f\ s)$ *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 $T$, $\text{filter}_p\ s = \text{mask}_{m'}\ s$ *where $m' := \text{map}_p\ s$ 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, $\text{mask}_m\ s = \text{filter}_p\ s$ *where $p\ x := x \in \text{mask}_m\ s$ 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 $s_1 = \text{mask}_m\ s_2$.*

```
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 $T$, $\text{mask}_m\ s$ *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 $T$, $\text{index}_x\ s$ 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 $s$:*

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

```
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 $s_1 =_{\text{perm}} s_2$.

```
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 $s_1 =_{\text{perm}} s_2$ to rewrite a goal of the form $s_1 =_{\text{perm}} s_3$ to $s_2 =_{\text{perm}} s_3$, some lemmas, *e.g.*, Lemmas A.55 and A.56, are stated in the form of $\forall s_3, (s_1 =_{\text{perm}} s_3) = (s_2 =_{\text{perm}} s_3)$, 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 $T$, $s_1 =_{\text{perm}} s_2$ *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 $s_2$, $s_1 =_{\text{perm}} s_2$ holds iff $(s_1 =_{\text{perm}} s_3) = (s_2 =_{\text{perm}} s_3)$ 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 $s_1 =_{\text{perm}} s_2$ 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 := $(\forall x, x \in \text{s1} = x \in \text{s2})$ *means that* s1 *and* s2 *have the same set of elements.*

LEMMA A.53. *For any lists $s_1$ and $s_2$ such that $s_1 =_{\text{perm}} s_2$, $s_1$ 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 $=_{\text{perm}}$ is a congruence relation with respect to concatenation $+\!\!+$; that is, $s_1 =_{\text{perm}} s_2$ and $t_1 =_{\text{perm}} t_2$ imply $s_1 +\!\!+ t_1 =_{\text{perm}} s_2 +\!\!+ t_2$ 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 3.6).* Given a relation $R \subseteq T \times T$, a list $s :=$ $[x_0, \ldots, x_n]$ of type list $T$ is said to be *pairwise sorted* by $R$ 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 by 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 by 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 3.6).* Given a relation $R \subseteq T \times T$, a list $s :=$ $[x_0, \ldots, x_n]$ 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 by $R$.

In Coq, 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 by R iff s is sorted by 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 by R whenever s is sorted by 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* $s_1 =_{\text{perm}} s_2$ *and both* $s_1$ *and* $s_2$ *are sorted by* $\leq$.

```
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 by* $\leq$.

```
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 $i^{\text{th}}$ 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* $[\text{nth } x_0 \ s \ i \mid i \leftarrow \text{iota } 0 \ |s|]$.

```
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 by 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 $s_1 \ \mathbb{M}_{\leq} \ s_2$ is a list of type list $T$ defined as follows:

$$[] \ \mathbb{M}_{\leq} \ s_2 := s_2$$

$$s_1 \ \mathbb{M}_{\leq} \ [] := s_1$$

$$(x :: s_1) \ \mathbb{M}_{\leq} \ (y :: s_2) := \begin{cases} x :: (s_1 \ \mathbb{M}_{\leq} \ (y :: s_2)) & (x \leq y) \\ y :: ((x :: s_1) \ \mathbb{M}_{\leq} \ s_2). & (\text{otherwise}) \end{cases}$$

```
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 $T$, $|s_1 \barwedge_\leq s_2|_p$ *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 $T$, $s_1 \barwedge_\leq s_2$ *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 $T$, $s_1 \barwedge_\leq s_2$ *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 $T$, $s_1 \barwedge_{\leq_1} s_2$ *is sorted by the lexicographic order $\leq_{\text{lex}} := \leq_{(1,2)}$ 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 by $\leq_{\text{lex}}$.*

