Formally Verified Suffix Array Construction

Louis Cheung[1*], Alistair Moffat[1*] and Christine Rizkallah[1*]

[1]The University of Melbourne, Melbourne, Australia.

*Corresponding author(s). E-mail(s):
lfcheung@student.unimelb.edu.au; ammoffat@unimelb.edu.au;
christine.rizkallah@unimelb.edu.au;

Abstract

Suffix arrays are a data structure with numerous real-world applications. They are extensively used in text retrieval and data compression applications, including query suggestion mechanisms in web search, and in bioinformatics tools for DNA sequencing and matching. This wide applicability means that algorithms for constructing suffix arrays are of great practical importance. The SA-IS algorithm is an efficient but conceptually complex suffix array construction technique, and implementing it requires a deep understanding of its underlying theory. As a critical step towards developing a provably correct and efficient implementation, we have developed the SA-IS algorithm in Isabelle/HOL and formally verified that it is equivalent to a mathematical functional specification of suffix arrays, a task that required verifying a wide range of underlying properties of strings and suffixes. We also used Isabelle's code extraction facilities to extract an executable Haskell implementation of SA-IS, which albeit is inefficient due to using lists and natural numbers rather than arrays and machine words, demonstrates that our verified HOL implementation of SA-IS can be refined to an executable implementation in its current form.

Keywords: Formal Verification, Data Structures, Suffix Array Construction, SA-IS Algorithm, Isabelle/HOL

1 Introduction

The *suffix array* [1] is an indexing data structure that stores all of the suffixes of a text in lexicographical order, with each suffix represented by its location in the text. This data structure is widely used in pattern matching problems in particular, as a space-efficient alternative to the suffix tree. Indeed, the suffix array can completely replace a suffix tree when augmented by additional data derivable from the text [2]. Importantly, the suffix array can be constructed in time linear in the length of the input for texts drawn from constant-sized alphabets [3–6].

One specific application of the suffix array is to compute the *Burrows-Wheeler transform* (BWT) [7]. As suffix arrays can be constructed in linear time, the overall time cost for computing the BWT is also linear. The BWT is an invertible data transformation that is widely used in data compression, such as `bzip2` [8], and in compressed text search [9], most notably in bioinformatics where it is used for DNA analysis [10–13].

Even though the suffix array is widely used, no formal verification of a linear time construction algorithm exists, with the only verification being of suffix array construction based directly on generic string sorting (shown in Figure 1). This straightforward approach is only suitable for moderate-length input texts, and for texts that do not contain long subsequence repetitions. Important inputs such as genomes do not satisfy these requirements, and the worst-case time complexity of $\mathcal{O}(n^2 \log n)$ for sort-based suffix array construction for an input text of length n becomes infeasible once n reaches approximately 10^6 characters.

In this paper, we present a formal verification of the *suffix array construction by induced sorting* (SA-IS) algorithm [5, 6] in Isabelle/HOL [14]. The SA-IS algorithm is a linear time suffix array construction algorithm that is also very fast in practice. To demonstrate that the embedding of the algorithm corresponds to executable code, we utilize Isabelle/HOL's code extraction features to produce an executable Haskell implementation, requiring only code equation equivalence lemmas.

In detail, the contributions of this work are as follows:

- a specification of suffix arrays;
- an Isabelle/HOL embedding and verification of the SA-IS algorithm, including a formalization of properties about sequences that the algorithm depends on;
- a Haskell implementation of the SA-IS algorithm extracted from the Isabelle/HOL embedding; and
- a formalization of a lifting function that removes the assumption that all input sequences must be terminated by a unique symbol that is smaller than any other symbol in the sequence, which is a common assumption for many suffix array construction algorithms.

The remainder of the paper delivers on those goals. Section 2 provides definitions and the necessary background material, and lays the foundation for the new work. Section 3 describes the SA-IS algorithm, and provides an overview of its worst case linear time analysis. The correctness of the SA-IS method is then considered in detail in Sections 4 to 6, via a sequence of definitions, lemmas, and theorems. Sections 7 and 8 respectively discuss the experience that we built while constructing the proof, and summarize a range of related work. Finally, Section 9 concludes our presentation.

2 Problem Description

This section first introduces key concepts relevant to suffix array construction in Section 2.2. Building on those ideas, we then give a description of suffix arrays and their construction in Section 2.3, with in-depth treatment available in the work of Crochemore et al. [15, Chapter 4] if desired. Finally, we provide a brief overview of the Isabelle/HOL theorem prover in Section 2.4. For the reader interested in learning more about Isabelle/HOL and how to use it, we recommend the books by Nipkow et al. [14] and by Nipkow and Klein [16].

2.1 Notation

In this paper, different fonts are used to indicate whether a term is a function or variable, a predicate, or a constant. In particular, functions and variables are *italicized*, predicates are written in **bold** font, and constants, such as concrete letters or symbols from an alphabet, are presented in `typewriter` font. Unless otherwise stated, the symbols S, S' $\ldots$, $S'^{\cdots\prime}$ are used to represent lists or sequences that are inputs to a suffix array construction algorithm. Similarly, the symbols Σ and Σ' are alphabets or sets of symbols that elements of sequences are drawn from, and these alphabets are finite and have a total ordering, meaning that these can be monotonically mapped to a subset of the natural numbers, for example, the set containing all numbers from 0 up to the size of the alphabet. Finally, the symbol SA is used for the suffix array.

2.2 Problem Preliminaries

We consider finite sequences over an alphabet Σ that has a total ordering and a minimal element. Letters `a`, `b`, `c` and so on are used to represent elements of Σ and might be individual characters, or might be other symbols such as natural numbers. Sequences in Σ^* are represented as ordered lists of symbols. For example, given the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$, `aabdabb` is one of the many possible sequences of length seven.

Given that the alphabet Σ is finite and ordered, Σ has a monotonic mapping to a subset of the natural numbers. Moreover, we can infer an ordering on sequences drawn from Σ^* with the usual lexicographic rules as follows:

Definition 1 (Lexicographical Order, $<_{lex}$)

$$[] <_{lex} (y \# ys) := \mathsf{true} \tag{1a}$$

$$(x \# xs) <_{lex} [] := \mathsf{false} \tag{1b}$$

$$[] <_{lex} [] := \mathsf{false} \tag{1c}$$

$$(x \# xs) <_{lex} (y \# ys) := (x \leq y) \wedge (x = y \longrightarrow xs <_{lex} ys)\,, \tag{1d}$$

where $x \# xs$ is list/sequence the cons operation that inserts x at the start of the list/sequence xs.

For suffix array construction, sorting *suffixes* of sequences is a key concern. A sequence S of length n, denoted by $|S|$, contains n suffixes, with the i^{th} of those suffixes, denoted by $S[i \ldots]$, commencing at the i^{th} position in S and extending to the last position in S (inclusive), with the suffixes counted from zero to $n-1$. For example, if $S = \texttt{aabdabb}$, then $S[2\ldots] = \texttt{bdabb}$; and $S[6\ldots] = \texttt{b}$. By definition, $S[0\ldots] = S$ for all sequences S.

Most algorithms for suffix array construction including SA-IS, and algorithms for computing the BWT, further restrict the domain to *valid* sequences. A sequence is valid if it has a single occurrence of a unique symbol, called a *sentinel* and denoted by $\$$, that marks the end of the sequence and that is the smallest symbol in Σ. For example, the null byte serves this role for strings in C. The validity restriction is defined as follows:

Definition 2 (Valid Sequence)

$$\textbf{valid}\, S \;:=\; \exists T.\ (S = (T \mathbin{@} [\$])) \wedge (\$ \notin (\mathit{set}\, T)),$$

where $xs \mathbin{@} ys$ is the concatenation of two lists/sequences xs and ys, and $\mathit{set}\, xs$ turns the list/sequence xs into a set containing all element of xs.

While this domain restriction might appear at first to limit the applicability of the algorithms that will be introduced shortly, any finite and ordered alphabet Σ can be augmented so that any sequence $S \in \Sigma^*$ is mapped to a valid sequence.

2.3 Suffix Arrays

The suffix array SA of a sequence S is an array of all the suffixes of S in sorted order with respect to $<_{lex}$, with suffixes economically represented by their locations in S. Hence, the suffix array SA of a sequence S is a permutation of the $|S|$ values $[0 \ldots < |S|]$, where $[m \ldots < n]$ is the sequence of natural numbers from m up to but not including n. Moreover, for all $0 \le i < j < |S|$, we have $S[SA[i]\ldots] <_{lex} S[SA[j]\ldots]$. Formally, this axiomatic characterization of a suffix array is defined as follows:

Definition 3 (Suffix Array Predicate) Given a sequence S and a list of indices SA, SA is a suffix array of S, denoted by $\textbf{sa}\, SA$, iff

$$SA \sim [0 \ldots < |S|] \;\wedge\; \textbf{sorted}_{<_{lex}}\, (\mathit{map}\, (\lambda i.\, S[i\ldots])\, SA),$$

where $xs \sim ys$ means that xs is a permutation of ys and $\textbf{sorted}_{<_{lex}}$ is a predicate that checks if a list is sorted w.r.t. lexicographical order ($<_{lex}$).

Note that for any sequence, all of its suffixes have different lengths, meaning that there cannot be ties in the suffix array ordering, and hence, each sequence only has one suffix array. For example, the suffix array derived from the sequence $S = \texttt{aabdabb}$ is given by $SA = [0, 4, 1, 6, 5, 2, 3]$, corresponding to the ordered suffixes $[\texttt{aabdabb}, \texttt{abb}, \texttt{abdabb}, \texttt{b}, \texttt{bb}, \texttt{bdabb}, \texttt{dabb}]$.

The ordering in the suffix array SA for a sequence S makes it particularly useful for searching S for the existence of patterns. This is possible because in SA all of the suffixes of S that share any particular common prefix are located in a contiguous block. Thus, the search in S for a target pattern X mirrors a search in SA for a block of

```
function $simple(S)$
    $suffixes \leftarrow map\ (\lambda i.\, S[i \ldots])\ [0 \ldots < |S|]$
    $suffixes \leftarrow sort_{<_{lex}}\ suffixes$
    $SA \leftarrow map\ (suffix\text{-}id\ \ S)\ suffixes$
    return $SA$
end function
```

Fig. 1 Pseudocode for a simple suffix array construction algorithm, with S the input sequence and SA the computed suffix array.

suffixes that all commence with X. With the suffixes in sorted order, their prefixes are as well, and binary search can be used. This means that locating all the instances of an m character pattern X in a static n character text S can be achieved in $\mathcal{O}(m \log n)$ time, once the suffix array SA has been constructed. Note that in most situations the pattern is significantly smaller than the text, that is, $m \ll n$, and hence the search cost will be sublinear in n.

This relationship with indexed pattern search means that suffix array construction is an important problem, and a range of ways of constructing the suffix array of a sequence have been invented. One obvious option is to explicitly compute all of the suffixes, and then apply a string sorting algorithm to them. This approach, sketched in Figure 1, has the benefit of being straightforward. However, it also has a high worst-case execution time cost, requiring $\mathcal{O}(n^2 \log n)$ time, with each of the $\mathcal{O}(n \log n)$ string comparisons needed by a comparisons-optimal sorting algorithm potentially requiring $\mathcal{O}(n)$ time when processing highly repetitive sequences.

More efficient algorithms utilize alternative strategies that depend on the underlying features of suffixes to reduce the number of string comparisons. Puglisi et al. [17] provide an extensive survey of suffix array construction algorithms, categorizing each according to its strategy. The SA-IS algorithm [5, 6] that is described in Section 3 falls into the *recursive* category. Algorithms in this category adopt a common paradigm summarized as:

1. For the input sequence S, identify some subset of suffixes P that determine limits on the possible ordering of the other suffixes.
2. Construct a sequence S' whose suffixes correspond to the suffixes in P.
3. Recursively construct the suffix array of S' to obtain the relative order of the suffixes in P.
4. Sort the suffixes of S based on the ordered suffixes in P and other sequence/suffix properties.

That last step is called *induced sorting* and is an approach also used in several other suffix array construction algorithms [4, 18, 19]. Details of the SA-IS algorithm, the main focus of this paper, are provided in Section 3.

2.4 Isabelle/HOL

Isabelle/HOL [14] is an interactive theorem prover for classical higher-order logic based on Church's simply-typed lambda calculus. Internally, the system is built on an inference kernel that includes a small set of rules that are used to construct theorems that are established through formal proofs, which are complex deductions that are ultimately machine-checked through a small set of underlying rules. This approach, called LCF due to its pioneering system [20], guarantees correctness as long as the inference kernel is correct. Isabelle/HOL comes with a rich set of formalized theories, among which are natural numbers, sets, and lists; together with a range of library functions for manipulating lists and sets; plus facilities for handling type classes and polymorphism.

Proofs in Isabelle/HOL can be written in a style called Isabelle/Isar that is close to that of mathematical textbooks. The user structures the key points of the proof, and then the system fills in the gaps using automatic proof methods. Isabelle/HOL also supports locales, which provide a method for defining local scopes in which constants are defined and assumptions about them may be made. Theorems can be proven in the context of a locale and can use the constants and depend on the assumptions of this locale. A locale can then be instantiated to concrete entities if it can be demonstrated that those entities fulfil the locale assumptions.

Specifications in HOL within Isabelle/HOL can be semi-automatically translated to SML [21], OCaml [22], Haskell [23] or Scala [24] programs via Isabelle/HOL's code extraction facilities [25]. The translation from Isabelle/HOL text to the target language is not purely syntactic. The HOL specifications are first lifted, either automatically or with the help of hints, to a higher-order rewrite system within the logic of Isabelle/HOL. The rewrite system is then translated to the target language. Specifically, it is translated to the subset of the target language that can be viewed as an implementation of a higher-order rewrite system. Assuming that the final translation step is implemented correctly, the semantics of Isabelle/HOL text, that is functions and definitions, will then be carried into the program in the target language.

3 Calculating Suffix Arrays

The SA-IS algorithm [5, 6] is a linear time procedure for constructing suffix arrays that is also very fast in practice. The central idea is that a core subset of suffixes is identified and recursively sorted, based on which the order of the other suffixes can be induced. We first introduce suffix properties that the SA-IS algorithm relies on and then describe the algorithm itself.

