Author/s:
Cheung, Louis Francis

Title:
Formally Verified Indexed Pattern Search

Date:
2025

Formally Verified Indexed Pattern Search

by

Louis Francis Cheung

ORCID: 0000-0002-3530-6662

A thesis submitted in total fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering and Information and Technology
School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

October 2025

THE UNIVERSITY OF MELBOURNE

Abstract

Faculty of Engineering and Information and Technology
School of Computing and Information Systems

Doctor of Philosophy

by Louis Francis Cheung
ORCID: 0000-0002-3530-6662

Indexed pattern search is essential for text retrieval, data compression, and bioinformatics, where reliability and accuracy are critical. However, existing algorithms often lack formal correctness guarantees. This thesis addresses that gap by applying formal verification to two core algorithms: the SA-IS algorithm for linear-time suffix array construction, a key approach in indexed pattern search; and the Burrows-Wheeler transform (BWT), an invertible transformation widely used in data compression and compressed indexed pattern search. Using the Isabelle/HOL proof assistant, we formally verify the correctness of these algorithms, enhancing their reliability for high-assurance applications. This establishes the necessary foundations for verifying efficient pattern search implementations.

Additionally, this thesis demonstrates how to compositionally verify software components across languages with varying levels of static guarantees, as part of the Cogent project. Cogent is a restricted functional language with ownership types that interfaces with C code to reduce the effort required for developing efficient verified systems code. This part of the work demonstrates how one can compose automatically generated Cogent proofs with manual C proofs to obtain an overall verification of a multi-language system.

Declaration of Authorship

I, Louis Francis Cheung, declare that this thesis titled, "Formally Verified Indexed Pattern Search" and the work presented in it are my own. I confirm that:

- The thesis comprises only my original work towards the degree of Doctor of Philosophy except where indicated in the preface;
- due acknowledgement has been made in the text to all other material used; and
- the thesis is fewer than the maximum word limit in length, exclusive of tables, maps, bibliographies and appendices as approved by the Research Higher Degrees Committee.

Signed: _____

Date: _____

Preface

This thesis contains original research in Chapters 3 to 6. The following applies to all the works unless stated otherwise:

- Co-authorship has taken place in accordance with the Graduate Research Training Policy of the University of Melbourne.
- The author of the thesis, Louis Francis Cheung, co-conceived and co-designed the project.
- The author of the thesis, Louis Francis Cheung, carried out all the technical work.
- The author of the thesis, Louis Francis Cheung, wrote the initial drafts and responses to co-author, editor and reviewer feedback.
- The author of the thesis, Louis Francis Cheung, redrafted, edited and reviewed the final manuscript.
- The works were partially funded by the Australian Government Research Training Program Scholarships and Fee offsets.

Chapter 3 is based on the following paper:

> L. Cheung, A. Moffat, and C. Rizkallah. Formally Verified Suffix Array Construction. In *Journal of Automated Reasoning*, 69(3), no. pages 21, 2025. doi: 10.1007/s10817-025-09735-8.

This paper reports the method, proofs and findings for the first component of the primary project, and was published in the *Journal of Automated Reasoning* on 21/07/2025. The proofs were published in the peer-reviewed proof repository *Archive of Formal Proofs* on the 25/09/2024:

> L. Cheung and C. Rizkallah. Formally Verified Suffix Array Construction. *Archive of Formal Proofs*, September 2024. ISSN 2150-914x. https://isa-afp.org/entries/SuffixArray.html. Formal proof development.

The contributions of the co-authors to this component of work are as follows:

- A. Moffat redrafted, edited and reviewed the manuscript.
- C. Rizkallah co-conceived and co-designed the project, reviewed the technical work and redrafted, edited and reviewed the manuscript.

Chapter 4 is a revised version of the following paper:

> L. Cheung, A. Moffat, and C. Rizkallah. Formalized Burrows-Wheeler Transform. In *Proc. Certified Programs and Proofs*, pages 13-26. ACM, 2025. doi: 10.1145/3703595.3705883.

This paper reports the method, proofs and findings for the second component of the primary project, and was published in the peer-reviewed proceedings of the *International Conference of Certified Programs and Proofs* on the 10/01/2025. The proofs that are described in the chapter and paper were published in the peer-reviewed proof repository *Archive of Formal Proofs* on the 17/01/2025:

> L. Cheung and C. Rizkallah. Formalized Burrows-Wheeler transform. *Archive of Formal Proofs*, January 2025. ISSN 2150-914x. https://isa-afp.org/entries/BurrowsWheeler.html. Formal proof development.

The contributions of the co-authors to this component of work are as follows:

- A. Moffat redrafted, edited and reviewed the manuscript.
- C. Rizkallah co-conceived and co-designed the project, reviewed the technical work and redrafted, edited and reviewed the manuscript.

Chapter 5 is unpublished material that is not yet submitted for publication, but will be later in 2025. It reports the method, proofs and findings of the third and final component of the primary project. The proofs are included as an appendix (Appendix A), and is also included as a The contributions of the co-authors to this component of work are as follows:

- A. Moffat redrafted, edited and reviewed the manuscript.
- C. Rizkallah co-conceived and co-designed the project, reviewed the technical work and redrafted, edited and reviewed the manuscript.

Chapter 6 is a revised version of the following paper:

L. Cheung, L. O'Connor, and C. Rizkallah. Overcoming Restraint: Composing Verification of Foreign Functions with Cogent. In *Proc. Certified Programs and Proofs*, pages 13–26. ACM, 2022. doi: 10.1145/3497775.3503686.

This paper reports the method, proofs and findings of the secondary project, and was published in the peer-reviewed proceedings of the *International Conference of Certified Programs and Proofs* on the 11/01/2022. The contributions of the co-authors to this component of work are as follows:

- L. O'Connor redrafted, edited and reviewed the manuscript.
- C. Rizkallah co-conceived and co-designed the project, reviewed the technical work and redrafted, edited and reviewed the manuscript.

Acknowledgements

I would like to thank my supervisors Christine Rizkallah and Alistair Moffat for their guidance throughout my PhD candidature, and Toby Murray for additional guidance. I would also like to thank my family for their unconditional support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Pattern search is a computational task involving finding occurrences of a sequence of symbols or characters, known as a *pattern*, in some larger sequence, referred to as a *text*. As a simple example, one might wish to know the number of times the word "pattern" appears in this thesis. Pattern search is employed in many applications, such as text processing and computational biology, and is a fundamental problem in computer science. Hence, solving this problem correctly and efficiently is of great importance.

Due to the foundational nature of the problem, pattern search has been the object of much study. This has resulted in a wide array of algorithms that solve the various formulations of the problem, whilst possessing varying degrees of efficiency. One particularly efficient approach, known as *indexed pattern search*, preprocesses the text to construct indexing data structures that make it easier to search for patterns in the text. Importantly, the indexing data structure only needs to be constructed once, assuming that the text is not modified, amortizing the cost of its construction across all subsequent pattern searches.

One particularly useful indexing data structure is the *suffix array* [117]. This data structure stores suffixes of a text, represented by their location in the text, in sorted order. By doing so, all suffixes starting with the same prefix are located in one contiguous block. This is especially important in relation to pattern search, as finding the block of all suffixes starting with same pattern is equivalent to the pattern search problem. Finding such a block can be done efficiently even when using a straightforward algorithm.

For example, for a pattern of length m characters and text of length n characters, with $m \ll n$, that is m being significantly smaller than n, binary search can be used over the suffix array to find all locations where the pattern appears in the text in $\mathcal{O}(m \log n)$ time. Using additional data structures, such as an LCP array [117], which is an array

containing the length of the *longest common prefix* between two adjacent suffixes in a suffix array, the time complexity can be further reduced to $\mathcal{O}(m)$ [87]. Importantly, the suffix array can be constructed in linear-time in the length of the text, for constant sized alphabets [84, 94, 140, 141], that is, in $\mathcal{O}(n)$ time for a text of length n characters, thus making it an ideal indexing data structure. Moreover, since the suffix array is a sorted array of all the suffixes of a text and these suffixes are represented by their location in the text, that is, numeric values, the suffix array can be stored in $\mathcal{O}(n)$ space.

The Burrows-Wheeler transform (BWT) [35] is another key construct in indexing data structures. This invertible data transformation permutes sequences, producing large runs often only containing a few distinct symbols, making it ideal for lossless data compression [173], which it was originally invented for. The BWT has a one-to-one correspondence to the suffix array, and because of this, it is also used in various indexing structures [54–56]. It is widely used in applications dealing with especially large texts such as genome sequencing tools [105, 106]. Similar to the suffix array, the BWT can also be constructed in linear-time — in fact it is constructed from a suffix array — also making it ideal for indexed pattern search.

While both the suffix array and the BWT are well-suited for indexed pattern search due to their characteristics and linear-time construction, both lack rigorous correctness guarantees for their construction. This means that their construction may contain potentially undiscovered oversights that could lead to software vulnerabilities, and even catastrophes, especially in safety-critical systems, such as processing medical data in the context of research and diagnosis [40, 47, 96, 105, 106, 111, 112, 159]. Due to the complex nature of the construction algorithms, it is difficult to develop a deep understanding of these algorithms and the problems they solve without rigorous study.

Formal verification provides a solution to the lack of rigorous correctness guarantees and understanding of algorithms and their implementations. It achieves this by constructing functional correctness specifications of the problem, encodings of the algorithms, and proofs of the correctness of the algorithm encodings with respect to the specifications. Importantly, all of these are encoded in formal logic, that is mathematics, with meticulous rigor, such that there are no ambiguities nor undefined behaviors in the specifications and algorithm encodings. Moreover, the proofs consist of exhaustive logical steps from first principles, that is core axioms of the logic, through to the desired conclusion. All of these logical steps are checked for soundness by trusted mechanical proof assistants [25, 129, 137].

Formally verifying algorithms used in indexed pattern search — particularly those for

constructing suffix arrays and the BWT — provides not only rigorous correctness guarantees but also deepens the theoretical understanding of the algorithms and the underlying problem. Such verification enhances the reliability, security, and robustness of systems that depend on these structures and may even reveal opportunities for improving the algorithms themselves. Formal verification has already proven effective in establishing the correctness of numerous algorithms [14, 138], occasionally uncovering previously unknown bugs in the process [24, 45]. However, despite the importance of indexed data structures, their formal verification remains relatively underexplored. This gap presents an opportunity to extend the guarantees of correctness and trustworthiness to a critical class of algorithms used in high-performance and safety-critical applications.

The primary component of this thesis addresses this gap by formally verifying core components in indexed pattern search in the mechanical proof assistant Isabelle/HOL [137]. The specific contributions of this component of the thesis are as follows:

- The first section of this work presents the first formal verification of the *Suffix Array Construction by Induced Sorting* (SA-IS) algorithm [140], a linear-time suffix array construction algorithm. The verification, detailed in Chapter 3, begins with a formal characterization of suffix arrays that serves as the functional correctness specification. This chapter then presents the formal verification of the SA-IS algorithm, starting with the formalization of key properties about sequences and suffixes that underpin the SA-IS algorithm, and finishing with the proof of correctness of the SA-IS algorithm.