```
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 $\leq\ \subseteq T \times T$, $\barwedge_\leq$ 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* sig $D_1$ *and* sig $D_1$ *are subtypes of $T_1$ and $T_2$ collecting inhabitants satisfying $D_1$ and $D_2$, respectively. Then,* (val $x$, val $y$) $\in P$ *holds for any $x \in$ sig $D_1$ and $y \in$ 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* $\mathrm{map}_{\mathrm{val}}\ s'$, *whenever* $\mathrm{all}_p\ s$ *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 Coq, 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, $xs \sim \mathrm{sort}_{\leq}\ xs$ holds whenever the following four induction cases hold:*

- *for any lists $xs$, $xs'$, $ys$, and $ys'$ of type* list $T$, $(xs \mathbin{+\!\!+} ys) \sim (xs' \mathbin{⋏_{\leq}} ys')$ *holds whenever $xs \sim xs'$ and $ys \sim ys'$ hold,*
- *for any lists $xs$, $xs'$, $ys$, and $ys'$ of type* list $T$, $(xs \mathbin{+\!\!+} ys) \sim \mathrm{rev}\ (\mathrm{rev}\ ys' \mathbin{⋏_{\geq}} \mathrm{rev}\ xs')$ *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 3.3 and 4.1.                                                                                          □

LEMMA B.2 (THE NATURALITY OF sort [WADLER 1989, SECTION 3.3], LEMMA 3.10). *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:*

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

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

```
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 3.10.                                                                                           □

Lemma B.3. *For any* $\leq \; \subseteq T \times T$ *and $s$ of type* list $T$ *pairwise sorted by* $\leq$, $\mathsf{sort}_\leq s$ *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 $s_1' \curlywedge_\leq s_2' = s_1 + s_2$ holds whenever $s_1 + s_2$ is pairwise sorted by $\leq$, which is equivalent to the following conjunction (Lemma A.58):

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

Then,

$$s_1' \curlywedge_\leq s_2' = s_1 \curlywedge_\leq s_2 \qquad\qquad \text{((i) and I.H.)}$$
$$= s_1 + s_2. \qquad\qquad \text{((ii) and Lemma A.73)}$$

For the second case, we show that $\mathsf{rev}\,(\mathsf{rev}\,s_2' \curlywedge_\geq \mathsf{rev}\,s_1') = s_1 + s_2$ holds whenever $s_1 + s_2$ is pairwise sorted by $\leq$, which is equivalent to the following conjunction (Lemma A.58):

  (iii) both $s_1$ and $s_2$ are pairwise sorted by $\leq$, 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,

$$\mathsf{rev}\,(\mathsf{rev}\,s_2' \curlywedge_\geq \mathsf{rev}\,s_1') = \mathsf{rev}\,(\mathsf{rev}\,s_2 \curlywedge_\geq \mathsf{rev}\,s_1) \qquad\qquad \text{((iii) and I.H.)}$$
$$= \mathsf{rev}\,(\mathsf{rev}\,s_2 + \mathsf{rev}\,s_1) \qquad\qquad \text{((iv) and Lemma A.73)}$$
$$= s_1 + s_2. \qquad\qquad \text{(Lemmas A.19 and A.20)}$$

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 by* $\leq$, $\mathsf{sort}_\leq s$ *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.                                                                    □

Lemma B.5 (Lemma 3.7). *For any* $\leq_1, \leq_2 \; \subseteq T \times T$ *and xs of type* list $T$, $\mathsf{sort}_{\leq_1} xs$ *is sorted by the lexicographic order* $\leq_{\mathrm{lex}} := \leq_{(1,2)}$ *whenever* $\leq_1$ *is total and xs is pairwise sorted by* $\leq_2$.