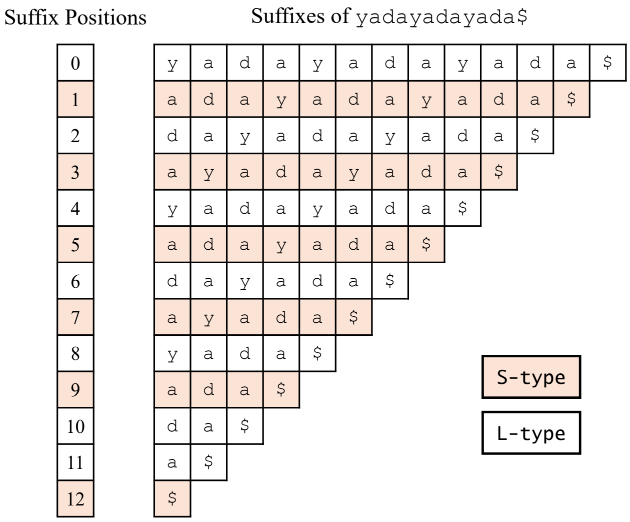
3.1 Required Suffix Properties

Central to the operation of SA-IS is the core property that suffixes of a valid sequence can be classified into two types.

Definition 4 (Suffix Type) For any valid sequence S and position i such that $i < |S|$:

- $S[i\ldots]$ is an L-type suffix iff $S[i+1\ldots] <_{lex} S[i\ldots]$ and $S[i\ldots] \neq \$$,
- $S[i\ldots]$ is an S-type suffix if $S[i\ldots] <_{lex} S[i+1\ldots]$, and
- $S[i\ldots]$ is an S-type suffix if $S[i\ldots] = \$$.

From this definition, the last suffix is always an S-type suffix. Moreover, since a valid sequence can never be equal to its first suffix, a suffix must always be either an



Suffix Positions	Suffixes of yadayadayada$												
0	y	a	d	a	y	a	d	a	y	a	d	a	$
1	a	d	a	y	a	d	a	y	a	d	a	$	
2	d	a	y	a	d	a	y	a	d	a	$		
3	a	y	a	d	a	y	a	d	a	$			
4	y	a	d	a	y	a	d	a	$				
5	a	d	a	y	a	d	a	$					
6	d	a	y	a	d	a	$						
7	a	y	a	d	a	$							
8	y	a	d	a	$								
9	a	d	a	$									
10	d	a	$										
11	a	$											
12	$												

Fig. 2 The suffix types of the sequence $S =$ yadayadayada$. Shaded values correspond to S-type suffixes, each of which is lexicographically less than the suffix that occurs immediately after it. Unshaded values are L-type suffixes, each of which is lexicographically greater than the suffix that occurs immediately after it.

L-type or an S-type. For example, consider the valid sequence $S =$ aabdabb$. Since $S[3\ldots] =$ dabb$ is greater than $S[4\ldots] =$ abb$, we know that $S[3\ldots]$ is an L-type. On the other hand, $S[1\ldots] =$ abdabb$ is an S-type as it is less than $S[2\ldots] =$ bdabb$. The third case means that $S[7\ldots] =$ $ is an S-type. Figure 2 provides further examples of suffix types, now for the sequence = yadayadayada$, an example that will be used through the remainder of this section.

Several desirable properties follow from Definition 4, with the following being one of the most vital:

Theorem 1 (Lemma 2 from Ko and Aluru [4]) *Let S be a valid sequence and let i and j be natural numbers greater than or equal to 0 and less than $|S|$, such that $S[i] = S[j]$. Further, suppose $S[i\ldots]$ is an* L-type *suffix and $S[j\ldots]$ is an* S-type *suffix. Then, $S[i\ldots] <_{lex} S[j\ldots]$.*

This relationship means that it is possible to partition suffixes both by their starting symbol, similar to a single pass of bucket sort, and by their corresponding suffix type; with all of the L-type suffixes being at the beginning of each possible start symbol's partition, and all of the S-type suffixes being at the end of that same partition.

For example, consider the six suffixes in Figure 2 that start with the symbol a, at positions 1, 3, 5, 7, 9, and 11. Five of these are S-type suffixes, with the one at position 11 being the only L-type suffix. From Theorem 1, it immediately follows that the suffix at position 11 is less than the others. That is, position 11 must appear in the suffix array SA before any of the other five positions.

In operational terms, a key aspect of the suffix type is that they can be computed efficiently using an iterative scan from the end of the sequence to the beginning, applying these mutually exclusive rules:

Definition 5 (Suffix Types Decision Procedure) Let S be a valid sequence, and let $0 \leq i < |S|$. Then:

1. if $i = |S| - 1$ then $S[i \ldots]$ is an S-type suffix;
2. if $S[i] < S[i+1]$ then $S[i \ldots]$ is an S-type suffix;
3. if $S[i] > S[i+1]$ then $S[i \ldots]$ is an L-type suffix; and
4. if $S[i] = S[i+1]$ then $S[i \ldots]$ has the same suffix type as $S[i+1 \ldots]$.

Rules 1–3 follow from Definitions 1, 4, and Rule 4 follows from the following:

Theorem 2 (Lemma 1 from Ko and Aluru [4]) *For a valid sequence S and $i < |S| - 1$, if $S[i] = S[i+1]$, then $S[i \ldots]$ and $S[i+1 \ldots]$ have the same suffix type.*

With the suffix type decision procedure and other relevant suffix type properties, it is possible to sort the S-type suffixes in linear time if the order of the L-type suffixes is known, and vice versa [4]. This approach, called *induced sorting*, is incorporated, and improved upon, in the SA-IS algorithm. Only a subset of the S-type suffixes needs to be sorted, and that subset can be guaranteed to contain at most half of the original suffixes. The suffixes making up this subset are called LMS-type suffixes and are defined as follows:

Definition 6 (Definition 2.1 from Nong et al. [5]) For a valid sequence S and $i > 0$, the suffix $S[i \ldots]$ is an LMS-type iff $S[i \ldots]$ is an S-type and $S[i-1 \ldots]$ is an L-type.

Using the earlier example, where $S =$ `aabdabb$`, we know $S[4 \ldots] =$ `abb$` is an LMS-type, since $S[4 \ldots]$ is an S-type suffix, and $S[3 \ldots] =$ `dabb$` is an L-type suffix.

The SA-IS algorithm sorts the LMS-type suffixes by mapping them to a reduced sequence and then recursively constructing a suffix array for the reduced sequence. To construct the required mapping, the LMS-type suffixes need to be roughly sorted, which is also achieved using an induced sorting approach. However, rather than sorting complete suffixes, this step sorts prefixes and substrings of the suffixes, using a bespoke substring comparison ordering. The prefixes, called LMS-prefixes, and substrings, called LMS-substrings, of suffixes are defined as follows:

Definition 7 (LMS-prefix) For a valid sequence S and $0 \leq i < |S|$, let $i \leq j < |S|$ be the next LMS-type location in S at position i or beyond, or $|S| - 1$ if no such LMS-type occurs. Then the subsequence of S starting from location i up to j, denoted by $S[i \ldots < j]$, is the i^{th} LMS-prefix.

Definition 8 (LMS-substring, Definition 2.2 from Nong et al. [5]) For a valid sequence S and $0 \leq i < |S|$, let $i < j < |S|$ be the next LMS-type location in S strictly beyond i, or $|S| - 1$ if no such LMS-type occurs. Then the subsequence $S[i \ldots < j]$ is the i^{th} LMS-substring.

Note that an LMS-prefix is either equal to the corresponding LMS-substring, or a one-symbol prefix of it; and that the LMS-prefix of a non-LMS-type suffix is the same as its LMS-substring. For example, for the set of suffixes shown in Figure 2, the zeroth LMS-prefix of S is ya, and the first is a, while the zeroth LMS-substring of S is ya, and the first is ada.

The substring ordering comparisons are defined as follows:

Definition 9 (Substring Ordering, Definition 2.3 from Nong et al. [5]) Let S be a valid sequence and suppose that the comparison is between a pair of LMS-prefixes or a pair of LMS-substrings that start at positions i and j, respectively. Then the comparison is carried out in a pairwise fashion, with elements being compared first w.r.t. to the alphabet ordering and then suffix types are compared. That is, the k^{th} pairwise comparison first compares $S[i + k]$ with $S[j + k]$. If $S[i + k] \neq S[j + k]$, then the substring comparison terminates with the result of $S[i + k] < S[j + k]$. Otherwise, it compares the suffix types of S at positions $i + k$ and $j + k$ with L-types defined to be less than S-types. If these are also equal, then the comparison proceeds to the next pair.

The substring ordering is defined in this way to ensure that an LMS-prefix of an L-type suffix is less than an LMS-prefix of an S-type suffix if the two suffixes start with the same symbol. The same also holds for LMS-substrings.

3.2 The SA-IS Algorithm

The SA-IS algorithm is presented in Figure 3, adapted from Nong et al. [5, Figure 1]. It constructs the suffix array for a valid sequence S by recursively constructing the suffix array of a smaller sequence S' that is derived from the LMS-type suffixes. The ordering of the LMS-type suffixes of S is obtained from the suffix array of S'. The ordering of the other suffixes of S is then induced from the sorted LMS-type suffixes. This high-level description indicates how the SA-IS algorithm depends on two core components: reducing the problem from S to a smaller sequence S' and induced sorting. These are explained in more detail in the following subsections. Both the sequence reduction and induced sorting run in linear time and $|S'|$ is always guaranteed to be at most $\lfloor |S|/2 \rfloor$, as this is the maximum number of LMS-type suffixes that S can have. For $|S| = n$, this results in the recurrence relation

$$T(n) = T(\lfloor n/2 \rfloor) + \Theta(n)\,,$$

whose solution is $\Theta(n)$. Hence, the SA-IS algorithm runs in linear time (and linear space) for finite alphabets.

```
function $SA\text{-}IS(S)$
    $ST$ : $[0 \ldots < n]$ of {L-type, S-type}
    $B$ : $[0 \ldots < |\Sigma|]$ of $\mathbb{N}$
    $P$ : $[0 \ldots < n']$ of $\mathbb{N}$
    $S'$ : $[0 \ldots < n']$ of $\mathbb{N}$
    $SA'$ : $[0 \ldots < n']$ of $\mathbb{N}$
    $SA$: $[0 \ldots < n]$ of $\mathbb{N}$
    compute the buckets for $S$ for each symbol in $\Sigma$ and store these in $B$
    compute the suffix type for each of the suffixes of $S$, and store them into $ST$
    scan $ST$ for LMS-types, storing their indexes in $S$ into $P$, and setting $n'$
    induce-sort the $n'$ LMS-substrings using $P$, $ST$ and $B$
    name each LMS-substring in the LMS-type suffixes with its rank, to obtain $S'$
    if each symbol in $S'$ is unique then
        set $SA'$ to be the sorted LMS-types
    else
        recursively call $SA\text{-}IS(S')$ to obtain $SA'$
        map $SA'$ back to suffixes in $S$
    end if
    induce-sort suffixes from $SA'$ using $ST$ and $B$ into $SA$
    return $SA$
end function
```

Fig. 3 The SA-IS algorithm, adapted from Nong et al. [5, Figure 1], in which S is the input sequence, $n = |S|$ is the length of S, n' is the number of LMS-type suffixes that occur in S, B is an array whose elements point to symbol intervals/buckets over the suffix array that indicate where suffixes beginning with that symbol should occur in the suffix array, and SA is the computed suffix array for S.

3.2.1 Sequence Reduction

The input sequence S is reduced to a shorter surrogate sequence S' so that the LMS-type suffixes can be ordered. That requires that the suffixes of S' must correspond to the LMS-type suffixes of S and that the mapping must preserve the lexicographical ordering of the S' suffixes. This is achieved via the following observations:

1. From the definition of a suffix, the first occurring LMS-type suffix contains all the LMS-type suffixes.
2. Each LMS-type suffix can be mapped to a sequence of LMS-substrings of LMS-type suffixes, and this mapping is injective. An analysis of the form of an LMS-substring of an LMS-type suffix shows that this mapping is monotonic, that is, preserves the lexicographical ordering between LMS-type suffixes.
3. Since the LMS-substrings of the LMS-type suffixes can be sorted and there is a finite number of them, a bijection between LMS-substrings of LMS-type suffixes and an ordered alphabet Σ' can always be found. Moreover, the bijective mapping is monotonic, where the ordering on LMS-substrings is as described by Definition 9.

Using these observations, S' is constructed as follows:

1. Take the first LMS-type suffix of S, denoted by S_1.
2. Map S_1 to its sequence of LMS-substrings of LMS-type suffixes of S.

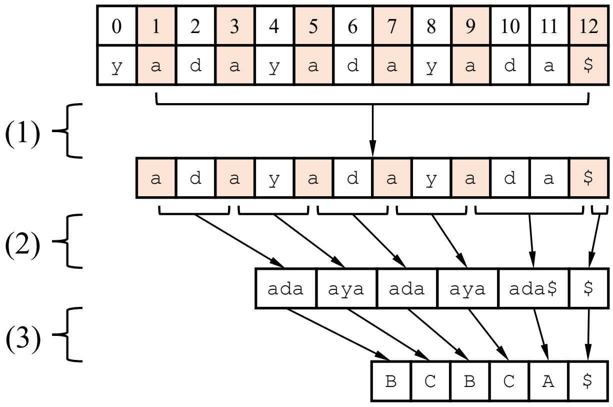


Fig. 4 The sequence $S =$ yadayadayada$ is reduced to a shorter surrogate $S' =$ BCBCA$ by: (1) taking the first LMS-type suffix $S_1 =$ adayadayada$; (2) mapping it to a sequence of LMS-substrings of LMS-type suffixes; and (3) mapping these LMS-substrings to their symbols in a new alphabet $\Sigma' = \{\$, \mathtt{A}, \mathtt{B}, \mathtt{C}\}$, where $ ↦ $, ada$ ↦ A, ada ↦ B and aya ↦ C. Locations of the LMS-types are shaded.

3. Map each of the LMS-substrings to the corresponding symbol in the new alphabet Σ'.

That is, the i^{th} LMS-type suffix of S is mapped to the i^{th} suffix of S', such that the lexicographical ordering is preserved. This is because using the same mapping described in observation (2) on other LMS-type suffixes of S will produce sequences that are suffixes of S'_1, such that the i^{th} LMS-type suffix of S is mapped to the i^{th} suffix of S'_1, which in turn means that it is mapped to the i^{th} suffix S'. Moreover, the mapping preserves ordering, meaning that computing the suffix array of S' is equivalent to sorting the LMS-type suffixes of S.

Figure 4 illustrates the reduction process applied to the sequence $S =$ yadayadayada$ shown earlier in Figure 2. The distinct LMS-substrings in sorted order are $, ada$, ada$ and aya, and each is mapped to a symbol in a new alphabet Σ'. In the example, the mapping is $ ↦ $, ada$ ↦ A, ada ↦ B and aya ↦ C respectively; with $\Sigma' = \{\$, \mathtt{A}, \mathtt{B}, \mathtt{C}\}$. With Σ' having been computed, the first LMS-type suffix, which is located at position 1, can now be transformed into S', by parsing it into the sequence [ada, aya, ada, aya, ada$, $], and then applying the mapping to generate $S' =$ BCBCA$.