- Building on the formalization of suffix arrays, this thesis presents the first formal verification of the BWT [35] and its inverse. The formalization, detailed in Chapter 4, contains two constructions of the BWT: one being its canonical construction and the other being a construction using suffix arrays, with both constructions being shown to be equivalent. The formalization also incorporates properties about the BWT, its correspondence to suffix arrays, and general properties for counting and locating values in lists. Finally, the formalization contains the verification of an efficient inverse transformation.

- To complete this component of the thesis, the formal verification of two indexed pattern search algorithms is then added, detailed in Chapter 5. These use the suffix array and the BWT as text indexes. The verification contains a formalization of the canonical pattern search problem, and a reformulation of the problem using suffix arrays. Using the pattern search formalization as a basis, the binary search algorithm on suffix arrays and the backward search algorithm [54], which is an indexed pattern search algorithm that uses the BWT, are formally verified.

These contributions collectively establish a robust, formally verified foundation for indexed pattern search. Thus, paving the way for the development of demonstrably correct and efficient implementations, enhancing the reliability of high-assurance applications that use them. By addressing the challenges of formalizing and verifying SA-IS algorithm and the BWT, this thesis not only guarantees their correctness but also deepens our understanding of indexed pattern search and approaches to solve it.

Developing demonstrably correct implementations from the formally verified algorithms and data structures is an orthogonal problem that is actively being researched. Given recent advancements, it is highly likely that with further maturation formally verified implementations can be automatically generated from formally verified algorithms and data structures [97, 148]. In turn, that will allow the formally verified suffix array and BWT construction algorithms presented in this thesis to eventually result in formally verified implementations.

However, it is highly unlikely that such implementations will be used in isolation. That is, these implementations will most likely be a component of larger software applications, where other components may be implemented in different languages with varying levels of static guarantees and expressiveness. Guaranteeing the correctness of such applications is quite difficult, as verifying each individual component does not necessarily mean that their correctness guarantees can be composed to obtain correctness of the whole application. While verifying the entire application as one monolithic structure would result in demonstrating its correctness, this would be a huge undertaking.

The final component of this thesis addresses this issue by demonstrating how to compose verifications of components of a multi-language system. This work is set in the context of Cogent and C. Cogent [148] is a restricted functional language designed to reduce the cost of developing verified systems code. It has a certifying compiler [165] that produces an embedding of the Cogent program in Isabelle/HOL, C code and a proof (also in Isabelle/HOL) that show that the C code refines the Cogent program embedding. Due to some of its restrictions, such as the lack of support for recursion and its strict uniqueness type system, Cogent provides an escape hatch in the form of a foreign function interface (FFI) to C code. This thesis demonstrates how one can compose automatically generated Cogent proofs with manual C proofs to obtain an overall verification of a multi-language system.

Specifically, this component of the thesis demonstrates how to compositionally verify Cogent and C components for word arrays and generic loop combinators written in C, and programs, such as binary search and file system operations, written in Cogent. The main contributions of this section of the work, which is presented in Chapter 6, are a reusable library of formally verified word arrays and generic loop combinators in C that

comply with Cogent's FFI constraints; verified implementations of binary search and a sum-list program in Cogent, built on top of this C library; and the discharge of proof obligations related to word arrays in the formally verified BilbyFs file system [9], which is a file system that is implemented in Cogent and C.

Together, these contributions demonstrate the feasibility of developing mixed-language applications that have strong guarantees of correctness for the whole application. They demonstrate how to verify components developed in multiple languages with differing levels of static guarantees and expressiveness.

Chapter 2

Background

This chapter provides a broad overview of the substring search problem and formal verification, laying the foundation for this thesis. Section 2.1 introduces the substring search problem and surveys the different approaches that are used to solve it. Section 2.2 then provides an overview of formal verification, with a focus on its foundational concepts and its application to proving the correctness of algorithms and programs. We finish this chapter with a summary of the thesis projects and a description of the tools used in these projects. Note that in-depth descriptions of the algorithms that are verified in this thesis are presented in the relevant chapters.

2.1 The Substring Search Problem

Many real-world problems [40, 42, 96, 105, 106, 111, 112] involve some element that relates to efficient search and pattern recognition. For instance, browsing the internet relies on retrieving relevant textual information from vast repositories for a given set of search terms, while DNA screening for genetic disorders or ancestry involves comparing genetic sequences against a database of known indicators. In both of these cases, and in many other applications, the underlying challenge is to efficiently identify occurrences of specific patterns within large datasets [106, 111, 188]. These pattern search problems, at their core, are based on the substring search problem [41].

The substring search problem, in its basic form, consists of searching for an instance of a pattern P in a text S, where patterns and texts are both sequences of symbols drawn from an alphabet Σ, which could be characters, words or even bits. Often, the substring search problem is extended to include counting all matching instances of P in S, and providing additional information such as the locations of the instances within S. More

advanced versions may also allow for inexact matches [42, 191], where search results may allow for minor differences.

There are many ways to solve the substring search problem. However, the approach taken greatly depends on constraints determined by the context of the search, such as, how often the text is modified, and the type of search queries made. Broadly speaking, there are three main approaches: brute force, that is, comparing all substrings of the text with the pattern; preprocessing the pattern to simplify the search; and text indexing where the text is preprocessed in a way that makes it easier to search. In general, text indexing approaches are much faster for each search, but require substantial preprocessing before the first search query can be carried out. The alternative approaches may be used instead in cases where the indexing cost cannot be amortized over sufficiently many search operations.

The following subsections provide an overview of various approaches in the context of exact substring search. It is assumed, unless stated otherwise, that all sequences are drawn from an alphabet Σ that is finite and has a total ordering. Hence, Σ has a one-to-one correspondence with the set of natural numbers from 0 up to but not including $\|\Sigma\|$, where $\|A\|$ means the cardinality of the set A. The asymptotic complexity of the substring search approaches described in the following subsections is defined using the random-access machine (RAM) model [39, Chapter 2.2]. Hence, integers and also symbols from Σ require $\mathcal{O}(1)$ space, that is, constant space, and can be loaded, stored and copied in $\mathcal{O}(1)$ time. Moreover, basic arithmetic, and integer and symbol comparisons take $\mathcal{O}(1)$ time. For more details on substring search algorithms and other related algorithms, refer to Crochemore et al. [41].

2.1.1 The Brute Force Approach

To search for a single instance of pattern P in a text S, the brute force approach compares P with the substring at each position in S until a match is either found or there are no more substrings of S to compare with. Naturally, if there is match, then P is a substring of S, and if there are no matches, then P is not a substring of S.

An example of the brute force approach for the single instance case is shown in Figure 2.1. Consider the pattern $P =$ `aba` and text $S =$ `aabbababa$`. Then the substrings S of length three at each position are `aab`, `abb`, `bba`, `bab`, `aba`, `bab`, `aba` and `ba$`, and these are compared with P in that order. In this example, five string comparisons occur, with the first four comparisons with the substrings `aab`, `abb`, `bba` and `bab` failing, and then the fifth comparison with `aba` succeeding. To find all occurrences of P, the search may

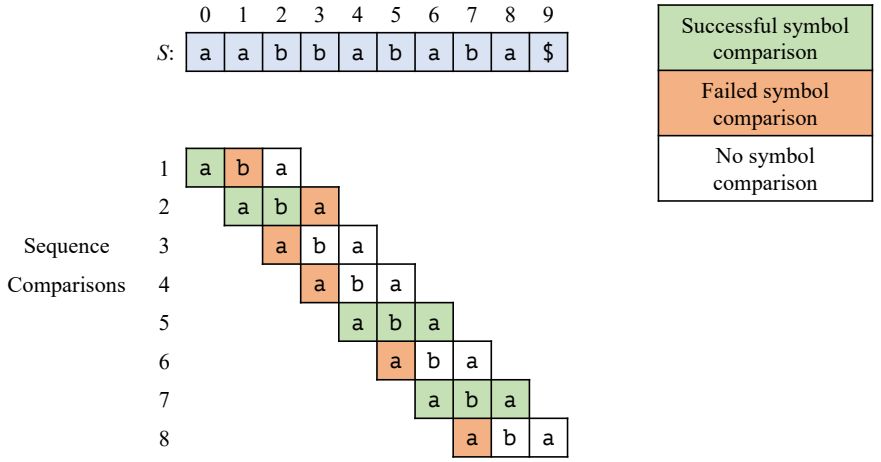


FIGURE 2.1: Brute force substring search example for the text $S =$ `aabbababa$` and pattern $P =$ `aba` with string indexing starting at 0. Green colored boxes indicate successful symbol comparisons, orange colored boxes indicate failed symbol comparisons, and uncolored boxes indicates no symbol comparisons.

proceed with remaining substrings, `bab`, `aba` and `ba$`, failing for the sixth and eighth, but succeeding for the seventh string comparison.

The brute force approach compares every substring of S with P until a match is found. When viewed in terms of the symbol comparisons with S, the brute force approach backtracks over S every time a symbol comparison fails. When P is short, this backtracking is negligible. However, as the length of P grows, so does the cost of the backtracking. Moreover, if there exist many prefixes of P in the text S, this cost might become severe.

For example, consider the earlier example, shown in Figure 2.1, with pattern $P =$ `aba` and text $S =$ `aabbababa$`. When the comparison with the substring at position 0, that is, `aab`, fails, the failure occurs due to mismatch between the second symbols of S and of P. Ideally, the next symbol comparison should continue from position 2, retaining knowledge gained from the previous failed comparisons to determine if the substring at position 2 is viable. However, the brute force approach does not retain any state information, and is forced to backtrack to position 1, beginning the comparison with P again.

If the length of P is m and the length of S is n, then the brute force approach takes $\Theta(mn)$ time in the worst case. This is due to the case where there may be many occurrences of prefixes of P in S, resulting in close to m symbol comparisons for each string comparison. For example, if $S =$ `aaaaaaaaa...aa` and $P =$ `aaaa...ab`. Such worst-case scenarios typically occur when a text contains many repeated substrings, such as DNA sequences. In the case that the alphabet size is greater than one, and that the symbols of the alphabet are both uniformly and independently distributed, the string comparisons tend to fail early. This results in an average time complexity of $\Theta(n - m)$ [41, Chapter 1.5].

While the time cost for the brute force approach may be high for large patterns, its space cost is virtually non-existent. This is because it does not store any additional information besides loop counters, which use $\mathcal{O}(1)$ space, and only uses the given pattern and text in the computations.