```
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 $\mathrm{sort}_\leq xs$ (Lemma B.1). See Sections 3.4.1 and 4.4.3 for details. □

LEMMA B.6 (COROLLARY 3.8). *For any $\leq_1, \leq_2 \subseteq T \times T$ and xs of type* list $T$, $\mathrm{sort}_{\leq_1} xs$ *is sorted by the lexicographic order $\leq_{\mathrm{lex}} := \leq_{(1,2)}$ whenever $\leq_1$ is total, $\leq_2$ is transitive, and xs is sorted by $\leq_2$.*

```
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 Lemmas A.60 and B.5. □

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

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

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

$$\begin{aligned}
|s_1' \wedge_\leq s_2'|_p &= |s_1'|_p + |s_2'|_p & \text{(Lemmas A.26 and A.71)} \\
&= |s_1|_p + |s_2|_p & \text{(I.H.)} \\
&= |s_1 + s_2|_p. & \text{(Lemma A.26)}
\end{aligned}$$

For the second case,

$$\begin{aligned}
|\mathrm{rev}\,(\mathrm{rev}\,s_2' \wedge_\geq \mathrm{rev}\,s_1')|_p &= |s_2'|_p + |s_1'|_p & \text{(Lemmas A.26, A.27 and A.71)} \\
&= |s_1'|_p + |s_2'|_p \\
&= |s_1|_p + |s_2|_p & \text{(I.H.)} \\
&= |s_1 + s_2|_p. & \text{(Lemma A.26)}
\end{aligned}$$

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* $\mathrm{sort}_\leq s$ *and s are equal, i.e.,* $|\mathrm{sort}_\leq s| = |s|$.

```
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 $\mathrm{sort}_{\leq}$ [] 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* $\mathrm{sort}_{\leq} s$ *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 3.4). *For any $\leq \subseteq T \times T$ and $s$ of type* list $T$, $\mathrm{sort}_{\leq} s$ *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 3.5). *For any $\leq \subseteq T \times T$ and $s$ of type* list $T$, $\mathrm{sort}_{\leq} s$ *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 $T$, $\mathrm{sort}_{\leq} s$ *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.    □

LEMMA B.14 (LEMMA 3.11 AND COROLLARY 5.2). *For any total preorder $\leq \subseteq T \times T$ and predicate $p \subseteq T$, $\mathrm{filter}_{p}$ commutes with $\mathrm{sort}_{\leq}$ under function composition; that is, the following equation holds for any $s$ of type* list $T$:*

$$\mathrm{filter}_{p} (\mathrm{sort}_{\leq} s) = \mathrm{sort}_{\leq} (\mathrm{filter}_{p} s).$$

```
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 Lemma 3.11 for details.    □

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

$$\mathrm{filter}_{p} (\mathrm{sort}_{\leq} s) = \mathrm{filter}_{p} s$$

*holds whenever $\mathrm{filter}_{p} s$ sorted by $\leq$.*

```
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 Lemmas B.4 and 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* $\leq_{\mathrm{lex}} := \leq_{(1,2)}$; *that is, the following equation holds for any s of type* list *T:*

$$\mathsf{sort}_{\leq_1} (\mathsf{sort}_{\leq_2} s) = \mathsf{sort}_{\leq_{\mathrm{lex}}} s.$$

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

$$\mathsf{sort}_{\leq'_1} (\mathsf{sort}_{\leq'_2} [0, \ldots, |s| - 1]) = \mathsf{sort}_{\leq'_{\mathrm{lex}}} [0, \ldots, |s| - 1]$$

where $\leq'_1$, $\leq'_2$, and $\leq'_{\mathrm{lex}}$ are the preimages of $\leq_1$, $\leq_2$, and $\leq_{\mathrm{lex}}$ under $\mathsf{nth}\ x_0\ s$, respectively. Thanks to Lemma B.6, each side of the above equation is respectively sorted by

- the (right-associative) lexicographic composition of $\leq'_1$, $\leq'_2$, and $<_{\mathbb{N}}$, and
- the lexicographic composition of $\leq'_{\mathrm{lex}}$ 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 $\leq'_{\mathrm{lex}}$ are total (Lemma A.9),
- $\leq'_2$, $<_{\mathbb{N}}$, and their lexicographic composition are transitive (Lemma A.9), and
- $[0, \ldots, |s| - 1]$ is sorted by $<_{\mathbb{N}}$ (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,* $\mathsf{sort}_{\leq} s$ *is sorted by* $\leq$ *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 by the trivial relation $R$ such that $x \; R \; y$ holds for any $x, y \in T$, Corollary 3.8 implies that $\text{sort}_\leq s$ is sorted by 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 $T$, $\text{sort}_\leq s_1 = \text{sort}_\leq s_2$ 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 $\text{sort}_\leq s_1$ and $\text{sort}_\leq s_2$ are respectively permutations of $s_1$ and $s_2$, $\text{sort}_\leq s_1 =_{\text{perm}} \text{sort}_\leq s_2$ holds iff $s_1 =_{\text{perm}} s_2$ holds. Therefore, ($\Rightarrow$) is trivial.