While it is possible in extreme cases for the recursive reduction to proceed through S', then S'', and so on, until a sequence $S'^{\cdots\prime}$ containing a single element is reached, the recursive reduction can also terminate earlier if all the symbols in some reduced sequence are distinct — that is, if all of the LMS-substrings of the LMS-type suffixes of the previous sequence are distinct. The latter implies that they are strictly sorted, and hence, the LMS-type suffixes of the previous sequence are sorted, which is precisely what the reduced problem was designed to achieve.

3.2.2 Induced Sorting — Overview and Initialization

The induced sorting procedure inserts the L-type and S-type suffixes into the array in lexicographical order to produce the suffix array. This same procedure is also used to sort LMS-substrings of the sequence in connection with the ordering in Definition 9 to produce the approximate suffix array, which is used in the sequence reduction.

The induced sorting procedure can be in turn divided into three subroutines. The first, denoted by *insert_lms*, inserts the LMS-types into the array in sorted order, initializing what order the suffixes will be sorted with respect to. The second, denoted by *induce_l*, induced sorts the L-types based on the order of the LMS-types. The third, denoted by *induce_s*, induced sorts the S-types based on the order of the L-types. This, and the next two subsections, address these three subroutines.

All three subroutines operate similarly when inserting suffixes into the suffix array. They use a process that is intuitively like bucket sort. The array is partitioned into buckets where each bucket corresponds to one symbol in the alphabet, with the buckets ordered according to the alphabet order. With this partitioning, a suffix that begins with the symbol x is then only inserted into the bucket that corresponds to the symbol x. For example, all sequences starting with the symbol a will occur in one contiguous block in the array, and this block will occur before the block corresponding to the symbol b.

Each bucket is further partitioned into two sections, where the first section is for L-types and the second section is for S-types. This is because, from Theorem 1, L-type suffixes are lexicographically less than S-type suffixes if they start with the same symbol. This is also the case for LMS-prefixes and LMS-substrings, according to the subsequence ordering in Definition 9. Due to this, both *insert_lms* and *induce_s* fill buckets in back-to-front order, while *induce_l* fills buckets in front-to-back order.

To ensure that the bucket creation and bucketing process is efficient, the alphabet Σ must be finite and have a monotonic mapping to a subset of the natural numbers, where the sentinel \$, the smallest symbol in the alphabet, maps to 0. That way, the buckets can be computed by doing two linear passes over the sequence S, where the first computes the frequency of each symbol in S, and the second computes the cumulative sum of those frequencies. The resulting buckets can then be stored in an array B that is treated like a key-value store, with the keys being natural numbers that correspond to symbols in the alphabet and the values either being the start of the buckets, in the case of *induce_l*, or the end of the buckets, in the case for *insert_lms* and *induce_s*. The B array also indicates where the next free location in the suffix array to do an insert and this is updated accordingly by the induced sorting subroutines.

The first subroutine *insert_lms* takes an array of LMS-types as input and returns an array with those LMS-types inserted into their designated buckets. Moreover, the relative order of the LMS-types starting with the same symbol is maintained, as specified by the input array, within each bucket. That is, if the LMS-types that begin with the symbol x are listed in a particular order in the input array, these will occur in that same order in the bucket corresponding to the symbol x. This means that when induced sorting suffixes, *insert_lms* requires the LMS-type suffixes to be listed in sorted order in the input array so that it can insert these into the suffix array in sorted order (w.r.t. lexicographical order). When induced sorting LMS-substrings, *insert_lms* inserts LMS-prefixes of LMS-type suffixes into the approximate suffix array in sorted order (w.r.t.Definition 9). In this case, the LMS-prefixes can be listed in the input array in any order as an LMS-prefix of an LMS-type suffix consists only of the first symbol of

```
function $induce\_l(S,\ ST,\ B,\ SA)$
    $i \leftarrow 0$
    while $i < |S|$ do
        $j \leftarrow SA[i] - 1$
        if $ST[j]$ is an L-type then
            $SA[B[S[j]]] \leftarrow j$
            $B[S[j]] \leftarrow B[S[j]] + 1$
        end if
        $i \leftarrow i + 1$
    end while
    return $SA$
end function
```

Fig. 5 The induced sorting subroutine for L-types, where S is the sequence; ST is an array containing the suffix types; B is an array of pointers to the next free position in a bucket towards the front that is initialized to point to the start of each bucket, that is, $B[c] = \|\{k \mid k < |S| \wedge S[k] < c\}\|$ for all $c \in \Sigma$; and SA is the suffix array.

that suffix, thus meaning that the LMS-prefixes starting with the same symbol in the input array will always be listed in sorted order.

3.2.3 Induced Sorting — L-types

The second subroutine *induce_l*, takes the output of *insert_lms* and inserts all the L-types into that array, placing L-types at the front of their respective buckets. When induced sorting suffixes, the L-type suffixes will be in lexicographical order. When induced sorting LMS-substrings, the LMS-prefixes of L-type suffixes will be sorted, and hence, the LMS-substrings of L-type suffixes will be sorted, as LMS-prefixes of non-LMS-type suffixes are also LMS-substrings.

The *induce_l* subroutine, shown in Figure 5, iterates over the array from the beginning to the end, inserting the current suffix's predecessor into its corresponding bucket if the predecessor is an L-type. To ensure that an L-type is inserted in the correct location in its bucket, the B array is used to store pointers to the next free position for each bucket. Initially, these are set to point to the start of each bucket, that is, for the symbol $c \in \Sigma$, $B[c] = \|\{k \mid k < |S| \wedge S[k] < c\}\|$, where $\|A\|$ is the cardinality of the set A. After an L-type is inserted into a bucket, the pointer corresponding to that bucket is incremented, pointing to the next free position.

Intuitively, the subroutine maintains the invariant that everything inserted so far is sorted. This is because scanning from the beginning to the end inserts the predecessors of smaller suffixes first. These will always be smaller than predecessors of subsequent suffixes that start with the same symbol, as both lexicographical order and Definition 9 have the property that the sequence $x \mathbin{\#} xs$ is less than the sequence $x \mathbin{\#} ys$ iff xs is less than ys.

Figure 6 demonstrates the induced sorting for L-type suffixes of the sequence $S =$ yadayadayada$. On the left are the LMS-type suffixes that have been inserted into their buckets in lexicographical order, and this is the result of the previous subroutine for inserting LMS-types. To the right of these are the L-type suffixes that are inserted

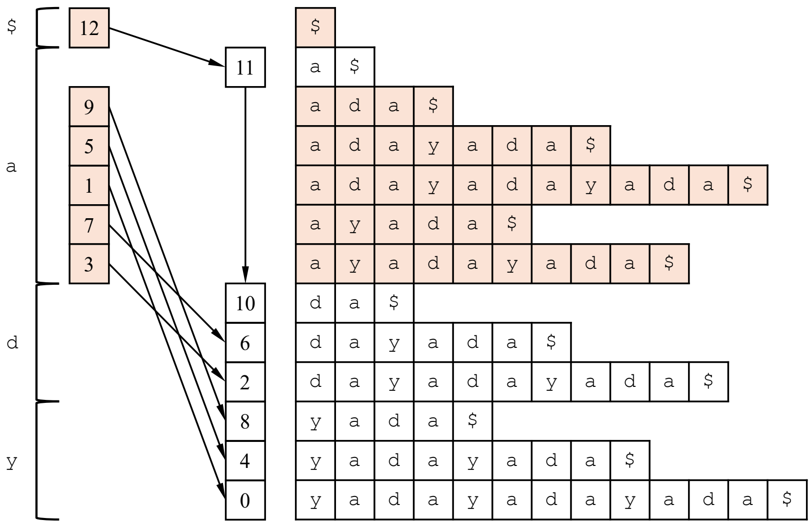


Fig. 6 Induced sorting the L-type suffixes from the LMS-type suffixes of the sequence S = yadayadayada$. The procedure occurs from top to bottom, starting from $S[12\ldots]$, then to $S[11\ldots]$, $S[9\ldots]$, and so on, finishing with $S[0\ldots]$. The LMS-type suffixes are shaded, the arrows indicate where an L-type suffix is inserted and from which suffix this was from, and the labeled brackets on the side signify the buckets, managed by the array B.

by iterating over the array from lexicographic smallest to lexicographic largest (top to bottom in the diagram).

For example, starting from $S[12\ldots]$ = $, $S[11\ldots]$ = a$ is inserted, as indicated by the arrow. Going to the next suffix in the array, which happens to be $S[11\ldots]$, $S[10\ldots]$ = da$ is inserted. Then going to the next, which is $S[9\ldots]$ = ada$, $S[8\ldots]$ = yada$ is inserted. This continues until the last suffix in the array is reached.

Note how each insertion maintains the sorted order. For example, $S[10\ldots]$ = da$ occurs before $S[6\ldots]$ = dayada$, as it is lexicographically smaller. This occurs because $S[11\ldots]$ = a$ is both lexicographically smaller and it is encountered before $S[7\ldots]$ = ayada$.

3.2.4 Induced Sorting — S-types

The third subroutine *induce_s*, takes the output of *induce_l* and inserts all the S-types into that array, placing them at the end of their respective buckets in sorted order. If the LMS-substrings of L-type suffixes in the output of *induce_l* are lexicographically sorted, then all the LMS-substrings of the sequence will be in sorted order, once the *induce_s* subroutine has finished executing. Hence, if the L-type suffixes in the output of *induce_l* are lexicographically sorted, the output of *induce_s* will be the suffix array.

The *induce_s* subroutine, shown in Figure 7, is very similar to the *induce_l* subroutine. However, it instead iterates over the array from the end to the beginning, inserting the current suffix's predecessor into the next free position of its respective bucket if the predecessor is an S-type. The next free position is similarly maintained by the array B. However, in this subroutine, they are initialized to the end of each

```
function $induce\_s(S, ST, B, SA)$
    $i \leftarrow |S| - 1$
    while $i > 0$ do
        $j \leftarrow SA[i] - 1$
        if $ST[j]$ is an S-type then
            $SA[B[S[j] - 1]] \leftarrow j$
            $B[S[j]] \leftarrow B[S[j]] - 1$
        end if
        $i \leftarrow i - 1$
    end while
    return $SA$
end function
```

Fig. 7 Pseudocode for the induced sorting subroutine for S-types, where S is the sequence; ST is an array containing the suffix types; B is an array of pointers to the next free position in a bucket that is initialized to point at the end of each bucket (exclusive), that is, $B[c] = \|\{k \mid k < |S| \wedge S[k] \leq c\}\|$ for all $c \in \Sigma$; and SA is the suffix array.

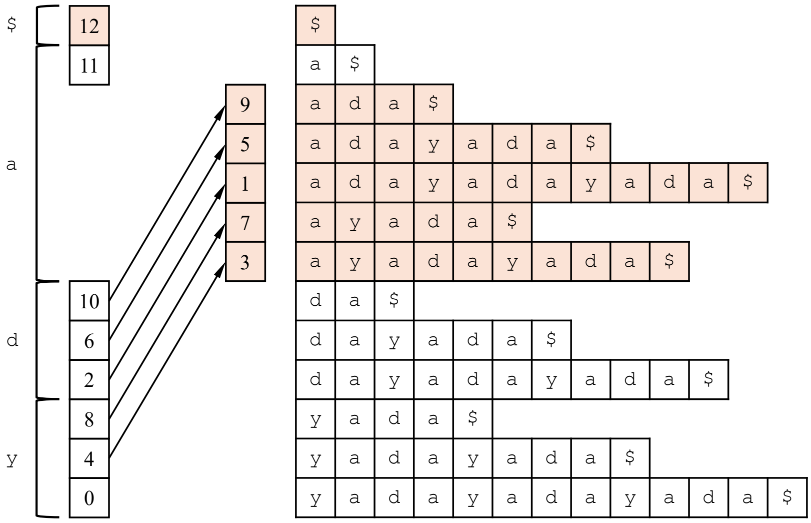


Fig. 8 Induced sorting the S-type suffixes from the L-type suffixes of the sequence S = yadayadayada$. The procedure proceeds from bottom to top, starting from $S[0\ldots]$, then to $S[4\ldots]$, $S[8\ldots]$, and so on, finishing with $S[12\ldots]$. The LMS-type suffixes are shaded, the arrows indicate where an S-type suffix is inserted and which suffix it came from, and the labeled brackets on the side scope the buckets.

bucket, that is, for the symbol $c \in \Sigma$, $B[c] = \|\{k \mid k < |S| \wedge S[k] \leq c\}\|$, and are decremented after each insertion.

The rationale behind *induce_s* is similar to that of *induce_l*. However, instead of inserting from smallest to largest, the opposite happens. One major difference though is that the suffix corresponding to the sentinel, which is also an S-type, is not inserted by this subroutine. It is the LMS-type insertion subroutine that inserts this into its bucket, and the *induce_l* and the *induce_s* subroutines do not modify its placement.

The example shown in Figure 8, demonstrates the induced sorting subroutine for S-type suffixes of the sequence $S =$ `yadayadayada$`. On the left are the L-type suffixes, which are inserted in sorted order by the induced sorting subroutine for L-types. The array is scanned from the largest to smallest lexicographically, from bottom to top in Figure 8, inserting the S-type suffixes, which are shaded.

For example, $S[0\ldots] =$ `yadayadayada$` is checked first, however, since it has no predecessor, nothing is inserted. The next suffix checked is $S[4\ldots] =$ `yadayada$` and from this, $S[3\ldots] =$ `ayadayada$` is inserted, as indicated by the arrow. The next suffix is $S[8\ldots] =$ `yada$`, and so $S[7\ldots] =$ `ayada$` is inserted. This continues until the start of the array is reached. As already noted, the placement of the last suffix $S[12\ldots] =$ `$` is neither modified by the induced sorting of L-types nor that of S-types. This is indicated by being on the left in both Figure 6 and Figure 8, where being on the left signifies not being inserted by the subroutine.

As with the induced sorting of the L-types, the induced sorting of the S-types maintains the sorted order. For example, $S[3\ldots] =$ `ayadayada$` occurs after $S[7\ldots] =$ `ayada$`, meaning that it was inserted before, and hence is lexicographically greater. This is because $S[4\ldots] =$ `yadayada$` is lexicographically greater than and occurs after $S[8\ldots] =$ `yada$`.

3.3 Summary

The SA-IS algorithm is a linear time algorithm for constructing suffix arrays that is also very fast in practice. It can also be used to construct the BWT in linear time, as the BWT can be constructed from a suffix array in linear time. The SA-IS algorithm has features that are also present in several other suffix array construction algorithms. These include categorizing suffixes into different types and induced sorting. If these were verified, both the implementations and proofs could be reused (or at least some parts) in the implementations and verifications of the suffix array construction algorithms that also have these features [4, 18, 19].