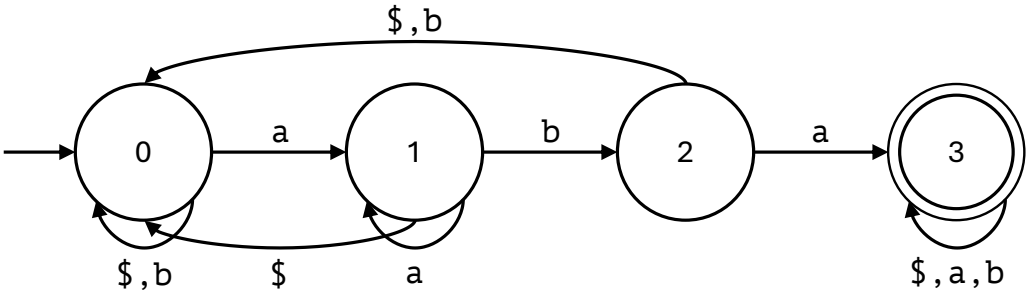
The brute force approach is simple to implement and requires no additional arrays or sequences. Moreover, for a short and fixed-length pattern P, the time cost is linear in terms of the length of the text S, making it a viable approach to use in this scenario. However, when S contains many repeated substrings and the length of P grows as S grows, that is, if S is of length n and P of length $m \approx kn$ for some constants $\epsilon < k < 1$, the time cost approaches $\mathcal{O}(n^2)$, making it unsuitable for this scenario. For other exact match search queries, the brute force approach is generally not ideal. For example, searching for all instances of a pattern requires the whole text to be searched. In addition, searching for multiple patterns is the same as searching for each pattern individually, resulting in repeated passes over the text.

2.1.2 Pattern Preprocessing Approaches

Preprocessing the pattern before searching reduces the number of redundant comparisons and increases both the theoretical and practical speed of the pattern search. Naturally, this approach requires additional time and space to preprocess and store the information obtained from the pattern. Ideally, these costs are proportional to the pattern length, with the pattern being scanned once and only information about the elements of the pattern being stored. The pattern preprocessing is independent of the text, and hence, the text can be modified without affecting this process. As such, search using pattern preprocessing approaches is ideally suited for single instance pattern searches on texts that are subject to change, or when there is limited memory availability. While there are various pattern preprocessing approaches, these can be divided into two main categories: preprocessing aimed at reducing redundant symbol comparisons and preprocessing aimed at replacing string comparisons with faster comparisons. The following subsections provide overviews of the two approaches.

2.1.2.1 Reducing Redundant Symbol Comparisons

Recall that the brute force approach backtracks over the text when a symbol comparison fails during a string comparison. The backtracking results in additional redundant symbol comparisons. One approach to pattern preprocessing is to avoid the redundant symbol matches (to varying degrees) by using information that is obtainable purely from



(A) The finite state machine representation of the DFA. States are represented by circles labelled with numbers. Transitions are represented by arrows labelled with symbols. The initial state is denoted by the state that has an unlabeled transition and the final state is denoted by the double border.

		Symbols		
		$	a	b
States	*0*:	*0*	1	*0*
	1:	*0*	1	2
	2:	*0*	**3**	*0*
	3:	**3**	**3**	**3**

(B) The transition lookup table representation of the DFA. States are represented by numeric values and actions are represented by symbols from Σ. The cell at the i^{th} row and column x contains result of the transition from the i^{th} state due to encountering symbol x. The initial state is *italicized* and the final state is in **bold** font to differentiate them from the intermediate states.

FIGURE 2.2: The DFA corresponding to the pattern $P = \texttt{aba}$ over the alphabet $\Sigma = \{\texttt{\$}, \texttt{a}, \texttt{b}\}$.

analyzing the pattern P and the alphabet Σ that the symbols of P and the text S are drawn from. There are several algorithms that fall in this category [8, 31, 93].

Each algorithm use heuristics that minimize the number of redundant symbol comparisons, and space and time costs. While the specifics of the approaches vary, they all essentially attempt to construct or approximate a deterministic finite state automaton (DFA). A DFA is a mathematical model of a computation consisting of states and deterministic transitions from states to other states due to some action or symbol. The DFA specifies a language by virtue of some states being denoted as accepting or final states. In the case of substring search, the language is the set of strings that contain the pattern. That is, if the text contains the pattern, then it will be part of the language accepted by the DFA, and rejected otherwise. Crucially, there is no backtracking when using a DFA and only a single pass over the text is needed to determine if a pattern is a substring.

For example, consider Figure 2.2 and the sequence $S = \texttt{aabbababa\$}$. If we wish to search for the pattern $P = \texttt{aba}$ in S, we can use the DFA starting in the initial state and following the transitions indicated by the elements of S. That is, starting in state 0, we move to state 1 due to encountering an `a`. Then, we stay in state 1 after encountering another `a`. We continue scanning over S, applying the relevant transitions until either we reach the final state, or finish scanning S. In this case, we end up with the trace 0, 1, 1, 2, 0, 1, 2 and 3, where 3 is the final state, meaning that `aba` is a substring of S. Notice that there is no backtracking, as we make use of the fact `a` is both a prefix and suffix of `aba`.

Figure 2.2 demonstrates two different representations of the DFA, The top figure being the classic finite state machine visualization, and the bottom figure being the transition lookup table from source states and symbols to destination states. While the former can be encoded as a linked list structure, where states are nodes and transitions are pointers to other nodes associated with a symbol, the latter is the form that is most likely to be implemented. This is because it can be encoded as either a 2D array or hash table, where the 2D array requires mappings from states and symbols to integers starting from 0, and the hash table uses states and symbols as keys. The worst-case space complexity of the 2D array representation is $\mathcal{O}(m\sigma)$ for a pattern of length m and alphabet Σ of size σ, as there are m states, corresponding to prefixes of the pattern, and each state has an outgoing transition for each symbol in Σ. The worst-case lookup time complexity is $\mathcal{O}(1)$ per symbol processed from S.

While the DFA ensures that there are no redundant symbol comparisons with the text, resulting in the text being scanned at most once, the construction of the DFA of a pattern P of length m is expensive. This is because the standard approach is to first construct the m state nondeterministic finite state automaton (NFA) of the pattern P (technically, the regular expressions $.^*P.^*$, where $.^*$ means any symbol zero or more times) using an algorithm like Thompson's construction [178] that runs in $\mathcal{O}(m)$ time. The NFA is then converted to a DFA, potentially containing 2^m states, using an algorithm like the subset construction [162], which runs in $\mathcal{O}(2^m)$ time. Since some of the states may be unreachable or redundant states, the DFA is minimized using an algorithm like Hopcroft's algorithm [76, Section 4.4.3], which runs in $\mathcal{O}(k\sigma \log k)$ time, where k is the number of states of the input DFA and σ is the size of the alphabet. Combined, the worst-case time complexity of constructing a DFA of a pattern of length m is $\mathcal{O}(m + 2^m + 2^m\sigma m \log 2)$, which reduces to $\mathcal{O}(2^{m+1}\sigma)$, and the worst-case space complexity is $\mathcal{O}(2^m\sigma)$, assuming that the DFA is represented as a 2D array.

The construction of the DFA and its size are major bottlenecks. To overcome these issues, other pattern preprocessing algorithms approximate the DFA using various heuristics,

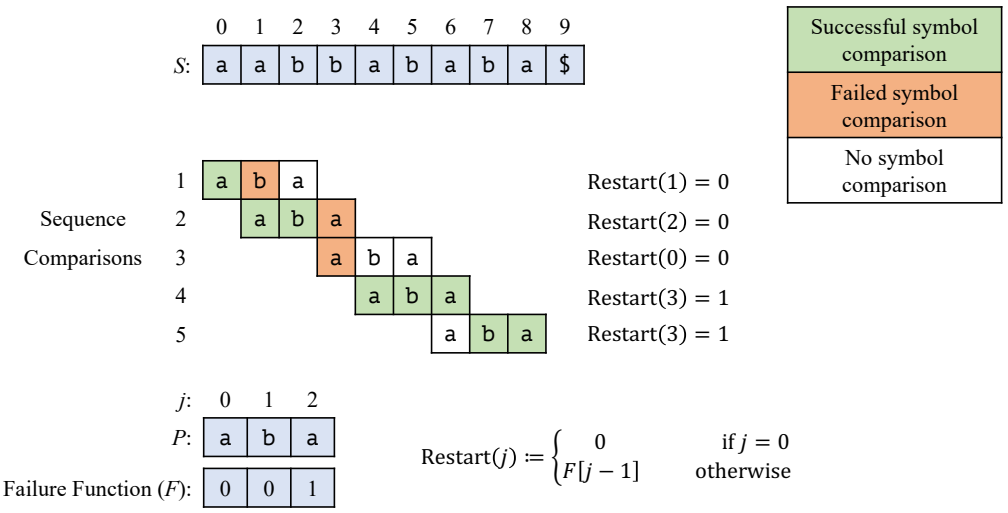


FIGURE 2.3: A visualization of the KMP algorithm searching for the pattern $P =$ `aba` over the text $S =$ `aabbababa$`, where Restart(j) indicates which location of P to start comparing from after a failed comparison at $P[j]$.

making trade-offs between construction and search costs. Two notable pattern preprocessing algorithms are the Knuth-Morris-Pratt (KMP) algorithm [93] and the Boyer-Moore (BM) algorithm [31].

The KMP algorithm [93] approximates the DFA by constructing a *failure function* that indicates which location in the pattern to start comparing from after a failed symbol comparison, as demonstrated in Figure 2.3. The failure function is computed by finding the length of the longest proper prefix that is also a suffix for each prefix of the pattern. For a text S of length n and pattern P of length m, the KMP algorithm has a worst-case search time complexity of $\Theta(n)$, as each element of S is only compared at most twice. The worst-case pattern preprocessing time is $\Theta(m)$, as the pattern is scanned at most twice. To store the failure function, $\mathcal{O}(m)$ space is required.

The BM algorithm [31] uses two heuristics, known as the *bad character* and *good suffix* heuristics, that determine how far to shift the pattern when a mismatch occurs. The algorithm maximizes the number of pattern shifts, by taking the maximum of the two heuristics, as demonstrated in Figure 2.4. Note that the BM algorithm differs from the DFA approach and KMP algorithm, as it starts comparing the pattern from the pattern's end. When a mismatch occurs for a symbol x in the text, the bad character heuristic shifts the pattern so that it aligns the last occurrence of x in the pattern. If x does not occur in the pattern, then the pattern is completely shifted past that symbol in the text. The good suffix heuristic shifts the pattern so that it aligns the longest matching suffix with an earlier occurrence. For a text of length n, pattern of length m and alphabet size σ, the BM algorithm has a worst-case search time complexity of $\mathcal{O}(mn)$, as there can be pathological cases, such as $S =$ `aaaa...a` and $P =$ `aaa...ab`, where both the bad