($\Leftarrow$) $\text{sort}_\leq s_1$ and $\text{sort}_\leq s_2$ are sorted by $\leq$ (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. [2024, THEOREM 2.9 (UNIQUENESS OF SORTING)]). *For any two stable sort functions* sort *and* sort′ *and total preorder $\leq \, \subseteq T \times T$, $\text{sort}_\leq$ and $\text{sort}'_\leq$ are extensionally equal; that is, $\text{sort}_\leq s = \text{sort}'_\leq s$ 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 $\text{map}_{\text{nth } x_0 \, s} [0, \ldots, |s| - 1]$ (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

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

where $x \leq_I y := \text{nth } x_0 \, s \, x \leq \text{nth } x_0 \, s \, y$.

Since $[0, \ldots, |s| - 1]$ is sorted by $<_\mathbb{N}$ (Lemma A.69), both sides of the above equation are sorted by $<_I$, 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 $T$, $\mathsf{sort}_{\leq} \; s = \mathsf{isort}_{\leq} \; s$ *holds for the insertion sort* isort *defined as follows.*

$$\mathsf{isort}_{\leq} \; [] \coloneqq []$$
$$\mathsf{isort}_{\leq} \; (x :: s) \coloneqq [x] \; \mathbb{M}_{\leq} \; \mathsf{isort}_{\leq} \; s.$$

```
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 our characterization property. □

LEMMA B.21. *For any total preorder $\leq \; \subseteq T \times T$, and $s_1$ and $s_2$ of type* list $T$, $\mathsf{sort}_{\leq} \; (s_1 + s_2) = \mathsf{sort}_{\leq} \; s_1 \; \mathbb{M}_{\leq} \; \mathsf{sort}_{\leq} \; s_2$ *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 $s_1$:

$$\mathsf{isort}_{\leq} \; (s_1 + s_2) = \mathsf{isort}_{\leq} \; s_1 \; \mathbb{M}_{\leq} \; \mathsf{isort}_{\leq} \; s_2.$$

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

$$\begin{aligned}
\mathsf{isort}_{\leq} \; (s_1 + s_2) &= [x] \; \mathbb{M}_{\leq} \; \mathsf{isort}_{\leq} \; (s_1' + s_2) & \text{(Definition)} \\
&= [x] \; \mathbb{M}_{\leq} \; (\mathsf{isort}_{\leq} \; s_1' \; \mathbb{M}_{\leq} \; \mathsf{isort}_{\leq} \; s_2) & \text{(I.H.)} \\
&= ([x] \; \mathbb{M}_{\leq} \; \mathsf{isort}_{\leq} \; s_1') \; \mathbb{M}_{\leq} \; \mathsf{isort}_{\leq} \; s_2 & \text{(Lemma A.75)} \\
&= \mathsf{isort}_{\leq} \; s_1 \; \mathbb{M}_{\leq} \; \mathsf{isort}_{\leq} \; s_2. & \text{(Definition)} \quad \square
\end{aligned}$$

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* $\mathsf{mask}_{m'} \; (\mathsf{sort}_{\leq} \; s) = \mathsf{sort}_{\leq} \; (\mathsf{mask}_m \; s)$.

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

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

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

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

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* $\mathsf{mask}_{m'}\ (\mathsf{sort}_{\leq}\ s) = \mathsf{mask}_m\ s$, *whenever* $\mathsf{mask}_m\ s$ *is sorted by $\leq$.*

```
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.                                              $\square$