This section has described the SA-IS algorithm and provided intuitive explanations for its various subroutines. These informal descriptions have been based more on rhetoric than on careful logic. What is required now — to provide absolute certainty in operational settings — is a formal specification and proof of correctness. We do just that in the next sections.

4 An Outline of the Formal Development

This section provides a brief overview of our formalization and total correctness proof of the SA-IS algorithm in the Isabelle/HOL theorem prover. Later sections dive into more detail and provide key insights and challenges. The complete Isabelle/HOL formalization is publicly available in Isabelle's Archive of Formal Proofs [26].

4.1 Formally Specifying and Embedding the SA-IS Algorithm

To formally verify the SA-IS algorithm, we first had to define the functional correctness specification and embed the algorithm in Isabelle/HOL. Both were straightforward to

```
function sais :: “nat list ⇒ nat list”
  where
“sais [] = []” |
“sais [x] = [0]” |
“sais (a # b # xs) =
  ( let
     S = a # b # xs;
     —< Compute the suffix types >
     ST = get_types S;
     —< Scan and extract LMS-type >
     P = extract_lms ST [0 ...< length S];
     —< Induced sort the LMS-substrings >
     SA = induce id S ST P ;
     —< Extract and name the sorted LMS-substrings of LMS-type suffixes >
     Q = extract_lms ST SA ;
     names = rename_mapping S ST Q ;
     —< Construct the reduced sequence >
     S′ = rename_string P names ;
     —< If distinct then Q is the list of sorted LMS-type suffixes, otherwise
         compute the suffix array of S′ and map the suffixes back to S >
     R = (if distinct S′ then Q else order_lms P (sais S′))
   —< Induced sort the suffixes >
   in induce id S ST R )”
```

Fig. 9 The Isabelle/HOL embedding of the SA-IS algorithm.

define given that index and suffix locations can be modeled as natural numbers and sequences and arrays can be modeled as lists. Moreover, since the elements of these sequences and arrays are drawn from finite, linearly ordered alphabets, these are also modeled as natural numbers, and hence, only lists of natural numbers are used in the formalization (at least for the core part of the formalization).

The functional correctness specification consists of two parts: termination and soundness of the result of the suffix array construction. Termination is handled automatically by Isabelle/HOL, as Isabelle/HOL automatically generates domain predicates, which indicate what values the function terminates for, whenever a function is defined. We utilize Isabelle/HOL embeddings of Definitions 2 and 3, which are just transliterations of these definitions, to define the soundness of suffix array construction as:

Definition 10 (Soundness of Suffix Array Construction on Valid Lists of Naturals) A suffix array construction function f, that takes as input a list of natural numbers, is sound iff

$$\forall S.\ \mathbf{valid}\, S \implies \mathbf{sa}\,(f\ S)\,.$$

```
fun repeatatm :: “nat ⇒ ('a ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b ⇒ 'a) ⇒ 'a ⇒ 'b ⇒ 'a”
  where
“repeatatm 0 _ _ acc _ = acc” |
“repeatatm (Suc n) f g acc obsv =
  (if f acc obsv then acc else repeatatm n f g (g acc obsv) obsv)”

definition repeat :: “nat ⇒ ('a ⇒ 'b ⇒ 'a) ⇒ 'a ⇒ 'b ⇒ 'a”
  where
“repeat n f a b = repeatatm n (λx y. False) f a b”
```

Fig. 10 General loop combinators in Isabelle/HOL for repeated application of a function to an accumulator for up to n many iterations, where *repeatatm* also permits early termination.

To keep the underlying theories general, that is, not specific to the SA-IS algorithm, we define an Isabelle/HOL locale for suffix array construction with Definition 10 as the locale assumption, thereby enabling us to rely on existing theories and ease structuring of new theories in a general manner to maximize future reuse.

The SA-IS algorithm is implemented as a recursive function in Isabelle/HOL. This function, *sais*, presented in Figure 9, is largely a transliteration of the algorithm description, shown in Figure 3. It has a few differences, such as the addition of the base cases for the recursion, shifting the bucket pointer array B definition from the top level function *sais* into the induced sorting function *induce*, and the introduction of additional lists, Q and R shown in Figure 9, that are used to store the locations of LMS-substrings suffixes after they have been sorted w.r.t. the substring ordering and w.r.t. lexicographical ordering, respectively. In addition to this, functions, such as *induce_l* and *induce_s* that induced sort L-types and S-types, are transformed into functional programs using a loop combinator, developed in previous work [27]. The loop combinator *repeat*, shown in Figure 10, simply repeatedly applies a function f to an accumulator *acc* with an observer argument *obsv* n many times. Note that *repeat* is defined using a more general loop combinator, *repeatatm*, that permits early termination. With *repeat*, functions like *induce_l* can then be implemented as step functions that are just the bodies of the loop, as shown in Figure 11.

4.2 Formally Verifying the Correctness of SA-IS

The verification of the SA-IS algorithm necessitated formalizing a substantial amount of background theory. This included general properties about orders, lists, and monotonic functions that are useful additions to the existing theories found in Isabelle/HOL's standard library. The background also included properties that are specific to the context of verifying the SA-IS algorithm, such as an alternative lexicographic order, which as we will see, was convenient to use for formalization purposes. Section 5 provides an overview of such key definitions and theorems that are specific to SA-IS. This also included suffix type properties that were informally presented in Section 3.1.

The verification of the SA-IS algorithm itself involved proving that the Isabelle/HOL embedding of the algorithm, *sais*, terminates, and that it is sound with

```
fun induce_l_step ::
    "nat list × nat list × nat ⇒
    (('a :: {linorder, order_bot}) ⇒ nat) × 'a list × SL_types list ⇒
    nat list × nat list × nat"
  where
"induce_l_step (B, SA, i) (α, T, ST) =
  (if SA ! i < length T
   then
     (case SA ! i of
        Suc j ⇒
          (case ST ! j of
             L_type ⇒
               (let k = α (T ! j);
                    l = B ! k
                in (B[k := Suc (B ! k)], SA[l := j], Suc i))
           | _ ⇒ (B, SA, Suc i))
      | _ ⇒ (B, SA, Suc i))
   else (B, SA, Suc i))"
```

Fig. 11 The step function that implements one iteration of the *induce_l* function, that is, the loop body of Figure 5.

respect to the formal specification. These are presented in more detail in Section 6. Note that termination had to be proved manually because this depends on suffix type properties. The soundness proof, which is the major component of the verification, was proved in two steps, as displayed in Figure 12. The first step is the definition and proof of soundness w.r.t. Definition 10 for an abstract version of the SA-IS algorithm. The second step is an equivalence proof between the abstract version and *sais*.

We defined and verified an abstract embedding, denoted by *abs_sais*, to simplify the verification. The embeddings simplify the verification because it forgoes the efficient suffix type pre-computation (Definition 5) and queries and instead uses the mathematical suffix type definition. Due to this, the embeddings of induced sorting and reduced alphabet creation, which are the core procedures in the SA-IS algorithm, were simplified. Note that even with these simplifications, this section of the verification required most of the proof effort.

We verified that the SA-IS algorithm embedding, *sais*, is correct by proving that this is equivalent to the abstract embedding *abs_sais* for valid lists. The proof is not particularly complex, relying mostly on induction on the arguments of *sais*/*abs_sais*, term rewriting, and on the correctness of the suffix type pre-computation (Definition 5), denoted by *get_types* in the formalization. The correctness theorem for *get_types* itself was not too difficult to prove as it closely resembles the intuitive argument described in Section 3.1, and hence, was largely an exercise in translating these arguments into Isabelle/HOL syntax.

As mentioned in Section 2.2, algorithms, such as SA-IS, assume that the input sequence is valid. We demonstrate in Section 6.5 that the validity assumption can be removed by providing a lifting function that given any alphabet and sequence

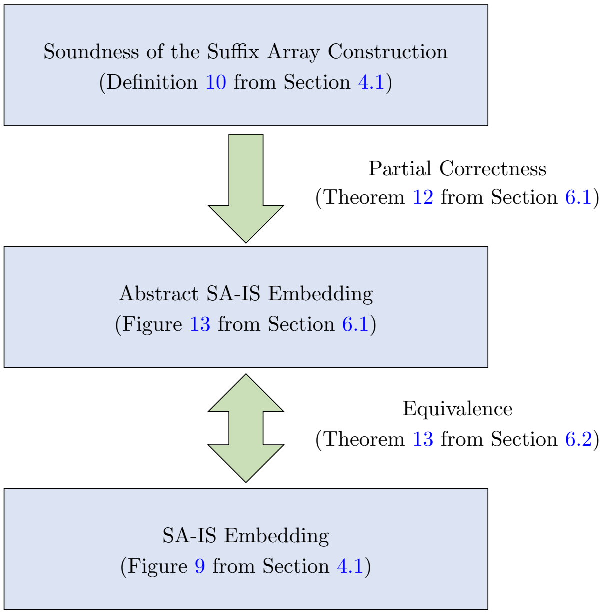

Soundness of the Suffix Array Construction
(Definition 10 from Section 4.1)
Partial Correctness
(Theorem 12 from Section 6.1)
Abstract SA-IS Embedding
(Figure 13 from Section 6.1)
Equivalence
(Theorem 13 from Section 6.2)
SA-IS Embedding
(Figure 9 from Section 4.1)

Fig. 12 A visualization of the approach taken to prove soundness w.r.t. to the formal specification (Definition 10 from Section 4.1) for the SA-IS algorithm. Boxes represent specifications and embeddings and arrows represent proofs, with double-ended arrows meaning equivalence.

augments them into ones that obey the validity condition. The resulting suffix array on the augmented sequence can then be easily transformed into a suffix array of the original sequence. This demonstrates that such augmentation and transformation is always possible, and hence, results in a mechanized proof of total correctness for a more general algorithm than the one established in the original paper. Namely, an algorithm that works for any list drawn from a linearly ordered alphabet with a bottom element.

4.3 Haskell Extraction

As a final step, we demonstrate that our algorithm can be extracted to Haskell using Isabelle/HOL's built-in facilities with minimal effort. The Haskell implementation, albeit inefficiently, can be executed to construct suffix arrays for valid lists of natural numbers, thus demonstrating that what we have verified can be refined to an executable implementation. The implementation is inefficient because it still uses lists and natural numbers rather than arrays and machine words. This means that indexing and arithmetic operations are not executed in constant time, thus making the implementation inefficient. Note that this code extraction step is merely a sanity check and is not meant to produce an efficient implementation. While the latter is important and will be investigated in future work, it is an orthogonal problem to the aim of this paper, which is to verify the correctness of the SA-IS algorithm.

5 Formalizing Underlying Properties

To prove the correctness of the SA-IS algorithm, the underlying theory about sequences and suffixes from Sections 2.2 and 3.1, respectively, is first formalized.

Formalizing properties about sequences, that is, lists, is straightforward because there is an existing definition of lexicographical order on lists in Isabelle/HOL. Moreover, this definition satisfies the assumptions of the total order locale, called `linorder`. This makes it possible to instantiate the locale with this definition of lexicographical order, and hence, provides access to numerous useful theorems about linear orders instantiated with lexicographic order. The validity predicate can essentially be defined as shown in Definition 2.

Formalizing suffix properties, however, is much more difficult. This is because both the suffix type and the subsequence comparison functions (Definitions 4 and 9, respectively), as defined by Nong et al. [5], are partial. Neither function is defined for sequences that do not satisfy the validity requirement. Moreover, the suffix type is not defined for indices that are out of bounds. Partial functions are not ideal for formal reasoning, as various theorems that are automatically generated for total functions either must be proved manually or are no longer true. To solve this, both the suffix type and subsequence comparison definitions need to be modified in such a way that makes them total while still preserving their current behavior.

Another issue is that the subsequence comparison (Definition 9) is only defined when both its arguments are a subsequence of the same sequence. That is, a subsequence of a sequence S and a subsequence of a sequence T are incomparable if $S \neq T$ according to Definition 9. Hence, the subsequence comparison function is a ternary operator. Since most of the proofs depend on properties about total orders, it makes sense to try to show that the subsequence comparison satisfies the `linorder` locale assumptions, and hence, is an instance of a total order. However, this means that the subsequence comparison needs to be transformed into a binary comparison between arbitrary sequences.

Note that alternatively, all the necessary total order properties for Definition 9 could be manually proved, however, this would complicate the verification as well as future-proof maintenance. Moreover, as the formalization extensively relies on properties from the `linorder` locale, including some that are not in the existing Isabelle/HOL library, adapting the formalization to use an instance of the `linorder` locale meant that various new reusable properties could be added to the context of this locale, rather than proving them for a custom comparison function that is specific to the suffix array context.

The solution to both issues of partiality of definitions and that Definition 9 is a ternary operator is to replace the lexicographical comparison in the suffix type definition and the subsequence comparison (Definition 9) with the following comparison function.

Definition 11 (Alternative Lexicographical Order)

$$[] <_{lex'} (y \# ys) := \mathsf{false} \tag{2a}$$

$$(x \# xs) <_{lex'} [] := \mathsf{true} \tag{2b}$$

$$[] <_{lex'} [] \quad := \mathsf{false} \tag{2c}$$

$$(x \# xs) <_{lex'} (y \# ys) \quad := (x \leq y) \wedge (x = y \longrightarrow xs <_{lex'} ys)\,. \tag{2d}$$

This new comparison function is total, which is demonstrated in the formalization by proving that it is an instance of the `linorder` locale. The rest of this section describes this new comparison function and its implication on the background theory, specifically, why it can replace the standard lexicographical order used in the suffix type definition and why it can replace the subsequence comparison (Definition 9).

Definition 11 ($<_{lex'}$) is almost the same as the classical lexicographical order ($<_{lex}$), except that it differs when the first list is a prefix of the second. In this case, rather than returning true, as the classical version would, it returns false. This results in scenarios such as a being greater than aa. Interestingly, both lexicographical comparison functions produce the same result when the lists being compared are both valid. Since this is always the case when $<_{lex}$ is used in the SA-IS algorithm, it can be replaced by $<_{lex'}$ without altering the outcome of the algorithm. Moreover, $<_{lex'}$ satisfies all the requirements for a total order, and hence, all the theorems in the `linorder` locale are also applicable to $<_{lex'}$.

An important property about $<_{lex'}$ is that any non-empty list is less than the empty list, which means that the list $S = \$$ is less than the empty list. This behavior is exactly what is required by the suffix type definition, as described earlier in Definition 4, and so the suffix type can be defined using $<_{lex'}$, removing the valid list requirement and the special case for $S = \$$.

Definition 12 (List Type) A list S is an L-type iff $S[1\ldots] <_{lex'} S$, and is an S-type iff $S <_{lex'} S[1\ldots]$.