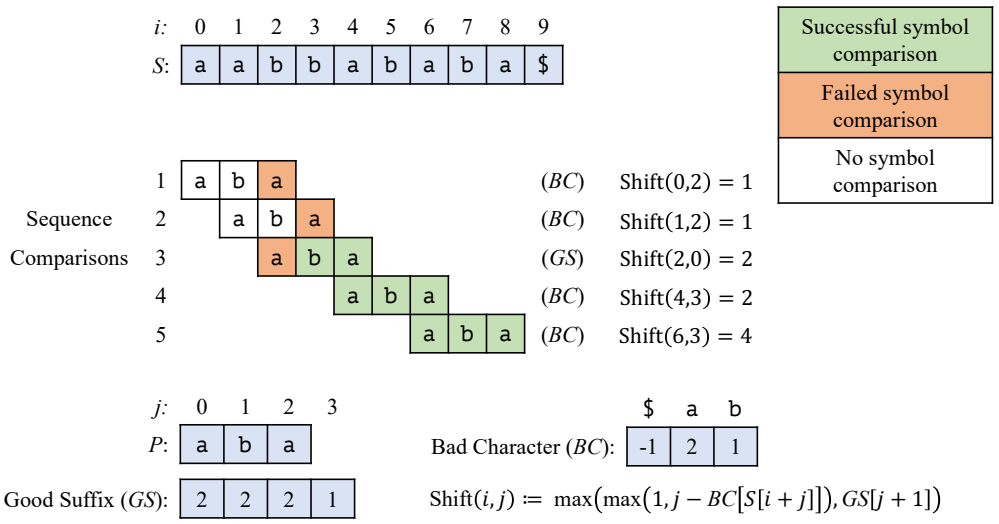


FIGURE 2.4: A visualization of the BM algorithm searching for the pattern $P =$ `aba` over the text $S =$ `aabbababa$`, where Shift(i, j) indicates how far to shift the pattern after a failed symbol comparison between $S[i + j]$ and $P[j]$.

character and good suffix heuristics only result in shifts of 1. The worst-case pattern preprocessing time complexity is $\Theta(m)$, as it scans pattern twice, and to store the bad character and good suffix heuristics, an array of length σ and an array of length $m+1$ are required, respectively, resulting in a worst-case space cost is $\mathcal{O}(m + \sigma)$. While the BM algorithm has a worst-case search time complexity of $\mathcal{O}(mn)$, which is the same as the brute force approach, and a worst-case space complexity of $\mathcal{O}(m+\sigma)$, which is larger than the brute force approach, the average-case search time complexity is sublinear when the symbols of the alphabet are uniformly and independently distributed [31]. Moreover, with modifications to the algorithm, the worst-case search time complexity can be reduced to $\mathcal{O}(n)$ [12].

2.1.2.2 Replacing String Comparisons

A second approach is to replace the expensive string comparisons in the brute force approach with cheaper alternatives, like numeric comparisons. Specifically, the new comparison is used as an initial filter, and only if this initial filter succeeds is the standard full string comparison used to verify the potential match. The most notable algorithm that uses this approach is the Rabin-Karp algorithm [85]. This algorithm does the initial filtering by comparing hashes of the substrings of the text with the hash of the pattern, where hash comparisons are basic integer comparisons.

The success of the Rabin-Karp algorithm depends heavily on the hash function used. That is, the hash function must satisfy the standard requirements of mapping values to fixed sized outputs, being deterministic and have reasonable collision resistance. The

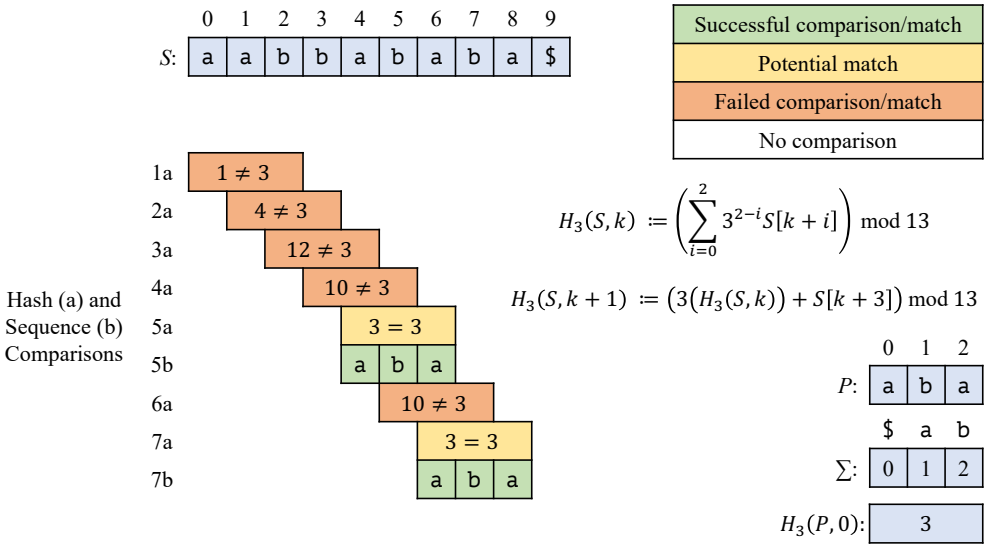


FIGURE 2.5: A visualization of the Rabin-Karp algorithm searching for the pattern $P =$ `aba` over the text $S =$ `aabbababa$`.

hash function must also be computable in constant time for each substring of the text. These requirements are achieved by using a rolling hash function, which is a hash function that uses the previous hash value to compute the next hash value for a window of values.

The rolling hash function that often used in the Rabin-Karp algorithm is the Rabin-Karp hash function [85, Section 3], but other hash functions, like the Rabin fingerprint [161], can also be used. The Rabin-Karp hash function essentially maps a sequence of length m to a polynomial of degree $m-1$, where the elements of the sequence are coefficients of the polynomial and m is also the length of the pattern that is being searched for. The polynomial is then evaluated at some base number a. Since the value that is computed can become quite large, the value is reduced modulo p, where p is some prime number, and the resulting value becomes the hash of the sequence. If the first element of the sequence is removed and a new element is concatenated to the end of the sequence, the hash can be updated by multiplying the previous hash by the base number then adding the new element, and then finally, reducing the value modulo p. Mathematically, the Rabin-Karp hash function is defined as follows:

Definition 2.1 (Rabin-Karp Hash Function)**.** Given an n-length sequence S of numbers, a prime number p, a base number a, and a hash window size m, the hash values produced by the Rabin-Karp hash function for the k^{th} and $k+1^{\text{th}}$ substrings of S of size m are:

$$H_m(S,k) := \left(\sum_{i=0}^{m-1} a^{m-1-i}(S[k+i])\right) \bmod p\,, \text{ and}$$

$$H_m(S,k+1) := H_m(S,k)a + S[k+m] \bmod p\,, \text{ respectively.}$$

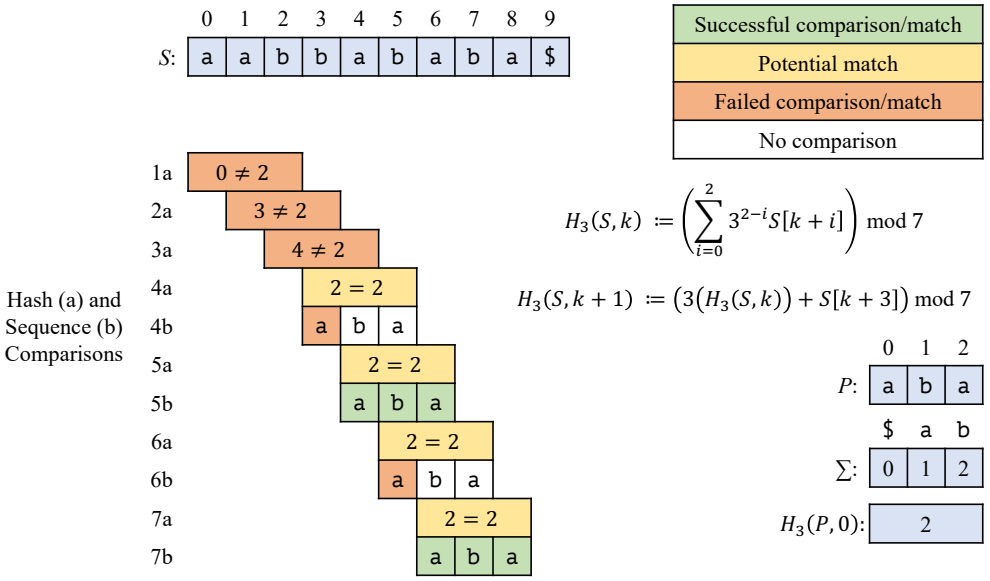


FIGURE 2.6: A visualization of the Rabin-Karp algorithm searching for the pattern $P =$ `aba` over the text $S =$ `aabbababa$` in the case of hash collisions.

Figure 2.5 demonstrates an example of the Rabin-Karp algorithm, using the Rabin-Karp hash function, for the sequence $S =$ `aabbababa$` and the pattern $P =$ `aba`. The prime and base numbers that are used in the hash are 13 and 3, respectively. In this particular example, no collisions occur, and so, both string comparisons that occur are those that validate true positives. Figure 2.6, however, demonstrates the case where collisions do occur for the same sequence, pattern, and base number, but for the prime number 7. In this case, the substring `bab` causes two collisions, which occur at locations 3 and 5 in S. Hence, to eliminate the false positives that occur at those locations, two additional string comparisons are used.

When searching for a pattern of length m in a text of length n, the worst-case time complexity of the Rabin-Karp algorithm is $\mathcal{O}(mn)$, with an additional once-off cost of $\mathcal{O}(m)$ for computing the hash for the pattern. This time complexity is due to the fact that the hash function may still have collisions, resulting in false positives that must be eliminated using the $\mathcal{O}(m)$ time string comparison. However, if the alphabet is uniformly distributed and the prime number p that is used in the hash function is randomly chosen from a sufficiently large set of primes numbers, then the probability of a collision is less than or equal to $2.511/(n-m+1)$ [85, Corollary 4a], resulting in an expected time complexity of $\mathcal{O}(n+m)$ [85, Theorem 5]. Since hashes are the only values that need to be stored, the worst-case space complexity is $\mathcal{O}(1)$, as only two hashes need to be stored at any one time. That is, the hash of the pattern and the hash of the current substring.

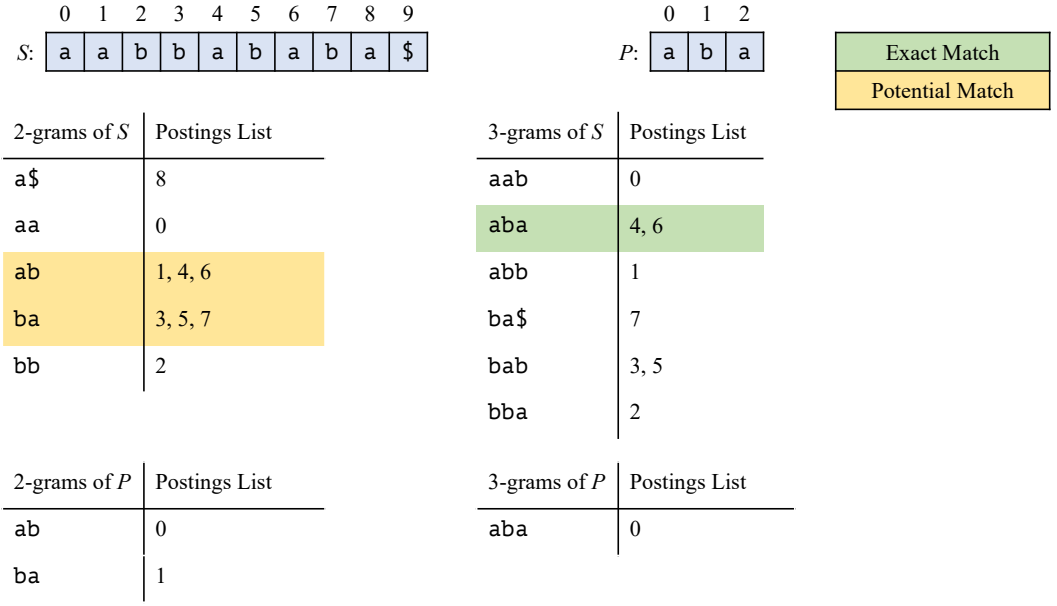


FIGURE 2.7: The inverted indexes of the string $S =$ `aabbababa$` for the 2-gram and 3-gram cases, with the potential and exact matches of the pattern $P =$ `aba` shaded green and yellow, respectively.