Lemma B.24. *For any total preorder $\leq\ \subseteq T \times T$, $\mathsf{sort}_{\leq}$ preserves the subsequence relation; that is, for any $t$ and $s$ of type* list $T$, $\mathsf{sort}_{\leq}\ t$ *is a subsequence of* $\mathsf{sort}_{\leq}\ s$ *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 $t = \mathsf{mask}_m\ s$, we are left to show that $\mathsf{sort}_{\leq}\ (\mathsf{mask}_m\ s)$ is a subsequence of $\mathsf{sort}_{\leq}\ s$. Now, by Lemma B.22, there is a mask $m'$ such that $\mathsf{sort}_{\leq}\ (\mathsf{mask}_m\ s) = \mathsf{mask}_{m'}\ (\mathsf{sort}_{\leq}\ s)$ which is indeed a subsequence of $\mathsf{sort}_{\leq}\ s$ (Lemma A.45).                                              $\square$

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* $\mathsf{sort}_{\leq}\ s$ *whenever $t$ is a subsequence of $s$ and sorted by $\leq$.*

```
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* $\mathsf{sort}_\leq s$ *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 $\mathsf{sort}_\leq s$, 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 $\mathsf{sort}_\leq [x, y]$ is a subsequence of $\mathsf{sort}_\leq s$. Since $x \leq y$, we have that $\mathsf{sort}_\leq [x, y] = [x, y]$ (*e.g.*, by computation through Lemma B.20), hence $[x, y]$ a subsequence of $\mathsf{sort}_\leq s$. □

## C COMPARISON OF FORMULATIONS OF THE STABILITY

In this appendix, we compare the statements of the stability results we proved with those of 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 3.12). *For any total preorder $\leq$ on T, the equivalent elements, such that x and y satisfying $x \sim y := x \leq y \wedge y \leq x$, 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 $T$:

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

```
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 Lemma B.15. It suffices to show that $[y \leftarrow s \mid x \sim y]$ is sorted by $\leq$, which is trivial since its all elements are equivalent to $x$. □

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 $T \rightarrow T'$.

Lemma C.2 (Leino and Lucio [2015]; Sternagel [2013]). *Let $k : T \rightarrow T'$ 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., $x \leq_T y := k\,x \leq_{T'} k\,y$. For any s of type* list $T$, *the relative order of the elements of s having the same key is preserved by* $\mathsf{sort}_{\leq_T} s$; *that is, the following equation holds for any x of type T:*

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

```
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. 10. Structurally-recursive non-tail-recursive merge in Coq, *i.e.*, the Coq counterpart of merge in Figure 7.

```
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 ($x \leq_T y \wedge y \leq_T x$). Since $\leq_T$ is total preorder, this follows from Lemma C.1.                                                                                            □

As it can be seen, both Lemmas C.1 and C.2 are immediate consequences of Lemma B.15, which follows from Lemma B.14. As we argued in Section 3.4.2, Lemmas B.14 and B.15 have the advantage that it can be used in a place Lemmas C.1 and C.2 do not directly apply, *i.e.*, where the predicate is not one collecting equivalent elements. although Lemmas B.17 and C.1 imply Lemma B.6, and thus, Lemma B.14.

# D  DEFINITIONS OF MERGESORTS IN COQ

In Section 5.1, we presented structurally-recursive non-tail-recursive and tail-recursive mergesorts in OCaml. In this appendix, we present their Coq reimplementations, including some optimized ones such as smooth variants (Section 4.3). The formal correctness proofs of these implementations are omitted in this appendix and available only in the supplementary material.