The above definition is referred to as *list type* rather than suffix type because it no longer depends on the suffix position argument. Note that suffix type and list type are interchangeable as the suffix type of list S at position i is the same as the list type of the suffix $S[i\ldots]$, and the list type of a list S is the same as the suffix type of S at position 0.

The list type definition results in two useful properties. The first is the generalization of theorems, such as Theorem 1 and Theorem 2. The second is that LMS-prefixes and LMS-substrings have the same list type as their corresponding suffix. These two properties can then be used to show that for every input that the subsequence comparison (Definition 9) is defined for, $<_{lex'}$ is equivalent to it, and hence, the subsequence comparison (Definition 9) can be replaced by $<_{lex'}$ without affecting the result of the SA-IS algorithm.

Replacing $<_{lex}$ with $<_{lex'}$ and suffix type with list type in Theorem 1 and Theorem 2, eliminates the validity requirements. Furthermore, for Theorem 1, the two lists need not be suffixes of the same list. The generalized theorem of Theorem 1 and a sketch of its proof is shown below.

Theorem 3 (List Type Ordering) *Let S and T be lists that start with the same symbol. If S is an* L-type *and T is an* S-type*, then $S <_{lex'} T$.*

Proof Suppose S and T start with the symbol a. In the case that T is a prefix of S, the theorem follows from the definition of $<_{lex'}$. Otherwise, from the list type definition, the list can be written as $S = as \mathbin{@} xs \mathbin{@} b \mathbin{\#} bs$ and $T = as \mathbin{@} ys \mathbin{@} c \mathbin{\#} cs$, where as is the non-empty list consisting only of the symbol a, xs and ys are potentially empty lists that consist only of a, b is the symbol where $a > b$, c is the symbol where $a < c$, and bs and cs are potentially empty lists. The proof then follows from case analysis of the values of xs and ys.

- Case 1: $|xs| = |ys|$. The comparison between S and T reduces to comparing b and c.
- Case 2: $|xs| < |ys|$. Since the list ys consists of only a's, the comparison reduces to comparing b with a. As $b < a$, then $S <_{lex'} T$.
- Case 3: $|xs| > |ys|$. Likewise, since the list xs only consists of a's, the comparison simplifies to comparing a with c. As $a < c$, then $S <_{lex'} T$

□

Both the LMS-substrings and LMS-prefixes of a list have the same list types as the list. The two theorems and proofs are similar, and so, only the theorem for LMS-substrings is presented.

Theorem 4 (List Type of LMS-substrings) *Let S be a list and i be an index, where $i < |S|$. Then the list type of the* LMS-substring *corresponding to the i^{th} suffix is equal to the list type of the i^{th} suffix.*

Proof There are two cases, either $S[i\ldots]$ is an S-type or it is an L-type.

- Case 1: $S[i\ldots]$ is an S-type. If there is no LMS-type at any position after i, then the LMS-substring is the same as the suffix and hence is an S-type. Otherwise, the LMS-substring is of the form $xs \mathbin{@} [x, y] \mathbin{@} ys$, where $xs \mathbin{@} [x]$ is a non-decreasing list with xs being potentially empty, x and y are symbols such that $x < y$, and ys is a potentially empty list.
  - Case 1.1: xs is empty. Since $x < y$, then the LMS-substring is an S-type.
  - Case 1.2: xs is not empty. Then each element of $xs \mathbin{@} [x]$ is less than or equal to its subsequent element. Since $x < y$, the combined list must be an S-type.
- Case 2: $S[i\ldots]$ is an L-type. Then the LMS-substring is of the form $xs \mathbin{@} [x, y]$, where $xs \mathbin{@} x$ is a non-increasing list with xs being potentially empty, and x and y are symbols such that $x > y$.
  - Case 2.1: xs is empty. Then $[x, y]$ is an L-type because $x > y$.
  - Case 2.2: xs is not empty. Then each element of $xs \mathbin{@} [x]$ is greater than or equal to its subsequent element. Since $x > y$, the combined list must be an L-type.

□

From Theorems 3 and 4, it follows that LMS-substrings of L-type suffixes are less than LMS-substrings of S-type suffixes w.r.t. $<_{lex'}$, for LMS-substrings starting with the same symbol, as LMS-substrings have the same list type as their corresponding suffix (Theorem 4), and L-type lists are less than S-type lists for lists starting with the same symbol (Theorem 3) w.r.t. $<_{lex'}$. This result is exactly what Definition 9 wishes to enforce, and hence, can be replaced with $<_{lex'}$.

With the necessary underlying properties formalized, the SA-IS algorithm can now be verified.

6 Formally Verifying the SA-IS Algorithm

This section provides a more in-depth description of the verification of the SA-IS algorithm that was summarized in Section 4.2, with Section 6.1 describing the soundness proof of the abstract embedding *abs_sais*; Section 6.2 describing the equivalence proof between the *abs_sais* and the embedding of the SA-IS algorithm, *sais*; Section 6.3 describing the termination proof; and Section 6.4 describing the composition of these proofs to obtain total correctness of the SA-IS algorithm. Finally, this section is completed by Section 6.5, which is a small extension to generalize the SA-IS algorithm to non-valid sequences.

6.1 Formally Verifying the SA-IS Abstract Embedding

The abstract embedding *abs_sais*, shown in Figure 13, is very similar to the concrete embedding, shown in Figure 9. However, the suffix types are no longer pre-computed and instead, suffix type queries, which occur in *abs_extract_lms*, *abs_induce* and *abs_rename_mapping*, are computed using the definition whenever required. Moreover, additional array indexing checks are added to *abs_induce* and its subroutines, as these have intricate array updates. All of these changes make the verification easier by shrinking theorem statements and pruning cases.

The soundness of *abs_sais* is proved in two steps. First, *abs_sais* is shown to satisfy the permutation component, that is:

$$\textbf{valid}\, S \implies (abs\_sais\; S) \sim [0 \ldots < |S|]\,.$$

Then, using this fact, we show that it satisfies the sorted component,

$$\textbf{valid}\, S \implies \textbf{sorted}_{<_{lex}} (map\,(\lambda i.\, S[i \ldots])\,(abs\_sais\; S))\,.$$

The proofs of both components depend on the correctness of each subroutine of *abs_sais*, which informally are as follows:

- The *abs_extract_lms* function is correct if it extracts all LMS-type suffix locations from a list of suffix locations, where suffix locations are represented as natural numbers.
- The *abs_induce* function is correct if induced sorts the suffix locations based on the ordering of the given LMS-type suffix locations. Concretely, this means that it

```
function abs_sais :: "nat list ⇒ nat list"
  where
"abs_sais [] = []" |
"abs_sais [x] = [0]" |
"abs_sais (a # b # xs) =
  ( let
     S = a # b # xs;
     —< Scan and extract LMS-type >
     P = abs_extract_lms S [0 ...< length S];
     —< Induced sort the LMS-substrings >
     SA = abs_induce id S P ;
     —< Extract and name the sorted LMS-substrings of LMS-type suffixes >
     Q = abs_extract_lms S SA ;
     names = abs_rename_mapping S Q ;
     —< Construct the reduced sequence >
     S' = rename_string P names ;
     —< If distinct then Q is the list of sorted LMS-type suffixes, otherwise
         compute the suffix array of S' and map the suffixes back to S >
     R = (if distinct S' then Q else order_lms P (abs_sais S'))
   —< Induced sort the suffixes >
   in abs_induce id S R )"
```

Fig. 13 The abstract Isabelle/HOL embedding of the SA-IS algorithm.

produces a list of suffix locations that is a permutation of $[0 \ldots < |S|]$ and these are ordered according to the LMS-substring ordering w.r.t. $\leq_{lex'}$. Moreover, if the LMS-type suffix locations it is given are ordered according to suffix ordering w.r.t. $\leq_{lex'}$, then the list of suffix locations will also have the same ordering. Note that *abs_induce* is just the sequential application of the functions *abs_insert_lms*, which initially inserts the LMS-types at locations in the list according to their starting symbol, *abs_induce_l*, which induced sorts the L-types based on the order that the LMS-types, and *abs_induce_s*, which induced sorts the S-types based on the order of the L-types.

- The *abs_rename_mapping* function is correct if it produces the mapping from LMS-type suffix locations to the rank of its LMS-substring, where the rank is a natural number that represents how many LMS-substrings of LMS-type suffixes it is larger than.
- The *rename_string* function is correct if it produces a reduced list S', where the i^{th} LMS-type suffix of S maps to the i^{th} suffix of S' and the i^{th} LMS-type suffix of S is less than the j^{th} LMS-type suffix of S iff the i^{th} suffix of S' is less than the j^{th} suffix of S'.
- The *order_lms* function is correct if it maps the suffix array of S' to a list of LMS-type suffixes of S preserving the ordering.

Of these, the functions *abs_extract_lms*, *rename_string* and *order_lms* have very simple definitions, as these are just calls to *filter* and/or *map*, and functional correctness specification, and hence, have very simple correctness proofs, and so, these are not presented here for brevity. The functions *abs_induce* and *abs_rename_mapping* have more intricate functional correctness specifications with *abs_induce* also having a more intricate definition, and so, have much more difficult correctness proofs. These are described briefly below.

For the *abs_induce* function, we proved three separate theorems. One for the permutation component and two for different sorting components. For the permutation component, we proved the following:

Theorem 5 (Induced Sorting Permutation) *If a valid list S has more than one element, then:*

$$\begin{gathered}\textbf{distinct}\, xs \ \wedge \ set\, xs = \{i \in \mathbb{N} \mid \textbf{is\_lms}\, S\, i\} \implies \\ (abs\_induce\, S\, xs) \sim [0 \ldots < |S|]\,,\end{gathered}$$

where $\textbf{is\_lms}\, S\, i$ *is a predicate that returns true if* $S[i\ldots]$ *is an* LMS-type *suffix of* S *and false otherwise.*

Proof The proof of the above reduces to proving that each of its subroutines *abs_insert_lms*, *abs_induce_l* and *abs_induce_s* correctly insert all the LMS-type, L-type and S-type suffix locations, respectively, into the appropriate regions of their respective buckets, where a suffix's bucket is the one that matches its starting symbol. This was achieved by showing that each of the subroutines established and maintained invariants, such as the distinctness of inserted items, and that items are only inserted in their intended bucket regions. From these, we showed that no duplicates were inserted and that the suffix locations were inserted in the correct places in the list.

For the *abs_insert_lms* subroutine, we defined a version with ghost state that had a list of what had been inserted and a list of what still needed to be inserted. We then proved as an invariant that the concatenation of these lists contained of all the LMS-type locations, and that after *abs_insert_lms* completed, the list for what still needed to be inserted would become empty, thus implying that all LMS-type locations had been inserted. Finally, the two ghost state and non-ghost state versions were shown to be equivalent.

For *abs_induce_l* and *abs_induce_s* subroutines, we proved that these maintain invariants about what suffix locations were inserted. Specificallyi, if a suffix location x is in the list, then $x+1$ must also be in the list, assuming that $x+1$ is a valid suffix location, and that $x-1$ is also in the list if $x \neq 0$ and if the location where x occurs in the list has already been iterated over. These and the other invariants are then used in a proof by contradiction, showing that all L-type and S-type suffix locations must have been inserted.

Using the fact that all suffix locations inserted were distinct, were inserted in their designated locations, and all suffix locations were inserted, it then follows that the *abs_induce* function produces a permutation of $[0\ldots<|S|]$. □

To prove that *abs_induce* sorts LMS-substrings, we show the following:

Theorem 6 (Induced Sorting LMS-substrings) *If a valid list S that more than one element, then:*

$$\textbf{distinct}\, P \ \wedge \ set\, xs = \{i \in \mathbb{N} \mid \textbf{is\_lms}\, S\, i\} \implies$$

$$\mathbf{sorted}_{\leq_{lex'}}\,(map\,(\lambda i.\,abs\_lms\_slice\,S\,i)\,(abs\_induce\,S\,P))\,,$$

where abs_lms_slice $S\,i$ returns the LMS-substring *at position i in S.*

Proof The proof of the above reduces to proving that each subroutine of *abs_induce* maintains an invariant about inserting suffix locations in sorted order, as described in Sections 3.2.2 to 3.2.4, and these were proved using the invariants utilized in Theorem 5 and Theorems 3 and 4 to show that the ordering is preserved between L-type and S-type LMS-substrings. □

A similar proof is also used to show that *abs_induce* sorts suffixes, that is:

Theorem 7 (Induced Sorting Suffixes) *If a valid list S has more than one element, then:*

$$\mathbf{distinct}\,R\ \wedge\ set\,R = \{i \in \mathbb{N} \mid \mathbf{is\_lms}\,S\,i\}\ \wedge\ \mathbf{sorted}_{\leq_{lex'}}\,(map\,(\lambda i.\,S[i\ldots])\,R)\ \Longrightarrow$$
$$\mathbf{sorted}_{\leq_{lex'}}\,(map\,(\lambda i.\,S[i\ldots])\,(abs\_induce\,S\,R))\,.$$

For the *abs_rename_mapping* function, we show that it constructs the mapping from LMS-substrings of LMS-type suffixes to the new alphabet, preserving the LMS-substring ordering w.r.t. $\leq_{lex'}$. In our formalization, each LMS-substring is mapped to a natural number, which we refer to as its *rank*, that represents how many LMS-substrings of LMS-type suffixes are smaller than it w.r.t. $<_{lex'}$. Mathematically, the *rank* function is defined as follows:

Definition 13 (Rank) Given a comparison function $<$ and a set X, the rank of value x is defined as follows:

$$rank_{<}\,X\,x = \|\{y \mid y \in X \wedge y < x\}\|\,.$$

If X is finite and $<$ is transitive and irreflexive, then the function *rank* will be monotonic.

The *abs_rename_mapping* function constructs the mapping by iterating over a sorted list of LMS-substrings with an accumulator that is initialized to 0. During each iteration, the current element is mapped to the current accumulator. In addition to this, the current element is compared with the next element. If they are equal, the accumulator remains unchanged, otherwise, the accumulator is incremented by one. The mapping is then stored in a list.

The correctness theorem for *abs_rename_mapping* is then as follows:

Theorem 8 (Correctness of the New Alphabet Mapping)

$$\mathbf{distinct}\,Q\ \wedge\ \mathbf{sorted}_{\leq_{lex'}}\,(map\,(\lambda j.\,abs\_lms\_slice\,S\,j)\,Q)\ \wedge\ i \in set\,Q\ \wedge\ i < |S| \longrightarrow$$
$$(abs\_rename\_mapping\,S\,Q)[i] = rank_{<_{lex'}}\,\{abs\_lms\_slice\,S\,j \mid j \in set\,Q\})\,(abs\_lms\_slice\,S\,i)\,,$$

where abs_lms_slice computes the LMS-substrings.