2.1.3 Text Indexing

Both brute force and pattern preprocessing approaches necessitate scanning the entire text S to find all existences of a pattern for every such search query. Hence, all such approaches will always have a running time of at least $\Omega(n)$ for a text of length n. Text indexing, on the other hand, can support faster search queries by preprocessing textual data, making it easier to locate symbols or substrings. Approaches to text indexing vary significantly as text indexing can refer to more than just structures used for substring search. Such examples include inverted indexes [42, Chapter 5.3] that map terms, such as words, to their positions in the text, and trie-like data structures [172, Chapter 5.2] that store the text in a tree structure.

The focus of this section is on text indexing approaches for substring search with a particular emphasis on inverted indexes based on *k-grams* [118, Chapter 3.2.2], *suffix trees* [69, Chapter 5], *suffix arrays* [117], and text indexes based on the *Burrows-Wheeler transform* [35]. In the following sections, it will be assumed that texts are all uniquely terminated by a symbol, denoted by `$` and called the sentinel, that is the smallest symbol in the alphabet Σ.

2.1.3.1 Inverted Indexing Based on k-grams

Inverted indexes [191] are a key data structure in information retrieval systems. These map terms, such as substrings, words or phrases, to a *postings list*, which is list of documents or positions in which the terms appear. When employed for exact substring search, one approach is to use *k-grams* [118, Chapter 3.2.2] as the terms of the inverted index. A k-gram is a substring of the text that is of length k. Hence, a k-gram based inverted index of a text S maps each distinct, k-length substring of S to a list of its occurrences in S. Figure 2.7 shows the inverted indexes of the text $S =$ `aabbababa$` using 2-grams and 3-grams.

To search for a pattern P using a k-gram-based inverted index of a text S, P is decomposed into a sequence of overlapping k-grams. As an example, consider Figure 2.7. When $P =$ `aba` and $k = 2$, the 2-grams of P are `ab` and `ba`, and when $k = 3$, the 3-gram of P is just `aba`. For each k-gram, the inverted index of S is queried to obtain the set of positions in which the k-gram occurs. These sets contain potential locations of P. To obtain the exact locations, a filtering step is required. A simple approach is to do a substring comparison with P for each potential location. Alternatively, the intersection of the sets could be taken, accounting for which k-gram of P a set corresponds to, to find the matching occurrences. For example, consider the 2-gram case of Figure 2.7. The set of locations of `ab` in S is $\{1, 4, 6\}$ and the set of locations of `ba` is $\{3, 5, 7\}$. Subtracting 1 from each element from the set for `ba` and intersecting this new set with the set for `ab`, results in the set $\{4, 6\}$, which are locations of P in S.

The effectiveness of the k-gram based inverted index approach depends on the data structured used to represent the index and the choice of k. Given that a text S of length n can have at most $n - k + 1$ distinct k-grams, and each distinct k-gram has a postings list associated with it, the data structure used to store the k-grams and postings lists takes is either $\Omega(3n - 2k + 2)$ or $\Omega((k + 1)(n - k + 1))$ or space. This is because the space needed to store all postings lists is at least $\Omega(n - k + 1)$ as the postings lists are lists of offsets into the text no two postings lists have shared elements. How the k-grams are stored depends on whether the text is stored with the index. If the text is stored with index, then offsets into the text can be used to represent a k-gram, resulting a $\Omega(2n - k + 1)$ space complexity. Otherwise, the k-grams need to be stored explicitly, resulting in a $\Omega(k(n - k + 1))$ space complexity.

For a pattern of length m, $m - k + 1$ queries are made to the inverted index. Notice that a smaller k results in more queries, while a larger k results in less queries. Moreover, a smaller k also results in more filtering as more false positives occur. While larger k are ideal, the space of the index becomes infeasible. Furthermore, the value of k is always

bounded by the pattern length, otherwise the search is no longer viable as the pattern does not have any k-grams.

A data structure that provides both efficient storage and efficient construction and lookup operations for k-gram based inverted indexes is the B-tree [22]. A node of the B-tree corresponds to a k-gram, and hence stores the representation of the k-gram and its associated postings list. Note that the postings list can be stored as an array or linked list of offsets into the text. The worst-case time complexity for looking up a k-gram is $\mathcal{O}(k \log n)$, where the k factor arises due to the comparison of k-length strings. The worst-case time complexity for construction is $\mathcal{O}(k(n-k+1)\log n)$ as each node insertion is $\mathcal{O}(k \log n)$ and there are at most $n-k+1$ nodes. Assuming that the k-grams are stored explicitly and the postings lists are stored as linked lists, the worst-case space complexity is $\mathcal{O}(k(n-k+1))$. This is because there are $n-k+1$ k-grams each requiring $\mathcal{O}(k)$ space, and the space needed for all the postings lists is $\mathcal{O}(n-k+1)$ as there are only $n-k+1$ offsets corresponding to a k-gram of the text.

When searching for a pattern P of length m using the B-tree implementation of the index, the lookup time cost for the k-grams of P is $\mathcal{O}((m-k+1)(k \log n))$, as the pattern P has $m-k+1$ k-grams and looking up each k-gram in the B-tree costs $\mathcal{O}(k \log n)$ [22] with k factor arises due to the comparisons between k-grams. The post-filtering process has a time cost of $\mathcal{O}(cL(m-k+1))$, where L is the longest postings list and c is the cost of filtering one candidate. Note that the exact cost for filtering is dependent on the filtering algorithm, and so we omit this as this is a whole other algorithmic issue [43]. Hence, the worst-case substring search cost is $\mathcal{O}((m-k+1)(k \log n + cL))$. Notice that larger values of k result in smaller post-filtering costs, with the largest value, where $k = m$, actually being a special case where there is no post-filtering is required, as demonstrated for the 3-gram case in Figure 2.7.

2.1.3.2 Suffix Trees

One of the most versatile structures for string search problems is the suffix tree [69, Chapter 5]. The suffix tree of text S is a tree data structure in which each edge is labelled with a non-empty substring of S; each internal node, excluding the root node, must have at least two child nodes; and all edge labels that originate from the same parent node must start with a distinct symbol. Moreover, every path from the root node to a leaf node maps to a suffix of S, where the concatenation of the labels on a path is the suffix, and this mapping is bijective. That is, every suffix of S also maps to a path in the suffix tree from the root to a leaf node. Due to this structure, the suffix tree can be used to solve many string search problems. For example, the paths from the root node

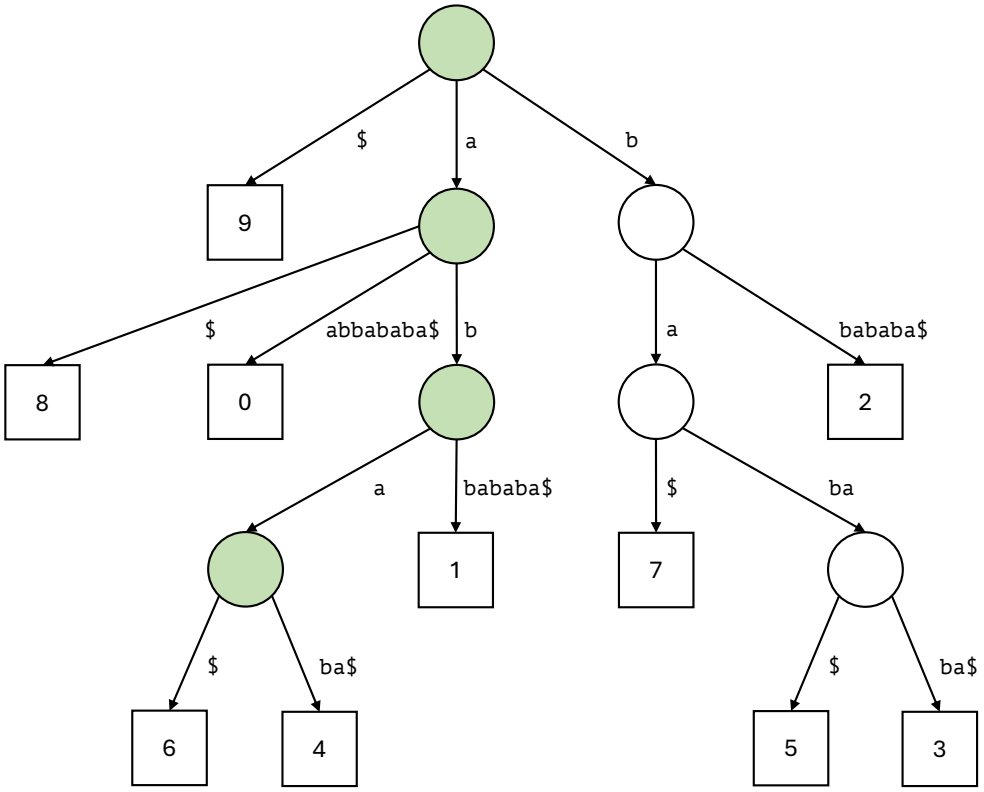


FIGURE 2.8: The suffix tree of the string $S =$ `aabbababa$` with the successful path corresponding to the pattern $P =$ `aba` colored in green.

to internal nodes in the suffix tree correspond to shared prefixes between suffixes, and so problems such as the longest common prefix can be easily solved [69, Chapter 7].

To solve the substring search problem, a pattern of length m is located (or not located) by traversing the suffix tree starting from the root while scanning the pattern. The criteria for selecting each next edge in the traversal is that its label is a prefix of the unscanned portion of the pattern, or the unscanned portion of the pattern is a prefix of the label. After selecting an edge, the unscanned portion of pattern is then updated accordingly. If no edge satisfies the criteria or there are no edges, that is, the current node is a leaf node, then the pattern does not occur in the text. Otherwise, the pattern occurs in the text and all the occurrences can be found by traversing the subtree down to its leaves.

Figure 2.8 demonstrates an example using the suffix tree of the text $S =$ `aabbababa$` to find the pattern $P =$ `aba`. The search for P begins by traversing the tree from the root, following the edges labelled with `a`, then `b` and then finally `a`; that is, the path colored green. Continuing the traversal to leaves then shows that there are two occurrences of the pattern at positions 4 and 6 in the text.