## D.1  Non-tail-recursive mergesorts in Coq

In Figure 10, we define a module type [The Coq Development Team 2024c] 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; for example, one may replace the merge function with

```
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. 11. Structurally-recursive non-tail-recursive mergesorts in Coq. The push, pop, sort1rec, and sort1 functions in this figure are the Coq counterparts of push, pop, sort_rec, and sort in Figure 7. Some optimized implementations are omitted here as marked by [..] and shown in Figures 12 and 13.

one that uses a single fixpoint and an explicit termination proof, which has better performance in extracted [Letouzey 2004] OCaml and Haskell code, and thus, is used in our benchmark (Appendix F). Therefore, the non-tail-recursive mergesort functions in Figure 11 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 10, or Definition A.70).

Inside the CBN_ module, we introduce a type T and a relation leT on T as section variables [The Coq Development Team 2024b]. 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 7, 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,

```
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. 12. A structurally-recursive non-tail-recursive mergesorts in Coq, that takes three elements from the input at a time. This implementation is omitted in the place marked by [..] in Figure 11.

```
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. 13. A smooth structurally-recursive non-tail-recursive mergesort in Coq. This implementation is omitted in the place marked by [..] in Figure 11.

```
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. 14. Structurally-recursive tail-recursive merge in Coq. The merge_rec function in this figure is the Coq counterpart of revmerge in Figure 9, and revmerge in this figure instantiate it with [] as the accumulator.

the sort3 function in Figure 12 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 13 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 Coq.

## D.2 Tail-recursive mergesorts in Coq

Similarly to the non-tail-recursive counterpart (Figures 10 and 11 in Appendix D.1), we define a module type RevmergeSig and its instance Revmerge implementing a tail-recursive merge function (Figure 14), and the tail-recursive mergesort functions are provided as a functor module CBV_ that takes an instance M of RevmergeSig as a parameter (Figure 15).

The sort2 and sort3 functions in Figures 15 and 16 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 17 is a smooth

```
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. 15. Structurally-recursive tail-recursive mergesorts in CoQ. The push, pop, sort1rec, and sort1 functions in this figure are the CoQ counterparts of push, pop, sort_rec, and sort in Figure 9. Some optimized implementations are omitted here as marked by [..] and shown in Figures 16 and 17.

```
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. 16. A structurally-recursive tail-recursive mergesorts in Coq, that takes three elements from the input at a time. This implementation is omitted in the place marked by [..] in Figure 15.

```
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. 17. A smooth structurally-recursive tail-recursive mergesort in Coq. This implementation is omitted in the place marked by [..] in Figure 15.