Proof The above is proved by generalizing the *abs_rename_mapping* function so that it initializes the accumulator to some arbitrary value k, and then we show, by induction on the list of natural numbers Q, that each LMS-substring of an LMS-type suffix is mapped to its rank plus k. The proof of the two base cases, when Q is empty and when it is a singleton list, follows from the generalized definition of *abs_rename_mapping* and Definition 13. In the inductive case, where $Q = a \# b \# Q'$, we have two inductive hypotheses. One for when the LMS-substrings at a and b are equal and one for when this is not the case. Hence, we prove the inductive case using these cases.

In the first case, where the LMS-substring at a is equal to the LMS-substring at b, their ranks will be both equal to 0, due to the sorted assumption. Moreover, the accumulator in *abs_rename_mapping* will not be incremented after comparing the two LMS-substrings, leading to both a and b mapping to k according to the definition of *abs_rename_mapping*. Then the rest of the proof for this case follows from the first inductive hypothesis.

In the second case, since LMS-substrings at a and b are not equal, a will be mapped to k and b will be mapped to $k+1$, according to the definition of *abs_rename_mapping*. Moreover, we know that $\mathit{abs\_lms\_slice}\,S\,a \ <_{lex'}\ \mathit{abs\_lms\_slice}\,S\,b$, as this follows from the sorted assumption, and that LMS-substrings at a and b are not equal. Using the fact that Definition 13 is monotonic, we then have that the ranks of $\mathit{abs\_lms\_slice}\,S\,a$ and $\mathit{abs\_lms\_slice}\,S\,b$ will be equal 0 and 1 respectively. The rest of the proof then follows from the second inductive hypothesis. □

Besides the correctness theorems of the subroutines, the correctness of *abs_sais* also depends on showing that the mapping from the i^{th} LMS-type suffix of S to the i^{th} suffix of the reduced list S' is monotonic. This is achieved by formalizing the observations and mapping as described in Section 3.2.1. The most important of these was to prove that the $<_{lex'}$ comparison LMS-substrings implies the $<_{lex'}$ comparison of suffixes, as the monotonicity component of the mapping depends on this. Formally, this is defined as:

Theorem 9 (LMS-substring Ordering Implies Suffix Ordering) *For i and j less than $|S|$,*

$$\mathit{abs\_lms\_slice}\,S\,i <_{lex'} \mathit{abs\_lms\_slice}\,S\,j \implies S[i\ldots] <_{lex'} S[j\ldots]\,.$$

Proof There are two cases. The first case is the following:

$$\exists b\ c\ as\ bs\ cs.\ \mathit{abs\_lms\_slice}\,S\,i = as\,@\,b \# bs \wedge \mathit{abs\_lms\_slice}\,S\,j = as\,@\,c \# cs \wedge b < c\,,$$

where there exists a mismatch at some position of between the two LMS-substrings. The second case is the following:

$$\exists a\ as.\ \mathit{abs\_lms\_slice}\,S\,i = \mathit{abs\_lms\_slice}\,S\,j\,@\,a \# as\,,$$

where the LMS-substring at position j is a prefix of the LMS-substring at position i.

Proving the first case is trivial as the LMS-substring is always a prefix of the suffix, and so, from the definition of $<_{lex'}$ we have that $S[i\ldots] <_{lex'} S[j\ldots]$.

In the second case, using the definition of an LMS-substring, we have that there exists some ys and k such that $\mathit{abs\_lms\_slice}\,S\,j = bs\,@\,[S[k], S[k+1]]$, where $S[k] > S[k+1]$ and $S[k+1\ldots]$ is an LMS-type. With this, we then have that for some a and as

$$\mathit{abs\_lms\_slice}\,S\,i = bs\,@\,[S[k], S[k+1], a]\,@\,as\,.$$

Moreover, there exists some xs and ys such that $S[i\ldots] = \mathit{abs\_lms\_slice}\, S\, i \mathbin{@} xs$ and $S[j\ldots] = \mathit{abs\_lms\_slice}\, S\, j \mathbin{@} S[k+1\ldots]$. And so, the problem reduces to showing that $S_i <_{lex'} S_j$, where $S_i = [S[k+1], a] \mathbin{@} as \mathbin{@} xs$ and $S_j = S[k+1\ldots]$.

We know that $S[k+1]$ must be greater than or equal to a, otherwise this would lead to a contradiction. That is, if $S[k+1] < a$, then $[S[k+1], a] \mathbin{@} as \mathbin{@} xs$ would be an S-type, and since $S[k] > S[k+1]$ then it would in fact be an LMS-type, thus meaning that

$$\mathit{abs\_lms\_slice}\, S\, i \neq \mathit{abs\_lms\_slice}\, S\, j \mathbin{@} a \mathbin{\#} as\,,$$

which would be a contradiction.

Since $S[k+1] \geq a$, this makes S_i an L-type, while S_j is an S-type as $S[k+1\ldots]$ is an LMS-type. and from the definition of LMS-substring. Using Theorem 3, we then have that $S_i <_{lex'} S_j$. □

With all the correctness theorems of the subroutines and that the LMS-type suffixes of S map to suffixes of the list S', preserving the ordering w.r.t. $<_{lex'}$, we can then prove that *abs_sais* produces a permutation by induction on the list S, that is:

Theorem 10 (*abs_sais* Produces a Permutation)

$$\textbf{valid}\, S \implies (\mathit{abs\_sais}\, S) \sim [0 \ldots < |S|]\,.$$

Proof Its base cases, which are $S = []$ and $S = [x]$ for some arbitrary x, are trivial to prove as both of these are degenerate cases for permutations.

In the inductive case where $S = a \mathbin{\#} b \mathbin{\#} xs$, we use Theorems 5, 6 and 8, the correctness theorems of *abs_extract_lms* and *rename_string* and the correctness of the mapping itself to show that the reduced list S' is constructed correctly, and hence, satisfies the valid list requirement. Then we do a case analysis on the distinctness of the elements of S'. If the elements of S' are distinct, we can then use Theorem 5 to show that *abs_sais* produces a permutation of $[0 \ldots < |S|]$. If the elements of S' are not distinct, we use the inductive hypothesis

$$\textbf{valid}\, S' \implies (\mathit{abs\_sais}\, S') \sim [0 \ldots < |S'|]$$

and the correctness theorem of *order_lms* to show that mapping the suffixes of S' back to suffixes of S produces a list of all LMS-type suffixes of S without duplicates. Then Theorem 5 can be used to complete the proof. □

A similar proof approach can be used to show that *abs_sais* sorts suffixes, that is:

Theorem 11 (*abs_sais* Sorts Suffixes)

$$\textbf{valid}\, S \implies \textbf{sorted}_{<_{lex}} (\mathit{map}\, (\lambda i.\, S[i\ldots])\, (\mathit{abs\_sais}\, S)).$$

Proof We first prove by induction on the list S that

$$\textbf{valid}\, S \implies \textbf{sorted}_{<_{lex'}} (\mathit{map}\, (\lambda i.\, S[i\ldots])\, (\mathit{abs\_sais}\, S))\,.$$

Its base cases, which are $S = []$ and $S = [x]$ for some arbitrary x, are also trivial to prove as both of these are degenerate cases for sorting.

For the inductive case, when $S = a \mathbin{\#} b \mathbin{\#} xs$, we follow a similar approach to Theorem 10, of showing that the construction of the reduced list S' and handling the two cases when S'

is distinct and when it is not. However, we now use Theorem 7 and the inductive hypothesis, which is now

$$\textbf{valid}\, S' \implies \textbf{sorted}_{<_{lex'}}\, (map\, (\lambda i.\, S[i \ldots])\, (abs\_sais\, S'))\,,$$

to show that the suffixes are sorted w.r.t. $\leq_{lex'}$. Note that we also use Theorem 10 to show that the suffix array of S' is a permutation of $[0 \ldots < |S'|]$ as this is an assumption of the correctness theorem of *order_lms*, which is used to map suffixes of S' back to LMS-type suffixes of S.

Having proved that the suffixes of S are sorted w.r.t. $<_{lex'}$ if S is valid, we then use the fact that $<_{lex'}$ and $<_{lex}$ are equivalent for valid lists to show the suffixes are sorted w.r.t. $<_{lex}$. □

By combining Theorems 10 and 11, we obtain partial correctness of *abs_sais*.

Theorem 12 (Partial Correctness of *abs_sais*)

$$\textbf{valid}\, S \implies \textbf{sa}\, (abs\_sais\, S)\,.$$

6.2 Proving Equivalence of SA-IS and Abstract Embedding

Recall that the *abs_sais* function is an abstract embedding of the SA-IS algorithm, where the suffix types are computed each time rather than using a precomputed value like in the algorithm. Moreover, additional array indexing checks are added in the induced sorting subroutines. We now show that concrete embedding (*sais*) of the SA-IS algorithm, which precomputes the suffix types and does not have the additional array indexing checks, is equivalent to the abstract embedding in two phases.

In the first phase, we show that using the precomputed suffix types, which are computed by *get_types* at the start of each call to *sais*, in the functions *extract_lms*, *rename_mapping*, *induce* and its sub-functions, results in *sais* producing the correct suffix array, that is, *sais* is equivalent to *abs_sais*. These proofs are straightforward and follow from the correctness theorem of *get_types*. The *get_types* function is simply the implementation of Definition 5, and its proof is the formalization of the intuitive arguments presented in Section 3.1.

In the second phase, we show that the additional array indexing checks in the *induce* function and its sub-functions can be removed. This is a little more challenging, as this information is still needed. Fortunately, the required information is derivable from the assumptions of the correctness theorems of the *induce* function, that is, it is given a valid list consisting of more than one element and a list containing all LMS-type suffix positions without duplicates. Relying on these assumptions means the equivalence is restricted to only valid lists.

By combining the two phases, we then obtain the equivalence as follows:

Theorem 13 (Equivalence of *abs_sais* and *sais*)

$$\textbf{valid}\, S \implies sais\, S = abs\_sais\, S\,.$$

6.3 Termination

Both *abs_sais* and *sais* require manual termination proofs, as Isabelle's termination checker is not powerful enough to automatically determine if these terminate. These proofs, however, are also not too difficult as termination for both of these follows from the fact that the reduced list, that is, the new list whose suffixes correspond to the LMS-type suffixes of the original list, is always guaranteed to be strictly smaller than the original list — in fact, it is less than or equal to half the size of the original list. This fact is a direct consequence of the LMS-type suffix definition, which states that an S-type suffix is an LMS-type suffix only if it occurs directly after an L-type suffix, and this naturally means that for an n length list, there can only be at most $\lfloor n/2 \rfloor$ many LMS-type suffixes. From this, it follows that *sais* (and *abs_sais*) terminate.

Theorem 14 (Termination of *sais*)

$$\textbf{terminates}\, \textit{sais}\,.$$

Note that in Isabelle/HOL, there is no **terminates** predicate, and this is simply for presentation purposes. Instead, Isabelle/HOL automatically generates a domain predicate, which returns true for inputs that a function terminates for, when a function is defined, and termination is proved by showing that the domain predicate is always true for all inputs.

6.4 Verifying the Total Correctness of the SA-IS Algorithm

Using Theorem 13 to replace *abs_sais* with *sais* in Theorem 12, and combining this with Theorem 14, we obtain the total correctness of *sais*.

Theorem 15 (Total Correctness of *sais* for Valid Lists of Naturals)

$$(\textbf{terminates}\, \textit{sais}\,)\ \wedge\ (\forall S.\, \textbf{valid}\, S \longrightarrow \textbf{sa} S\, (\textit{sais}\ S))\,.$$

6.5 Verifying the Generalized SA-IS Algorithm

Recall from Section 2.2 that many suffix array construction algorithms assume that the input list is valid. This is because most are based on the C string convention, where strings are terminated by the null character, which is the smallest ASCII character. Note that the valid list requirement is essential, as algorithms, like SA-IS, can fail if this requirement is not met. For example, giving the SA-IS algorithm the list `aaaa` will fail to produce a result the list has no LMS-type suffixes.

Algorithms, like SA-IS, can still construct suffix arrays for non-valid lists. This can be achieved by using a lifting function that maps the symbols in a non-valid list S to symbols in an augmented alphabet, excluding the sentinel symbol, in an order preserving manner, and then appends the sentinel to the new list to become a valid list S'. The suffix array of S' can now be computed using the sophisticated suffix array construction algorithms. The suffix array of the original list S is then simply the array formed by taking the elements of the suffix array of S' from position 1 up to and including position $|S|$, that is, the last element. Formally, this generalized suffix array construction, denoted by *lift*, is as follows:

Definition 14 (Generalizing Suffix Array Construction)

$$\mathit{lift}\, f\, S \;:=\; (f\,((\mathit{map}\,(\lambda x \in \Sigma.\, \alpha\, x + 1)\; S) \mathbin{@} [0]))[1 \ldots < |S|]\,,$$

where f is an arbitrary suffix array construction function and α is an order preserving mapping from the alphabet Σ to the natural numbers.

Note that there always exists such a mapping α. For example, the mapping $(\lambda x \in \Sigma.\, \|\{i \mid S[i] < x\}\|)$ could be a potential instantiation for α. From the definition of *lift* (Definition 14), any suffix array construction function that terminates and satisfies the correctness specification can then be generalized to construct suffix arrays for any list, regardless of whether they satisfy the valid list condition or not.

Theorem 16 (Total Correctness Generalizes)

$$\begin{gathered}\textbf{terminates}\, f \longrightarrow \textbf{terminates}\,(\mathit{lift}\;\, f) \quad \wedge \\ (\forall S.\, \textbf{valid}\, S \longrightarrow \textbf{sa} S\,(f\; S)) \longrightarrow (\forall S.\, \textbf{sa} S\,(\mathit{lift}\, f\; S))\,.\end{gathered}$$

By instantiating f with *sais* in Theorem 16, and using Theorem 15, the correctness theorem for a generalized SA-IS algorithm can then be obtained.

Corollary 17 (Total Correctness of Generalized SA-IS)

$$\textbf{terminates}\,(\mathit{lift}\; \mathit{sais}\;\;) \wedge (\forall S.\, \textbf{sa} S\,(\mathit{lift}\; \mathit{sais}\;\; S))\,.$$

7 Discussion

We now summarize the proof effort and share some observations and experiences that arose during this project.

7.1 Proof Effort

While difficult to characterize in exact terms, we believe it is still valuable to consider the amount of effort that was required to complete such a project. Accurately estimating the verification effort that was required for a formalization involves factors such as the tools that were used and their *trusted computing base* (TCB), which is all the hardware and software that must be trusted to function as intended so that the proofs are sound, the time needed to develop the proofs, the fact that multiple directions may have been explored as part of the overall project, and the level of experience of the proof engineer, who in this case had two and half years of experience in interactive proof development, all of which used Isabelle/HOL, prior to undertaking this work. All of these factors in turn may affect the length and complexity of the proofs [28, 29]. Nevertheless, an estimate of proof complexity can be determined by considering the

Table 1 The number of lines of common definitions and theories that the verification of *simple* and *sais* both rely on; including properties about suffixes and classical lexicographical order.