The worst-case time complexity for searching for a pattern of length m in a text of length n is $\mathcal{O}(m)$. This is under the assumption that the lookup and traversal operations on the suffix tree occur in $\mathcal{O}(1)$ time each. This can be achieved if the edges are stored in an array indexed by symbols of the alphabet. If a hash map is used instead, the expected times of the lookup and traversal operations are $\mathcal{O}(1)$, resulting in an expected search time of $\mathcal{O}(m)$. Since the search only uses the suffix tree, text and pattern, the space complexity, excluding the text and pattern, is at least $\Omega(n)$, as the size of the suffix tree is proportional to the length of the text. The upper bound depends on the representation of the suffix tree.

The maximum size of a suffix tree of a text S of length n in terms of the number of nodes is $2n - 1$. This is because the maximum number of leaf nodes is always n, as S has n distinct suffixes. Moreover, there is always only one root node. The maximum number of internal nodes is $n - 1$, as each internal nodes must have at least two child nodes, introducing at least one new branch to the suffix tree. Since each branch in the suffix tree must eventually lead to a leaf node, then k internal nodes results in at least $k + 1$ leaf nodes. Given that there are at most n leaf nodes, the maximum number of internal nodes is then $n - 1$. Note that since there are at most $2n - 2$ internal (not including the root) and leaf nodes, and each of these only has one incoming edge, the maximum number of edges is $2n - 2$

While the size of the suffix tree is linear in terms of the number of nodes (and also edges), the actual space required for each node can be much larger depending on how a node is represented. This because the root and internal nodes contain multiple pointers to other nodes, representing the outgoing edges, and these pointers need to be stored in a way that lookup and traversal are preferably $\mathcal{O}(1)$ time operations. One approach to achieve this is to have an array of size σ for each node, where σ is the size of the alphabet, that stores each edge in the cell that corresponds to the first element of its label. For example, in Figure 2.8, each node would have an array of size 3; and edges with labels starting with `$`, `a` and `b` would be stored at indices 0, 1 and 2, respectively. The array approach results in a $\mathcal{O}(\sigma)$ space cost for each node and $\mathcal{O}(n\sigma)$ for the suffix tree. Note that there is an additional cost per node to store the label of an incoming edge, that is, a substring of the text, but this can be achieved in $\mathcal{O}(1)$ space per node using two offsets that indicate where it occurs in the text.

Operations, like substring search, rely on the suffix tree having been pre-constructed. There exist several approaches for constructing suffix trees. These range from simple but inefficient for large texts, to efficient but quite complex to implement. The naïve construction, where suffixes are inserted into the tree one at a time, is simple but can result in a time complexity of $\mathcal{O}(n^2)$ for text of length n, as each suffix of S is scanned at least

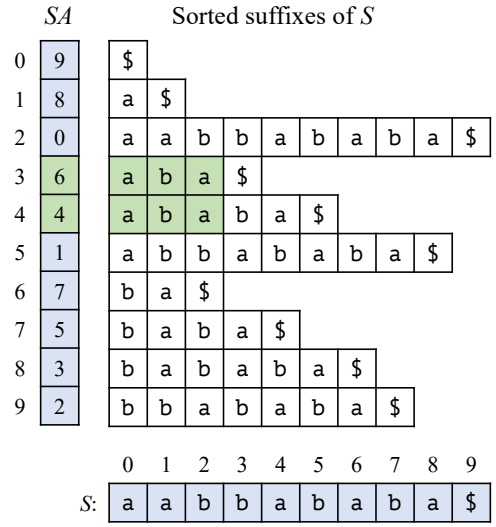


FIGURE 2.9: The suffix array (SA) and sorted suffixes of the string $S =$ `aabbababa$` with the block of prefixes matching the pattern $P =$ `aba` shaded in green.

once [69, Chapter 5.4]. More sophisticated approaches, on the other hand, can construct the suffix tree in $\mathcal{O}(n)$ time and space for constant sized alphabets. These algorithms use various approaches such as making use of underlying suffix tree properties [122], implicit ordering of suffixes [53] or incremental construction using implicit suffix trees [180].

2.1.3.3 Suffix Arrays

While the suffix tree can be used to solve many problems related to strings, its tree-like structure potentially requires multiple pointers per node, resulting in large memory requirements [98] that make it unsuitable for large texts. An excellent alternative to the suffix tree is the suffix array [117]. The suffix array of a text S is an array containing pointers to all the suffixes of S in sorted order. To save space, the suffixes are represented by their offsets in the text, that is, the suffix represented by i for S is the suffix located at position i or the i^{th} suffix of S, assuming that positions start counting from 0. Due to this representation of suffixes, suffix arrays have significantly lower memory costs compared to suffix trees, but the suffix array for a text might still be several times larger than the text itself. This is because a suffix array of text S of length n is an array containing all n offsets of S, and each offset needs at least $\log n$ bits of space to be represented.

The usefulness of the suffix array for substring search can be seen in Figure 2.9, where all the occurrences of the pattern $P =$ `aba` (colored green) in the text $S =$ `aabbababa$` are located in one contiguous block in the suffix array of S. This result is due to the suffixes being stored in sorted order. Storing suffixes in sorted order results in the shared prefixes of the suffixes being located in one contiguous block. Since these prefixes are also

substrings of the text, the substring search problem for suffix arrays reduces to finding the contiguous block of prefixes of suffixes that match the pattern. Locating this block is straightforward as the suffixes occur in sorted order, thus making it amenable to existing search algorithms over sorted arrays.

For example, all occurrences of a pattern P of length m in a text S of length n can be found by applying binary search [92, Chapter 6] to the suffix array of S. This because two applications of binary search can be used to find the locations of the first and last suffix in the suffix array of S that start with P, respectively, and these locations correspond the start and end of the block in the suffix array that contains all suffixes of S that start with P. Since binary search takes $\mathcal{O}(\log n)$ time for comparisons that take $\mathcal{O}(1)$ time, and comparisons with P take $\mathcal{O}(m)$ time in the worst-case, the worst-case time complexity for substring search is $\mathcal{O}(m \log n)$.

A suffix array of a text can be augmented with additional information that enhances its capabilities. The additional information is obtained by analyzing the text and the suffix array itself. With these enhancements, the suffix array is then able to fully replicate the capabilities of a suffix tree with the same time complexities and on average lower space complexities [2]. For substring search, a data structure known as the LCP array [117], which contains the length of the longest common prefix between two adjacent suffixes in the suffix array, is used to augment the suffix array, resulting in a worst-case time complexity of $\mathcal{O}(m)$ for a pattern of length m and a worst-case space complexity of $\mathcal{O}(n)$ for a text of length n [1]. The LCP array itself can be constructed from the suffix array in linear-time and space [87].

For suffix arrays to be usable in practice, they must be constructible in a reasonable amount of time. Simple approaches using comparison based sorting algorithms to sort suffixes have worst-case time complexities of $\mathcal{O}(n^2 \log n)$ time. This is because comparison based sorting algorithms require $\Omega(n \log n)$ [39, Chapter 8.1] comparisons, and each suffix comparisons can take $\mathcal{O}(n)$ time in pathological cases, such as highly repetitive texts like the text $S = \texttt{aaaa...a}$. While these simple approaches are sufficient for small texts, the quadratic factor in the time cost causes such approaches to become infeasible for large texts. Hence, more sophisticated approaches are required. There exist several construction approaches, and while the specifics vary, these can be classified into three general categories [160]: *prefix doubling*, *recursive* and *induced-copying*.

Prefix doubling algorithms [107] utilize a dynamic programming mechanism based on an approach developed by Karp et al. [86] for rapid identification of patterns. This is motivated by 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 $\mathcal{O}(n \log n)$ and the worst-case space

```
  0: .me.ee...,dsSgee,y,n;eyyyxfdpdaayoleen,g,ss,ymlskgeytsrhdhdnnhanresstd
 70: ddLertnnnnersdg:ysgfsnreoylyr,,n#Tn)nlsrtsAshrslsshesh$(#/#-$#OHI#W###
140: B-tt#siit#olr#mimvc####m#rh####rrtrg###eueeee##s#-hddmcmdtcpppppraaa#ia
210: #iiianerarrrina########iueecannneeeeen##nainnndhhhsrchrwhrmllnssssnrrs
280: sxxxht#herrbdlt#imslvhvvvttttvshehhnncr#rrr#riwddd#tkooeuiff####s###ns
350: #oiinnnnn#illll#cccgccnttttttWtwstt#TTttttllttflftwcrrrrhbbt#####loty
420: ysc#bstttstttes#hhlsxrrrrrllfc#aaaaa#blpeaaaaeep#biieaelprhrrrrrieiooo
490: hhhoiiaerrrroooaaaaaauIiii#ttiiiiiieoooaaoeaeaeiwtro##iccciiiiiiiicrf
560: fffcggggfffffccfHra####mppaaammmp#ooioeoeauuttaraaaaoaerr##pdpp#teeeo
630: ooocooooooeeeeppraoouetaiiinssmeeimcwnnemIse####usssnniUsseseeeeeeeeaa
700: eiin#sanxnaassnnftttt##########iiiinria-saacaaacc###esaaaa#iigrsocsB#e
770: ee###n#o##toieeeeetbecatrfllfl
```

FIGURE 2.10: The BWT of the first paragraph of this thesis' abstract with spaces replaced with # symbols.

complexity is $\mathcal{O}(n)$. The original suffix array construction algorithm by Manber and Myers [117] used this prefix doubling approach.

Recursive suffix array construction algorithms [84, 94, 141], reduce the suffix sorting problem for sequence S by constructing a new input sequence S', based on a subset of the suffixes. Then the suffix array of S' is constructed, which could either be achieved again by the recursive procedure or handled by some base case procedure when S' reaches a degenerate case, such as when its elements are all distinct. Using the suffix array of S', the order of the suffixes of S can be determined, and so, a linear sorting procedure, referred to as *induced sorting*, is used to order the suffixes of S, thereby obtaining its suffix array. These algorithms are asymptotically linear, having worst-case time and space complexities of $\mathcal{O}(n)$.

Induced-copying algorithms [58, 78] are similar to the recursive algorithms in that some subset of suffixes is sorted and then an induced sorting procedure is used to sort the remaining suffixes. However, rather than recursively sorting a subset of suffixes, these induce-copying algorithms sort the suffixes iteratively using general sorting algorithms. While such algorithms have worst-case asymptotic time complexities of $\mathcal{O}(n^2 \log n)$ [78] or $\mathcal{O}(n \log n)$ [58], depending on the algorithm, and worst-case space complexities of $\mathcal{O}(n)$, these tend to be fast and have small memory footprints in practice. For instance, the DivSufSort algorithm [58] is regarded as the most efficient algorithm in practice, outperforming the linear-time recursive suffix array construction algorithms [18].