```
let rec merge (<=) xs ys =            and merge_dps dst i (<=) xs ys =
  match xs, ys with                     match xs, ys with
  | [], ys -> ys                        | [], ys -> dst.i <- ys
  | xs, [] -> xs                        | xs, [] -> dst.i <- xs
  | x :: xs', y :: ys' ->               | x :: xs', y :: ys' ->
    if x <= y then                        if x <= y then
      let dst = x :: Hole in                let dst' = x :: Hole in
      merge_dps dst 1 (<=) xs' ys;          dst.i <- dst';
      dst                                   merge_dps dst' 1 (<=) xs' ys
    else                                  else
      let dst = y :: Hole in                let dst' = y :: Hole in
      merge_dps dst 1 (<=) xs ys';          dst.i <- dst';
      dst                                   merge_dps dst' 1 (<=) xs ys'
```

Fig. 18. The non-tail-recursive merge function (Figure 1a) turned into a tail-recursive merge function in destination-passing style by the Tail Modulo Cons (TMC) transformation.

structurally-recursive tail-recursive mergesort, implemented in the same way as the non-tail-recursive counterpart (Figure 13).

We stress that the recursive functions defined and used in Figures 14 to 17 are tail recursive except for push in Figure 15, whose depth of recursive calls is logarithmic in the length of the input. Therefore, a linear consumption of stack space does not occur in any of the mergesort functions presented in this section.

# E   NON-TAIL-RECURSIVE MERGESORT IS TAIL-RECURSIVE MODULO CONS

In this appendix, we show that non-tail-recursive mergesort can actually be transformed to a tail-recursive function in destination-passing style [Larus 1989; Minamide 1998].

Let us recall that the recursive calls of merge in Figure 1a appear in non-tail position and its caller still have to compute one cons cell. Nevertheless, if in-place update of constructor parameters is allowed, one may write an equivalent tail-recursive function in destination-passing style. It can be described as the merge_dps function in Figure 18 written in pseudo-OCaml syntax borrowed from Bour et al. [2021]. The merge_dps function takes a new argument dst which is a cons cell, and writes the resulting list to the tail of dst instead of returning it. Focusing on the case that both arguments are not empty and x <= y holds, it first builds a cons cell dst' (:= x :: Hole) where Hole means an uninitialized field, updates the $1^{st}$ field (counting from 0, hence the tail) of dst with dst' by dst.1 <- dst', and then calls merge_dps recursively to let it fill the uninitialized tail of dst'. The merge function wraps this merge_dps function to provide the same interface as the non-tail-recursive merge function in Figure 1a.

In fact, this program transformation, called *Tail Modulo Cons* (TMC) transformation [Bour et al. 2021][Leroy et al. 2022, Chapter 24], introduced in OCaml 4.14, can automatically turn tail calls *modulo constructor* [Wadler 1984] into tail calls in destination-passing style.[5] For example, all the recursive functions in the naive bottom-up non-tail-recursive mergesort (Figure 2b) and structurally-recursive non-tail-recursive mergesorts (Section 5.1.2 and Appendix D.1) are tail-recursive-modulo-cons. The technique presented in Section 4.1 allows us to implement a top-down non-tail-recursive mergesort that is tail-recursive-modulo-cons except for the sort_rec function,

---

[5]At most one tail call modulo constructor in one branch can be turned into a tail call in destination-passing style. The recursive function to be transformed should be marked by the [@tail_mod_cons] attribute, and it is recommended to mark each tail call modulo constructor to be transformed by the [@tailcall] attribute.

whose depth of recursive calls is logarithmic in the length of the input. Therefore, this optimization technique works for both top-down and bottom-up non-tail-recursive mergesort functions.

In Appendix F, we show benchmark results indicating that structurally-recursive non-tail-recursive mergesorts (Appendix D.1) extracted to OCaml and then automatically turned into tail-recursive mergesorts by the TMC transformation are consistently slower than their tail-recursive counterparts (Appendix D.2).