Specification	Suffixes	Classical Lexicographical Order
29	57	45

Table 2 Number of lines of algorithm definition, and in the proofs of correctness and additional theorems.

Algorithm	Alg. Model	Background	Correctness	Equivalence	Miscellaneous
simple	18	0	186	0	0
sais	231	4693	13736	841	4006

number of lines needed for each line of code in the algorithm, and the time taken to complete it [28].

The simple algorithm, outlined in Figure 1, is approximately 20 lines when embedded in Isabelle/HOL, and its verification is approximately 180 lines of proof; giving a lines of code to lines of proof ratio of 1:9. Less than a single person-day was needed for that initial activity, shown in the first row of Table 2.

In contrast, the Isabelle/HOL embedding *sais* of the SA-IS algorithm is approximately 230 lines, including the various functions that are called. The formal verification of the abstract Isabelle/HOL embedding, *abs_sais*, is composed of over 18,000 lines of proof, plus approximately 4000 more for various underlying properties (including, for example, properties about sorting and bijections); also shown in Table 2. The equivalence to the concrete HOL implementation, which resembles Figure 3 more closely, adds approximately 840 lines of definitions and proof.

In total, the formalization required for this more complex algorithm consists of over 23,000 lines of Isabelle/Isar text, and results in a 1:100 ratio of lines of code to lines of proof, more than a ten-fold increase in ratio compared to the simple algorithm. The verification of the SA-IS algorithm took about two and a half person-years (a 500:1 ratio compared to the simple algorithm's one day) with each proof line taking significantly longer to develop than the proof lines for the simple approach.

One key factor that made the simple algorithm so much easier to verify is that it uses the native sorting function from the Isabelle/HOL library in its encoding. This sorting function has already been verified, meaning that the only remaining challenge in connection to the simple algorithm was to show that the mapping from suffix to suffix identifier (and vice versa) is correct, which was a relatively straightforward task.

On the other hand, the verification of the SA-IS algorithm required formalizing a library containing various properties of sequences, some of which, specific to the SA-IS algorithm. In addition, the algorithm is conceptually quite complex. It utilizes an additional sequence to store the current bucket states and both this additional sequence and the output sequence are modified through several levels of indirection.

While the verification of the SA-IS algorithm was time-consuming and complex, it was at least parallelizable, with the verification of each subroutine of the SA-IS algorithm independent of the other subroutines.

7.2 Proof Comparison

Our work formalizes the SA-IS algorithm that was developed by Nong et al. [5], and the formalized theorems and proofs thus resemble the sketch provided in their paper. However, our formalization also differs in some areas, and in the level of detail provided. We present three major instances: the first instance, described in Section 7.2.1, is about how the suffix type is defined; the second instance, described in Section 7.2.2, is about how the subsequence comparison is defined; and the third instance, described in Section 7.2.3, is about how the induced sorting subroutine was shown to be correct.

7.2.1 Suffix Types

As described in Section 5, we define the suffix type differently, using $<_{lex'}$ instead of $<_{lex}$ for the suffix comparison. This leads to the removal of the special case for the last suffix and the requirement that the sequence must be valid. It also makes the suffix type definition total. Having special cases is not ideal, since these cases add to the analysis burden.

To avoid the issue of invalid sequences we could have defined a new type, which incorporates the validity requirement into the type definition. While this approach would also lead to the desired result, it would have been somewhat heavy-handed, requiring additional set-up and minimal additional benefit. In addition, it would not have solved the special case issue and it does not resolve the case when the index is out of bounds.

An alternative that is somewhat similar to the one that we employed, is to provide sensible results for invalid sequences, covering situations in which the index is out of bounds. However, there is no consensus as to what those sensible values should be, especially in the case of the last suffix.

Since the last suffix of a valid sequence is always an S-type, it would suggest that the last suffix of an invalid sequence should also be an S-type. However, this would mean that for the sequence $S = \mathsf{aa}$, $S[0\ldots]$ would be an L-type, as $\mathsf{a} <_{lex} \mathsf{aa}$. While this is not a major issue as this is an invalid sequence and using the lifting function, Definition 14, would transform any sequence into a valid one, it still requires a special case for the last suffix. Moreover, the suffix type properties would not be generalizable to all possible sequences as these would still only apply to valid sequences.

We could have instead made the last suffix of an invalid sequence an L-type. This would be consistent with how $<_{lex}$ is defined, as for any sequence S of length $n+1$, we know that $S[n+1\ldots] <_{lex} S[n\ldots]$. But doing so would erode generality, as the last suffix of a valid sequence would be of a different type to the last suffix of an invalid sequence. That would then prevent having a generalized version of Theorem 1.

7.2.2 Subsequence Comparison

The subsequence comparison, described in Definition 9, is a partial function that takes three arguments: a valid sequence, and two suffix positions. As stated earlier in Section 5, this makes verification more difficult. Our formalization does not use this comparison function and instead uses the revised $<_{lex'}$ function. We further describe the reasoning behind this choice here.

Making Definition 9 total and a binary relation would be possible, assuming that LMS-prefixes and LMS-substrings have the same suffix types as their suffixes', that is, proving Theorem 4. The valid sequence and two suffix positions can be replaced by the LMS-prefixes or LMS-substrings at the two suffix positions, and simply ignoring the validity and the same sequence requirements would produce sensible results.

Proving that Theorem 4 and the corresponding theorem for LMS-prefixes, would mean that we could just use $<_{lex'}$ in conjunction with Theorem 3 for the subsequence comparison. Hence there is no major incentive to do the extra work of defining a total and binary version of Definition 9. Moreover, we find $<_{lex'}$ more elegant as it provides a better explanation of why the induced sorting subroutine can be used to sort LMS-substrings.

7.2.3 Induced Sorting

The correctness proof of the induced sorting subroutine in our formalization differs from the natural language justification provided by Nong et al. [5]. Our formalization has significantly more detail and precision. This is a direct consequence of our formalization being machine-readable and machine-checkable, aspects that require extreme rigor and formal presentation. The natural language justification provided by Nong et al. [5], on the other hand, only needs to provide enough detail for the reader to be convinced of its plausibility, thus leading to a significant difference in detail and level of precision.

To prove that the induced sorting subroutine is correct, the array after induced sorting must be shown to have all the suffixes or LMS-substrings in sorted order and contain all the suffixes or LMS-substrings without duplicates. The informal correctness argument provides sufficient detail to justify that the induced sorting subroutine sorts the array. However, the informal argument does not provide enough detail, if any at all, to show that all suffixes are inserted in their correct positions without any duplicates being introduced. This part perhaps seems trivial, but it is the largest and most complex part of the entire Isabelle/HOL verification effort. It almost accounts for half of the 23,000 lines of proof that were developed as part of this formalization.

The challenge arises because the sequence is updated in-place. Another difficulty is that what is being inserted next depends on what is currently in the sequence. All of this required much proof exploration to determine suitable invariants, and then further effort to demonstrate their validity.

7.3 Verification Approach

As described in Section 4, our verification approach was to verify the correctness of an abstract shallow embedding of the SA-IS algorithm and then show that this is equivalent to a shallow embedding of the algorithm, that resembles its actual implementation, albeit still using lists and natural numbers rather than arrays and machine words. Considering that the SA-IS algorithm is particularly intricate, we specifically chose an approach tailored to verifying the correctness rather than structural properties like efficiency. However, this approach also has its limitations as it cannot provide any formal guarantee about efficiency and requires that imperative programs be transformed into functional programs, which in this case was straightforward and in many others, but can be nontrivial in other cases.

Using Isabelle's Refinement Framework (IRF) to Imperative/HOL [30] would provide a structured approach to verifying imperative programs by refining high-level specifications into efficient executable code. This is because it supports stepwise refinement through a series of correctness-preserving transformations, employing monadic representations and separation logic to model imperative behavior. If we used the refinement framework in our formalization, there would be major benefits for the equivalence proof step, but few benefits for the correctness of the abstract embedding as this is the high-level specification that would be refined to an efficient implementation. Considering that the equivalence proof is only 840 lines, which is less than 4% of the formalization of the SA-IS algorithm, the overall benefit would be minimal. However, by using our formalization as the high-level specification of the SA-IS algorithm, the refinement framework could be used to systematically develop an efficient and correct-by-construction implementation of the SA-IS algorithm. Moreover, since the algorithm is refined to an Imperative/HOL implementation, the framework of Zhan and Haslbeck [31] could be used to handle proofs about asymptotic complexity.

7.4 Reusability

Our formalization of the SA-IS algorithm is reusable (at least in parts), as it captures structural properties common to many suffix array construction algorithms. These are the classification of suffixes into different types, and induced sorting. Moreover, our formalization provides a foundation for verifying applications that use suffix arrays.

Recall from Section 3.1 that the SA-IS algorithm classifies suffixes, LMS-prefixes and LMS-substrings into two types: S-types and L-types. The S-type suffixes can be further classified as LMS-type suffixes if they occur directly after an L-type suffix. The KA algorithm [4], which is the predecessor of the SA-IS algorithm, uses a classification that is almost identical, except that does not use the LMS-type suffixes and also classifies the last suffix, which is just the sequence containing the last symbol $, as both an S-type and L-type suffix. These differences would require only minimal modifications in order to be accommodated, and large portions of our formalization of suffix types could be reused unchanged. Several other algorithms [32–34], which are based on the SA-IS algorithm, use the exact same classification of suffixes, LMS-prefixes and LMS-substrings. This means that our formalization of the suffix types is is also applicable to the verification of these algorithms. Note that other algorithms also classify suffixes into different types [18, 19], however, while their classification of suffixes shares some features, there are also fundamental differences that would be likely to require major modifications of our formalizations.

Induced sorting, detailed in Sections 3.2.2 to 3.2.4, is an approach inherent to the many suffix array construction algorithms in the induced-copying category [17]. The KA algorithm [4] uses the same procedure, shown earlier in Figure 8, to induce-sort S-type suffixes, with a slightly altered procedure for the L-type suffixes, in which all the S-type suffixes are sorted rather than just the LMS-type suffixes. Slight modifications to our formalization of induced sorting L-type suffixes would be required; but since the non-S-type suffixes do not affect the order of L-type suffixes, the only substantial change would be an additional theorem for non-interference. The algorithms based on the SA-IS algorithm [32–34] have slight modifications to their induced sorting procedures, however, these are largely optimizations for computing suffix types on the fly rather than pre-computing and storing them, and how the bucket pointers are stored. Our formalization of induced sorting could be used as is, with additional theorems for showing refinement or equivalence with the induced sorting procedures of the algorithm based on the SA-IS algorithm. Other algorithms, such as the IT [18] and DivSufSort [19] techniques, use different suffix types; proofs of their correctness might also reuse some of our formalization, with the sorting arguments modified to account for the change in suffix types.

Our formalization can also be applied to other applications that use suffix arrays. These include data compression [7, 35] and string algorithms [2, 36]. In particular, recall from Section 1 that the BWT [7] can be computed from a suffix array in linear-time, and hence, when combined with the SA-IS algorithm, results in a linear-time construction algorithm for the BWT. In recent work [37], we formalized the BWT in Isabelle/HOL, available in the Archive of Formal Proofs [38]. By combining our formalizations of the SA-IS algorithm and the BWT, we then have a formally verified and linear-time construction of the BWT.

8 Related Work

Other than the related work we mention earlier, we further elaborate on two areas. The first contrasts various suffix array construction algorithms and the second gives an overview of verification efforts in this context.

8.1 Suffix Arrays

The suffix array was conceived by Manber and Myers [1] in the early 1990s as a space efficient alternative to suffix trees. Since then, various algorithms [17] have been developed to efficiently construct them. These generally lie within three categories: prefix-doubling, recursive and induced-copying.

Prefix-doubling algorithms [39] utilize a dynamic programming approach based on an approach by Karp et al. [40] for rapid identification of patterns. This is based on the observation that suffixes can be sorted by incrementally sorting their prefixes, doubling the prefix length at each iteration. Generally, the worst-case asymptotic time complexity of this approach is $\mathcal{O}(n \log n)$. The original suffix construction algorithm by Manber and Myers [1] used this prefix-doubling approach.

Recursive algorithms [3, 4, 6], such as the SA-IS algorithm, are founded on the idea that suffixes can be classified into types, which was first introduced by Farach [41].

The suffix type contains ordering information that can be used to sort the suffixes by induced sorting. This process requires the relative order of a subset of the suffixes to be known. These algorithms recursively sort this subset of the suffixes by constructing a reduced sequence from the suffixes and then building the suffix array of the reduced sequence. The worst-case asymptotic time complexity of these algorithms is typically linear;, however, their memory footprint tends to be larger.

Induced-copying algorithms [18, 19, 42] are similar to the recursive algorithms in that they also categorize suffixes into types and use induced sorting to sort the suffixes. However, rather than recursively sorting a subset of suffixes, these induced-copying algorithms sort the suffixes iteratively. While such algorithms usually have a worst-case asymptotic time complexity of $\mathcal{O}(n^2 \log n)$, tend to be faster in practice and tend to have a lighter memory footprint. For instance, the DivSufSort algorithm [19] is regarded as the most efficient algorithm in practice.

8.2 Verified Data Structures and Algorithms

Formally verifying data structures and algorithms is an important area of research. Not only does it provide strong guarantees that said data structures and algorithms behave as intended, but it also leads to a deeper understanding of their mechanics. There are many examples of formally verified algorithms and data structures [43, 44], here we highlight those that are most related to suffix arrays. That is suffix arrays themselves, sorting and string search.

With many suffix array construction algorithms having been described and their widespread use in pattern matching applications, one would expect that there would be more formally verified instances. However, the only documented instances are those that verify simple algorithms that are similar to the *simple* algorithm (Figure 1). These verified instances [45–47] were all part of solutions for an advanced task of the VerifyThis 2012 competition [48]. Note that all of the verified instances assume that the sorting function is correct, that is, the sorting algorithm used had already been verified.

Other than the simple algorithm, various other suffix array construction algorithms similarly rely on existing sorting algorithms, many of which have been verified [49–52]. For example, the DivSufSort suffix array construction algorithm uses introspective sort, which itself is an amalgamation of multi-key quicksort, heapsort and insertion sort, and this has been verified by Lammich [53].

Suffix arrays are one way to solve the substring matching problem. However, it involves building an index of the whole text. Alternative approaches include the Knuth-Morris-Pratt (KMP) algorithm [54] and the Boyer-Moore (BM) algorithm [55] which both apply some preprocessing to the pattern being searched for. The KMP algorithm [56–58] and the BM algorithm [59, 60] have been formally verified. A verified LLVM implementation of the KMP algorithm was synthesized by Lammich [61].