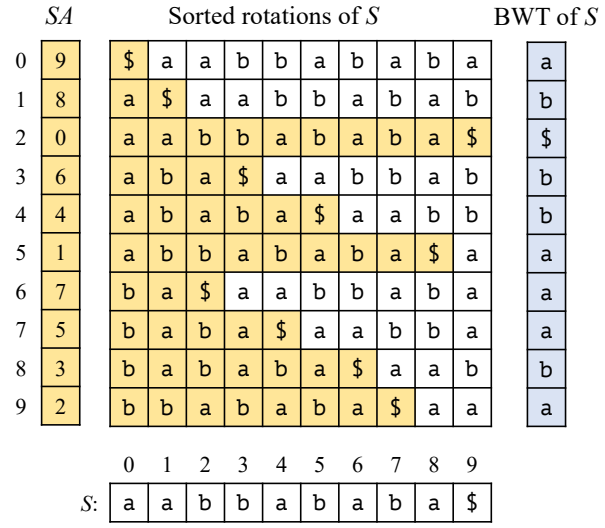


FIGURE 2.11: The suffix array (SA), sorted suffixes and rotations, and BWT of the string $S =$ `aabbababa$`. Note that last column of the sorted rotations is the BWT and its correspondence to the suffix array is highlighted by the colored sections of the sorted rotations, which are the sorted suffixes.

2.1.3.4 Text Indexes Based on the Burrows-Wheeler Transforms

When dealing with large and highly repetitive texts, such as DNA sequences that can be billions of base pairs long [38], the text indexes described earlier can become prohibitively large. An effective solution is to use text indexes that can be compressed and queried at the same time. Text indexes based on the Burrows-Wheeler transform (BWT) [35] are notable examples. The BWT is an invertible permutation that tends to produce long locally homogenous runs, resulting in those sections of having low-localized entropy. An example of the BWT is shown in Figure 2.10, which is the BWT of the first paragraph of this thesis' abstract. Notice that it contains several clusters like `fffcggggffffff`, starting at character 560, `ooocooooooeeee`, starting at character 630, and many others that are locally homogenous.

Due to the BWT being invertible and having long runs with low-localized entropy, it is often used in lossless data compression applications, typically prior to an entropy encoding step [40, 47, 159, 173]. In addition to data compression, the BWT is also used for substring search. This is because the BWT of a text is the last column of a matrix, whose rows are rotations of the text that are ordered lexicographically, as demonstrated by Figure 2.11 for the text $S =$ `aabbababa$`. Given that rotations and suffixes of a text have a one-to-one correspondence, the BWT and the suffix array of a text must also have a one-to-one correspondence. Figure 2.11 demonstrates this correspondence for the text $S =$ `aabbababa$`, where the sorted suffixes (colored yellow) are the prefixes of the sorted rotations.

Text indexes that are based on the BWT consist of the BWT of the desired text S and auxiliary structures that aid in quickly counting and locating occurrences of a symbol in the BWT of S [54]. Importantly, S is not used for substring search queries on S, but can be reconstructed from the BWT of S if desired. Moreover, the auxiliary structures can be computed from the BWT of S in linear-time, and hence, do not need to be stored (in the case for long term storage) or computed until required. Furthermore, both the BWT of S and auxiliary data structures can be compressed, albeit with some degradation in the speed of the operations that access them [55, 56, 116]. Even without compression, the BWT of S only takes as much space as S itself, since the BWT of S is a permutation of S. Hence, BWT based text indexes can be used for large and highly repetitive texts, such as collections of DNA sequences [96, 105, 106, 111, 112].

There are several approaches to construct the BWT of a text S of length n. The simplest approach is to generate all rotations of S, sort the rotations, and then take all the last elements from the sorted rotations. However, this approach results in a worst-case time complexity of $\mathcal{O}(n^2 \log n)$. This is because the worst-case time complexity for comparison-based sorting algorithms is $\Omega(n \log n)$ time [39, Chapter 8.1], assuming that the comparisons take $\mathcal{O}(1)$ time. Given that the comparison between rotations takes $\mathcal{O}(n)$ time, the time complexity for sorting rotations is thus $\mathcal{O}(n^2 \log n)$. The worst-case space complexity depends on the representation of the rotations, but will be at least $\Omega(n)$ as the BWT of S is a sequence of length n. If a rotation is represented as a sequence, then the worst-case space complexity is $\mathcal{O}(n^2)$, as there are n distinct rotations and each rotation is of length n. However, if a rotation is represented by the location of its first symbol in S, then the worst-case space complexity is $\mathcal{O}(n)$.

More efficient approaches exploit the one-to-one correspondence between the BWT and the suffix array, and can be computed from a suffix array with a single linear scan. Moreover, since there exist linear-time suffix array construction algorithms, the BWT can be constructed in $\mathcal{O}(n)$ time and space. Similar approaches that omit the suffix array but still rely on suffix sorting can also run in linear-time [150]. Other algorithms also exist that construct the BWT using dynamic/online approaches [75, 114], and these are typically used for real-time or streaming applications.

2.1.4 Summary and Relation to the Thesis

This section described various solutions to the substring search problem, beginning with a simple but inefficient brute force approach that compares every substring of the text of length n with the pattern of length m, resulting in a worst-case time complexity of $\mathcal{O}(mn)$. This was then followed by a description of pattern preprocessing approaches,

which for some approaches can achieve worst-case search time complexities of $\mathcal{O}(n)$ [93]. However, this is always accompanied by a pattern preprocessing cost that is proportional to the pattern length, resulting in a combined pattern preprocessing and search worst-case time complexity of $\Omega(n + m)$ per pattern. Finally, we described various text indexing approaches that preprocess the text to facilitate fast substring search, with some approaches having worst-case search time costs of $\mathcal{O}(m)$, such as suffix tree [69, Chapter 5], suffix array [1] and BWT approaches [54]. While the text index does need to be pre-built, resulting in an additional cost that is proportional to the text length, the suffix array [140] and BWT [35] text indexes can be constructed in $\mathcal{O}(n)$ time. Moreover, these text indexes only needed to be constructed once per text, meaning that every substring search after the initial search for a text has a $\mathcal{O}(m)$ worst-case search cost.

The brute force and pattern preprocessing approaches are easier to understand and implement, but have worse search worst-case running times than the text indexing approaches. The text indexing approaches, however, are more sophisticated, and hence, more difficult to understand and implement, especially for the construction of the text indexes. Informal correctness arguments exist for the construction and usage of the text index approaches. However, the nature of such arguments means that they are incomplete and lacking rigor, potentially resulting in undiscovered flaws or misunderstandings on how or why particular components work, all of which may result in software vulnerabilities. This thesis addresses that gap by developing rigorous correctness proofs for the algorithms used to solve substring search, focusing on suffix array construction, specifically the SA-IS algorithm [140]; the BWT and its inverse [35]; and finally indexed pattern search using these text indexes [54]. How this thesis accomplishes those proofs is described in the following section.

2.2 Formal Verification

When using any system, whether a program, algorithm or mathematical result, there is always the underlying assumption that it should be correct. Mainstream approaches that are used to validate the correctness assumption (or at least attempt to) are code inspection and testing, in the case of programs, and proofs written in prose that are checked by human experts, in the case of algorithms and mathematical results. However, such approaches are not always successful. For example, with code inspection and program testing, the industry average for the frequency of software bugs per 1,000 lines of code is between 1 and 25 [121, page 521], and 20% of medical device recalls in a five-year period prior to April 2020 by the TGA (Therapeutic Goods Administration) were due to software faults with this figure likely being an under-representation [5]. Similarly, prose

proofs checked by human experts may contain unstated assumptions or proof gaps, such as textbook descriptions of Dijkstra's shortest path algorithm [48] that contained implicit overflow assumptions [126], or the quantitative version of the Morse Lemma [175] that contained a gap in the proof for the central theorem [65]. Prose proofs may also be either erroneous or practically unverifiable by a human being, such as the numerous incorrect proofs and disproofs [186] of the Four Color Theorem [62] and the first complete computer proof of the Four Color Theorem [15, 16], respectively. With software, and hence the algorithms and mathematical results they implement, being so widespread, including in high integrity systems, alternative approaches are needed to ensure their correctness to avoid repeating catastrophes [44, 50].

Formal verification is a process that can be used to feasibly guarantee the correctness of programs, algorithms and mathematical results [115, 167]. It involves using formal logic proofs to demonstrate that a system satisfies its functional correctness specification, which rigorously describes its intended behavior. Both the specification and proofs are encoded in formal logic, thus removing much of the ambiguity and undefined behavior that arise from encoding them in prose. Moreover, the proofs are checked by a mechanical proof checker or proof assistant, which is a program that scrutinizes each proof step ensuring that each step can be inferred from the previous, and hence from the premises. Due to this, proofs also need to be comprehensive enough to facilitate this process. Note that many proof assistants are accompanied by some proof automation or other features that help users develop proofs [185].

When starting a formal verification undertaking, there are a few considerations that affect the approach taken, and may result in trade-offs between trustworthiness and feasibility of the approach [90]. These considerations include, among others, what behaviors the functional correctness specification contains and how to validate the specification; what logic is needed to encode or embed the specification, and hence the proofs; how trustworthy is the proof assistant; and how much automation it provides. Note that the last consideration is related to the fact that formal verification can be costly undertaking, as proof development can be complex. Hence, proof automation is very much a desired property as this can greatly reduce this cost.

The following subsections explore each of the considerations mentioned above with a particular emphasis on the verification of algorithms and programs. Before we begin, we first take a deeper look at the limitations of mainstream correctness guaranteeing approaches to highlight how formal verification overcomes those limitations.

2.2.1 Limitations of Mainstream Approaches

Program testing, unless it is exhaustive, only ensures that the tested subset behaves as expected with no guarantees on the untested subset [49]. On the other hand, exhaustive program testing does provide guarantees about all possible inputs, however, this is generally infeasible due to the inordinate number of test cases required. For example, the number of test cases that would be needed to show that a suffix array construction algorithm correctly constructs the suffix array of an arbitrary DNA sequence, such as the DNA of a human [145] that is over 3 billion base pairs long, would be over $(4^{3\times10^9} - 1)/3$, which is approximately $3.2 \times 10^{1806179973}$. Assuming that each check takes only one clock cycle and a 10 GHz CPU is used (where the fastest recorded overclocked CPU speed only reaches 9.12 GHz [133]) the time required to run each test would be at least $3.2 \times 10^{1806179963}$ seconds, which is approximately 1×10^{1806179955} years. Note the fastest suffix array construction algorithms execute in time proportional to the length of the sequence [64, 84, 94, 113, 139, 140], meaning that the numbers given for the time needed for testing is an unreachable lower bound.

Traditional prose proofs provide an intuitive understanding of algorithms and mathematical results, communicating their core ideas through the use of mathematical reasoning. Moreover, prose proofs avoid the state explosion issue that exhaustive testing experiences by quantifying over all possible inputs and outputs. While such proofs may boost our confidence in the correctness of the algorithm or mathematical result, the proofs may lack the necessary rigor and comprehensiveness to assuage all fears of reasoning leaps, missing cases, or implicit assumptions [175]. Moreover, these prose proofs rely on human experts to check that the arguments are sound. This approach is only feasible for small cases with the approach being infeasible for large, repetitive and complex proofs, due to the greater risk of human error [163]. For example, the initial proof of the Four Color Theorem had some 1,400 graphs that needed to be checked by a program (and hence manually inputted), and an additional prose proof that had to be checked by a human expert, which was described to be "extraordinarily complicated and tedious", resulting in the prose proof never being completely independently checked [166]. Note that code inspection [121, Chapter 21], which is where another human inspects a program for bugs, can be viewed as an undocumented prose proofs, and hence has the same limitations as a prose proof.