## F  PERFORMANCE EVALUATION

This section evaluates the performance of OCaml and Haskell code extracted [Letouzey 2004] from our CoQ implementation of structurally-recursive non-tail-recursive and tail-recursive mergesort functions (Appendix D) to verify our claims on the performance of mergesort argued in Section 4. We also compare them with the performance of mergesort functions in the standard libraries, *i.e.*, List.stable_sort of OCaml and Data.List.sort of Haskell. These benchmarks in OCaml and Haskell are intended to evaluate the performance of mergesort functions in call-by-value and call-by-need evaluations, respectively.

The CoQ implementations we compare are the "three elements at a time" and smooth variants (sort3 and sortN) of non-tail-recursive (Appendix D.1) and tail-recursive (Appendix D.2) merge-sorts. The "one/two elements at a time" variants (sort1 and sort2) are consistently slower than sort3 in any of our benchmark results, and thus excluded. In any of these, the merge function is replaced with one using accessibility proofs (Appendix D.1) to avoid extracting nested recursion.
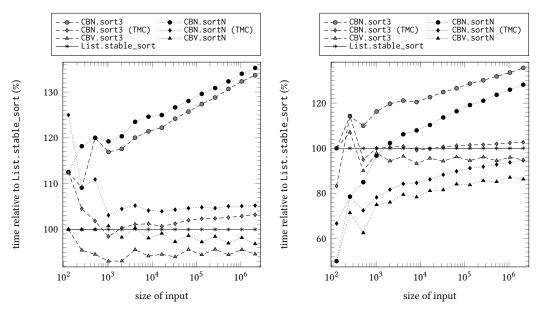
We evaluate their performance using two kinds of inputs: lists of random integers and those preprocessed so that every block of length 50 is sorted in ascending order (hereinafter called *partially-sorted input*). The latter kind of input is intended to evaluate their smoothness (Section 4.3). For Haskell, we also evaluated their performance in case of obtaining the first 25% of the output, to evaluate their laziness and incrementality (Section 4.2).

### F.1  OCaml

Figure 19 shows the benchmark results of extracted OCaml code, including non-tail-recursive mergesorts optimized by the TMC transformation (Appendix E), and List.stable_sort. Besides the facts implied from the figure, non-tail-recursive variants use up stack space but their TMC counterparts do not. Regardless of whether the inputs are random or partially-sorted, the non-tail-recursive variants (CBN.sort3 and CBN.sortN) are noticeably slower than any other, and their TMC counterparts (CBN.sort3 (TMC) and CBN.sortN (TMC)) are 4%–8% slower than their tail-recursive counterparts (CBV.sort3 and CBV.sortN), respectively. For random inputs (Figure 19a), any TMC or tail-recursive variant achieve quite comparable performances with List.stable_sort. For partially-sorted inputs (Figure 19b), the smooth variants become faster as expected.

### F.2  Haskell

Figure 20 shows the benchmark results of extracted Haskell code and Data.List.sort. If we compute only the first 25% of the output instead of the entire output (Figures 20a and 20b), the non-tail-recursive variants outperform their tail-recursive counterparts, in contrast to OCaml. This observation corresponds to the fact that non-tail-recursive mergesort in call-by-need evaluation is an optimal incremental sort algorithm (Section 4.2). Furthermore, Data.List.sort is even faster than our non-tail-recursive mergesort particularly for large random inputs (Figure 20a). However, we have not identified the clear reason of this performance difference yet. On the other hand, if we require the entire output (Figures 20c and 20d), the tail-recursive variants outperform the non-tail-recursive variants including Data.List.sort. For partially sorted inputs (Figures 20b and 20d), the smooth variants including Data.List.sort become faster as expected.

(a) For a random input, sort the entire list.

(b) For a input consisting of sorted blocks of length 50, sort the entire list.

Fig. 19. Benchmark results of sorting in OCaml. The program was compiled by OCaml 4.14.0 with an optimization flag -O3 for the Flambda [Leroy et al. 2022, Chapter 21] optimizer, and ran with ulimit -s unlimited to avoid stack overflow. To avoid performing garbage collections during a measurement, the minor heap was enlarged beforehand, and a full major garbage collection and a heap compaction (Gc.compact) were invoked before each measurement.
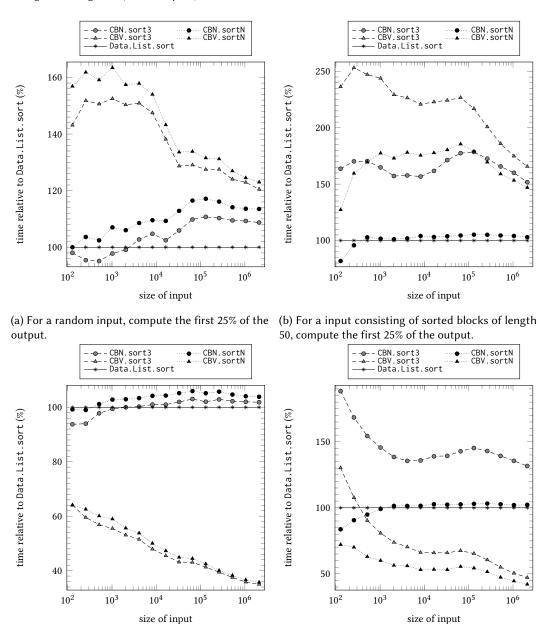
(a) For a random input, compute the first 25% of the output.

(b) For a input consisting of sorted blocks of length 50, compute the first 25% of the output.

(c) For a random input, sort the entire list.

(d) For a input consisting of sorted blocks of length 50, sort the entire list.

Fig. 20. Benchmark results of sorting in Haskell. The program was compiled by GHC 8.10.7 with an optimization flag -O2. To avoid performing garbage collections during a measurement, the allocation area was enlarged beforehand, and a major garbage collection (System.Mem.performMajorGC) was invoked before each measurement.