9 Conclusion and Future Work

We have demonstrated the first formal verification of the SA-IS algorithm; a linear time suffix array construction algorithm. Our Isabelle/HOL embedding of the algorithm contains enough detail that it can be extracted to executable Haskell code. During this process, we developed an axiomatic characterization of suffix arrays that we validated by verifying the correctness of a simple suffix array construction algorithm, which was also used as a point of comparison. Moreover, we also formalized suffix properties that are critical to the SA-IS algorithm and other suffix array construction algorithms, and we also generalized many of these properties, leading to a deeper understanding of these properties and the SA-IS algorithm.

Our formalization guarantees the correctness of an Isabelle/HOL embedding of the SA-IS algorithm. This leads to several potential avenues for future work, such as proving that the asymptotic time complexity of the SA-IS algorithm is indeed linear, and developing a demonstrably correct and efficient implementation in a procedural language. To show that the SA-IS algorithm has a linear asymptotic complexity, one could refine our Isabelle/HOL implementation to an Imperative/HOL implementation and use the framework of Zhan and Haslbeck [31] to prove asymptotic complexity. For an efficient and correct implementation, one could develop a handcrafted C implementation and show that it refines a verified HOL embedding [27, 62], using tools such as AutoCorres [63] and Cogent [64, 65].

Acknowledgements

This work is partially supported by an Australian Government Research Training Program (RTP) stipend.

Competing Interests

The authors have no competing interests to declare that are relevant to the content of this article.

Ethics Approval

No ethics approvals were required for this article.

Data, Materials and Code Availability

The formalization is available online in the Archive of Formal Proofs [26].

References

[1] Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing **22**(5), 935–948 (1993) https://doi.org/10.1137/0222058

[2] Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms **2**(1), 53–86 (2004) https://doi.org/10.1016/S1570-8667(03)00065-0

[3] Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proc. Automata, Languages and Programming. Lecture Notes in Computer Science,

vol. 2719, pp. 943–955. Springer, Eindhoven, The Netherlands (2003). https://doi.org/10.1007/3-540-45061-0_73

[4] Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. Journal of Discrete Algorithms **3**(2-4), 143–156 (2005) https://doi.org/10.1016/j.jda.2004.08.002

[5] Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proc. Data Compression Conference, pp. 193–202. IEEE Computer Society, Snowbird, UT, USA (2009). https://doi.org/10.1109/DCC.2009.42

[6] Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Transactions on Computers **60**(10), 1471–1484 (2011) https://doi.org/10.1109/TC.2010.188

[7] Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report (1994)

[8] Seward, J.: bzip2 Homepage. https://sourceware.org/bzip2/index.html. Accessed: 2023-09-12 (1996)

[9] Ferragina, P., Manzini, G.: Indexing compressed text. Journal of ACM **52**(4), 552–581 (2005) https://doi.org/10.1145/1082036.1082039

[10] Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nature Methods **9**(4), 357–359 (2012) https://doi.org/10.1038/nmeth.1923

[11] Langmead, B., Trapnell, C., Pop, M., Salzberg, S.L.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology **10**(3), 25 (2009) https://doi.org/10.1186/gb-2009-10-3-r25

[12] Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics **25**(14), 1754–1760 (2009) https://doi.org/10.1093/bioinformatics/btp324

[13] Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S., Kristiansen, K., Wang, J.: SOAP2: An improved ultrafast tool for short read alignment. Bioinformatics **25**(15), 1966–1967 (2009) https://doi.org/10.1093/bioinformatics/btp336

[14] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science, vol. 2283. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9

[15] Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge (2007)

[16] Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer, Berlin,

Heidelberg (2014). https://doi.org/10.1007/978-3-319-10542-0

[17] Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Survey **39**(2), 4 (2007) https://doi.org/10.1145/1242471.1242472

[18] Itoh, H., Tanaka, H.: An efficient method for in memory construction of suffix arrays. In: Proc. String Processing and Information Retrieval, pp. 81–88. IEEE Computer Society, Cancun, Mexico (1999). https://doi.org/10.1109/SPIRE.1999.796581

[19] Fischer, J., Kurpicz, F.: Dismantling DivSufSort. CoRR **abs/1710.01896** (2017) https://doi.org/arXiv:1710.01896

[20] Gordon, M., Milner, R., Wadsworth, C.P.: Edinburgh LCF. Lecture Notes in Computer Science, vol. 78. Springer, Berlin, Heidelberg (1979). https://doi.org/10.1007/3-540-09724-4

[21] Milner, R., Tofte, M., Harper, R.: Definition of Standard ML. MIT Press, Cambridge, Massachusetts, USA (1990)

[22] Minsky, Y., Madhavapeddy, A., Hickey, J.: Real World OCaml - Functional Programming for the Masses. O'Reilly, Sebastopol, California, USA (2013)

[23] Jones, S.L.P.: Haskell 98: Introduction. Journal of Functional Programming **13**(1), 0–6 (2003) https://doi.org/10.1017/S0956796803000315

[24] Odersky, M., Altherr, P., Cremet, B., Dragos, I., Dubochet, G., Emir, B., McDirmid, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Spoon, L., Zenger, M.: An overview of the Scala programming language. Technical report, EPFL, Lausanne, Switzerland (2004). https://infoscience.epfl.ch/handle/20.500.14299/214698

[25] Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Proc. Functional and Logic Programming. Lecture Notes in Computer Science, vol. 6009, pp. 103–117. Springer, Sendai, Japan (2010). https://doi.org/10.1007/978-3-642-12251-4_9

[26] Cheung, L., Rizkallah, C.: Formally verified suffix array construction. Archive of Formal Proofs (2024). https://isa-afp.org/entries/SuffixArray.html, Formal proof development

[27] Cheung, L., O'Connor, L., Rizkallah, C.: Overcoming restraint: Composing verification of foreign functions with Cogent. In: Proc. Certified Programs and Proofs, pp. 13–26. ACM, Philadelphia, PA, USA (2022). https://doi.org/10.1145/3497775.3503686

[28] Matichuk, D., Murray, T., Andronick, J., Jeffery, R., Klein, G., Staples, M.: Empirical study towards a leading indicator for cost of formal software verification. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. ICSE, vol. 1, pp. 722–732 (2015). https://doi.org/10.1109/ICSE.2015.85

[29] Ringer, T., Palmskog, K., Sergey, I., Gligoric, M., Tatlock, Z.: QED at large: A survey of engineering of formally verified software. Foundations and Trends in Programming Languages **5**(2-3), 102–281 (2019) https://doi.org/10.1561/2500000045

[30] Lammich, P.: Refinement to imperative HOL. Journal of Automatic Reasoning **62**(4), 481–503 (2019) https://doi.org/10.1007/S10817-017-9437-1

[31] Zhan, B., Haslbeck, M.P.L.: Verifying asymptotic time complexity of imperative programs in Isabelle. In: Proc. International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science, vol. 10900, pp. 532–548. Springer, Oxford, UK (2018). https://doi.org/10.1007/978-3-319-94205-6_35

[32] Nong, G.: Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. ACM Transactions on Information Systems **31**(3), 15 (2013) https://doi.org/10.1145/2493175.2493180

[33] Li, Z., Li, J., Huo, H.: Optimal in-place suffix sorting. Information and Computation **285**, 104818 (2022) https://doi.org/10.1016/J.IC.2021.104818

[34] Goto, K.: Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In: Holub, J., Zdárek, J. (eds.) Proc. Prague Stringtology, pp. 111–125. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, ??? (2019). https://doi.org/arXiv:1703.01009v5

[35] Nunes, D.S.N., Louza, F.A., Gog, S., Ayala-Rincón, M., Navarro, G.: Grammar compression by induced suffix sorting. Journal of Experimental Algorithms **27**, 1–111133 (2022) https://doi.org/10.1145/3549992

[36] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proc. Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 2089, pp. 181–192. Springer, ??? (2001). https://doi.org/10.1007/3-540-48194-X_17

[37] Cheung, L., Moffat, A., Rizkallah, C.: Formalized Burrows-Wheeler transform. In: Proc. Certified Programs and Proofs, pp. 13–26. ACM, ??? (2025). https://doi.org/10.1145/3703595.3705883

[38] Cheung, L., Rizkallah, C.: Formalised Burrows-Wheeler transform. Archive of Formal Proofs (2025). https://isa-afp.org/entries/BurrowsWheeler.html, Formal

proof development

[39] Larsson, N.J., Sadakane, K.: Faster suffix sorting. Theory of Computer Science **387**(3), 258–272 (2007) https://doi.org/10.1016/J.TCS.2007.07.017

[40] Karp, R.M., Miller, R.E., Rosenberg, A.L.: Rapid identification of repeated patterns in strings, trees and arrays. In: Proc. Symposium on Theory of Computing, pp. 125–136. ACM, Denver, Colorado, USA (1972). https://doi.org/10.1145/800152.804905

[41] Farach, M.: Optimal suffix tree construction with large alphabets. In: Proc. Symposium on Foundations of Computer Science, pp. 137–143. IEEE Computer Society, Miami Beach, Florida, USA (1997). https://doi.org/10.1109/SFCS.1997.646102

[42] Maniscalco, M.A., Puglisi, S.J.: An efficient, versatile approach to suffix sorting. Journal of Experimental Algorithms **12**, 1–211223 (2007) https://doi.org/10.1145/1227161.1278374

[43] Nipkow, T., Eberl, M., Haslbeck, M.P.L.: Verified textbook algorithms: A biased survey. In: Proc. Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 12302, pp. 25–53. Springer, Hanoi, Vietnam (2020). https://doi.org/10.1007/978-3-030-59152-6_2

[44] Appel, A.W.: Verified Functional Algorithms. Software Foundations, vol. 3. Electronic textbook, Online (2024). Version 1.5.5, http://softwarefoundations.cis.upenn.edu

[45] Bruns, D., Mostowski, W., Ulbrich, M.: Implementation-level verification of algorithms with KeY. International Journal on Software Tools for Technology Transfer **17**(6), 729–744 (2015) https://doi.org/10.1007/s10009-013-0293-y

[46] Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: Overview and VerifyThis competition. International Journal on Software Tools for Technology Transfer **17**(6), 677–694 (2015) https://doi.org/10.1007/s10009-014-0308-3

[47] Bobot, F., Filliâtre, J., Marché, C., Paskevich, A.: Let's verify this with Why3. International Journal on Software Tools for Technology Transfer **17**(6), 709–727 (2015) https://doi.org/10.1007/s10009-014-0314-5

[48] Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012: A program verification competition. International Journal on Software Tools for Technology Transfer **17**(6), 647–657 (2015) https://doi.org/10.1007/s10009-015-0396-8

[49] Dramnesc, I., Jebelean, T., Stratulat, S.: Certification of sorting algorithms using Theorema and Coq. In: Proc. International Symposium on Symbolic Computation in Softward Science. Lecture Notes in Computer Science, vol. 14991, pp. 38–56.

Springer, Tokyo, Japan (2024). https://doi.org/10.1007/978-3-031-69042-6_3

[50] Griebel, S.: Binary heaps for IMP2. Archive of Formal Proofs (2019). https://isa-afp.org/entries/IMP2_Binary_Heap.html, Formal proof development

[51] Sternagel, C.: Imperative insertion sort. Archive of Formal Proofs (2014). https://isa-afp.org/entries/Imperative_Insertion_Sort.html, Formal proof development

[52] Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.: Formally verifying an efficient sorter. In: Proc. Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 14570, pp. 268–287. Springer, Luxembourg City, Luxembourg (2024). https://doi.org/10.1007/978-3-031-57246-3_15

[53] Lammich, P.: Efficient verified implementation of Introsort and Pdqsort. In: Proc. International Joint Conference on Automated Reasoning. Lecture Notes in Computer Science, vol. 12167, pp. 307–323. Springer, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-51054-1_18

[54] Knuth, D.E., Jr., J.H.M., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing **6**(2), 323–350 (1977) https://doi.org/10.1137/0206024

[55] Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM **20**(10), 762–772 (1977) https://doi.org/10.1145/359842.359859

[56] Hellauer, F., Lammich, P.: The string search algorithm by Knuth, Morris and Pratt. Archive of Formal Proofs (2017). https://isa-afp.org/entries/Knuth_Morris_Pratt.html, Formal proof development

[57] Paulson, L.C.: Knuth–Morris–Pratt string search. Archive of Formal Proofs (2023). https://isa-afp.org/entries/KnuthMorrisPratt.html, Formal proof development

[58] Filliâtre, J.: Proof of imperative programs in type theory. In: Types for Proofs and Programs. Lecture Notes in Computer Science, vol. 1657, pp. 78–92. Springer, Kloster Irsee, Germany (1998). https://doi.org/10.1007/3-540-48167-2_6

[59] Boyer, R.S., Moore, J.S.: A Computational Logic Handbook. Perspectives in Computing, vol. 23. Academic Press, Cambridge, Massachusetts, USA (1979)

[60] Moore, J.S., Martinez, M.: A mechanically checked proof of the correctness of the Boyer-Moore fast string searching algorithm. In: Engineering Methods and Tools for Software Safety and Security. NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 22, pp. 267–284. IOS Press, Amsterdam, The Netherlands (2009). https://doi.org/10.3233/978-1-58603-976-9-267

[61] Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Proc. Interactive Theorem Proving. LIPIcs, vol. 141, pp. 22–12219. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Portland, Oregon, USA (2019). https://doi.org/10.4230/LIPICS.ITP.2019.22

[62] Noschinski, L., Rizkallah, C., Mehlhorn, K.: Verification of certifying computations through AutoCorres and Simpl. In: Proc. NASA Formal Methods Symposium, pp. 46–61 (2014). https://doi.org/10.1007/978-3-319-06200-6_4

[63] Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Proc. Interactive Theorem Proving. Lecture Notes in Computer Science, vol. 7406, pp. 99–115. Springer, Princeton, New Jersey, USA (2012). https://doi.org/10.1007/978-3-642-32347-8_8

[64] O'Connor, L., Chen, Z., Rizkallah, C., Amani, S., Lim, J., Murray, T.C., Nagashima, Y., Sewell, T., Klein, G.: Refinement through restraint: Bringing down the cost of verification. In: Proc. International Conference on Functional Programming, pp. 89–102. ACM, Nara, Japan (2016). https://doi.org/10.1145/2951913.2951940

[65] Rizkallah, C., Lim, J., Nagashima, Y., Sewell, T., Chen, Z., O'Connor, L., Murray, T.C., Keller, G., Klein, G.: A framework for the automatic formal verification of refinement from Cogent to C. In: Proc. Interactive Theorem Proving. Lecture Notes in Computer Science, vol. 9807, pp. 323–340. Springer, Nancy, France (2016). https://doi.org/10.1007/978-3-319-43144-4_20