2.2.2 Functional Correctness Specifications

The most important task when formally verifying a system is defining its functional correctness specification, that is, what does it mean for the system to be correct. For

algorithms, this is the encoding of the algorithm, assumptions and the mathematical relation, which defines what output should arise given any particular input. For programs, this could either be something similar to what is required for algorithms, or a theorem that states that the program is a refinement of a previously verified algorithm, that is, every behavior of the program is also exhibited by the algorithm. Note that the latter is typically easier to show, as the refinement proof mainly focuses on dealing with implementation details in the program. Specifications of programs may also include other properties besides correctness, such as use of other resources including memory.

A useful functional correctness specification of an algorithm or program should be *rich*, *two-sided*, *formal* and *live* [13]. A specification is said to be rich if it describes the complex behavior of the system in detail. It is two-sided if it can serve both as a specification for a system and a replacement for the system in the specification of client systems. It is formal if it is encoded in a mathematical notation with an unambiguous semantics that is understood by the necessary tooling. Finally, it is live if it is connected to the system by a proof that is checked by a mechanical proof checker. Both the formality and liveness properties are necessary requirements for a formal verification, as being formal ensures that specification has the necessary rigor, and live means that the specification and its associated proof are independently checked by a trusted mechanical proof checker.

The richness requirement is a necessity due to the fact that algorithms and programs contain complex behaviors. However, satisfying the richness requirement requires a sufficiently expressive logic, with more complex algorithms and programs resulting in richer specifications, and hence requiring more expressive logics. Constraints on the choice of logic influence the choice of proof assistant and the level of proof automation that is available [13, 90]. More expressive logics typically require proof assistants that require more human interaction and less proof automation.

Satisfying the two-side requirement increases the likelihood that a specification of a system is itself valid, that is, whether the specification accurately encapsulate all the intended (and prohibited) behaviors of the system. This is because a specification of a some system A is likely to be discovered to be unsatisfiable during the verification process of A, and the verification of a client system B that uses A would also uncover flaws in the specification of A, such as whether it actually encapsulates the desired behaviors of A [13]. For example, a model of an algorithm can be a two-sided specification as it can be the specification for an implementation of a program, but also a high level implementation for a mathematical relation. Verifying that the program is correct with respect to the model of the algorithm, and verifying that the algorithm model is correct with respect to its mathematical relation specification would determine if the algorithm was correctly specified.

Other formal verification approaches can be used to validate a specification of a system, such as proving that the specification is equivalent to alternative formulations, inferring other related properties and corollaries from the specification, or proving the correctness of a simpler algorithm that solves the same problem with respect to the specification. Other techniques such as simulation and model checking can be used to make the specification executable, testing against sample scenarios [79, 104], or attempting to find counterexamples that satisfy the specification but violate the desired properties [19, 189]. Besides using sophisticated tooling, one can also do a careful comparison between a written translation of the specification and formal specification to check that these align.

2.2.3 Modelling Algorithms and Programs

Proofs must be encoded in a formal logic so that they can be checked by a mechanical proof checker. In the context of algorithms and programs, this means defining a model or embedding of these in a logic that the proof checker supports. A key requirement for this is that the language used to implement these must have a formal semantic or formalization defined in that logic, that is, each syntactic construct is assigned a logical meaning. For algorithms, this is not usually a major obstacle, as these are defined in pseudocode, meaning that as long as the logic is expressive enough, algorithms can be embedded, albeit with some modification. For programs, on the other hand, this can become a major obstacle as these are tied to specific programming languages.

To embed a program written some programming language into the logic of the proof checker, the programming language must have a formalization in the proof checker. That is, each syntactic construct of the programming language must be assigned a meaning within the logic used by the proof checker. It should be noted that not all programming languages have formalizations, and so, programs written such languages cannot be formally verified until that exists. For some languages, like C, only a subset has been formalized [142]. In this case, only programs that written in that subset can be formally verified.

Assuming that such language formalizations exist, a program (and also algorithm) can be embedded deeply, shallowly, or a mix of both. When a program is deeply embedded, it is modelled as a data type in the logic. When a program is shallowly embedded, it is represented using equivalent expressions in the logic. This difference between the two embeddings can be seen in the following arithmetic expression:

$$3 + 4 = 7. \tag{2.1}$$

Using basic arithmetic, it is obvious that the left-hand side of equation Equation (2.1) is equal to the right-hand side, that is, the left-hand side is semantically equivalent to the right-hand side. However, the left-hand side is structurally/syntactically different from the right-hand side as the string `3+4` is different to the string `7`.

Both types of embeddings have their strengths and weaknesses. Deep embeddings are ideal for proving structural properties, like time and space complexities, but are more difficult to use for proving semantic properties, that is, properties relating to the meaning of statements, such as correctness. Shallow embeddings, on the other hand, are better suited to proving semantic properties, but cannot be used to prove structural properties. When proving functional correctness of an algorithm or program, shallow embeddings are preferred because only the semantics of the program are necessary [131]. Moreover, shallow embeddings are easier to manipulate as these are expressions in the logic of the proof assistant, and hence can use the existing proof infrastructure. Note that deep embeddings can still be used to prove functional correctness, but require explicit interpretation or evaluation functions.

Whether or not it is easy to represent/embed an algorithm or program as a shallow embedding depends on how the algorithm is described, or what programming language is used for the program. Typically, if the language and logic have similar paradigms, then shallow embedding are more easily obtained. On the other hand, if the language and logic do not have similar paradigms, shallow embeddings are harder to obtain.

Since the logic of many of the more trustworthy proof checkers are based on abstract logics, like higher-order logic (HOL), algorithms and programs implemented in high level programming languages, like functional programming languages, are more readily translated to shallow embeddings, making it easier to verify these. Moreover, if these can be defined as pure functions, that is, functions with no side effects, then these can be handled like standard mathematical functions. This means that equational reasoning, that is, substitution, algebraic laws and semantic preserving transformations, can be used.

Algorithms and programs implemented in low level languages, like imperative languages, can still be verified, however, these are usually deeply embedded. To prove functional correctness, one would then use explicit interpretations to evaluate the deeply embedded expressions, and manually prove correctness with these. Alternatively, one could use tools, assuming they exist, that lift deep embeddings to more abstract embeddings, such as purely shallow embeddings, or mixed deep and shallow embeddings. Correctness is then proved on the more abstract embeddings, which is generally easier. Note that the actual logic used to reason about imperative algorithms and programs is usually one that is based on Hoare logic [74], which describes how a program alters the program state.

In Hoare logic, the key element is the Hoare triple, which is of the form

$$\{P\}\ C\ \{Q\}\,. \tag{2.2}$$

This describes how the execution of a program C changes the program state. In addition, if a Hoare triple is valid, then if the precondition P holds before C is executed, then the postcondition Q is established after C is executed. For a simple imperative language, Hoare logic has several axioms and inference rules for the main constructs of the language, and from these rules, rules for other language constructs can be derived. The key axioms and rules are the following:

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}} \tag{2.3}$$

$$\frac{}{\{P[E/x]\}\ x := E\ \{P\}} \tag{2.4}$$

$$\frac{\{P\}\ S\ \{Q\},\ \{Q\}\ T\ \{R\}}{\{P\}\ S;\ T\{R\}} \tag{2.5}$$

$$\frac{\{P \wedge B\}\ S\ \{Q\},\ \{P \wedge \neg B\}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}} \tag{2.6}$$

$$\frac{P \longrightarrow P',\ \{P'\}\ C\ \{Q'\},\ Q' \longrightarrow Q}{\{P\}\ C\ \{Q\}} \tag{2.7}$$

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \textbf{endwhile}\ \{P \wedge \neg B\}} \tag{2.8}$$

The Hoare axiom for the empty statement Equation (2.3) states that the **skip** statement does nothing and does not change the state. The assignment axiom Equation (2.4) states that if the assertion P, in which all occurrences of the free variable x have been replaced with the expression E, is true, then after the assignment $x := E$, the assertion P is true. The sequential composition rule Equation (2.5) states that for two programs S and T, if the postcondition of S is the precondition of T, then if the precondition of S is true, then after the program $S;\ T$ is executed, the postcondition of T is true. The conditional rule Equation (2.6) states that the postcondition Q, which is common to both branches of the condition statement, is established if the precondition P is true. The consequence rule Equation (2.7) allows the strengthening of the precondition and the weakening of the postcondition of a Hoare triple. The while rule Equation (2.8) states that if the assertion P is preserved by the execution of the program C, then even after repeated executions of C, P still holds assuming that P held initially. The assertion P is known as a loop invariant and materializing a suitable invariant that will actually help prove correctness is one of the harder aspects in formal verification.

With standard Hoare logic, only partial correctness can be proved, because in the Hoare triple Equation (2.2), if C does not terminate then the postcondition can be anything,

since it will never be established. Often, the rules in Hoare logic are extended to include termination so that it can be used to prove total correctness, that is, in the Hoare triple Equation (2.2), the program C terminates and so the postcondition is guaranteed to be established.

2.2.4 Proof Assistants

Mechanical proof checkers are a key component of formal verification, as it inspects each step of a proof, making sure it infers the next step. By starting inspection from the premises and then continuing onto each subsequent step until reaching the conclusion, an entire proof can be checked for soundness. Note that the claims of soundness are based on the assumption that the proof checker is itself is correct, thus making it part of the *trusted computing base* (TCB) [169], the set of all components, such as hardware and software, that must be trusted to be correct so that the claims of proof soundness, and hence, the properties that these proof demonstrate, will be true. Note that the other components of a proof assistant do not need to be trusted so long as the proofs that they generate are checked by the trusted proof checker. Since a proof checker cannot prove its own correctness, (the Gödel incompleteness theorem) these must always be trusted; however, their trustworthiness can be strengthened. There are several approaches to do this, such as exporting proofs so that other proof checkers can check the soundness as well [17], or verifying the proof checker in other proof assistants [4]. However, the key approach is to minimize the size and complexity of the proof checker. We briefly describe two important approaches: LCF-style and type-checking. Note that there is another approach, which is to carefully implement the proof assistant [20, 23, 88, 151], but this approach then requires that the whole proof assistant to be trusted.

The LCF-style approach, named after Logic of Computable Functions system (LCF) [63], uses a proof checker that is a small kernel consisting of a core set of axioms and inference rules, as well as an inference engine that applies the axioms and rules to construct new theorems. The soundness check of a proof is then the inference engine reconstructing the proof from the core set of axioms and inference rules. Given the small kernel size, it then becomes feasible to careful construct and check that the kernel is correct using only human inspection. Notable proof assistants that use the LCF-style approach are Isabelle/HOL [137], HOL4 [176], and HOL Light [72].

The type-checking approach uses a type checker as the trusted kernel [183]. In this approach, a proposition corresponds to a type in a programming language, a proof for the proposition corresponds to a program that satisfies that type, and a normalization or simplification of a proof corresponds to the evaluation of a program. Hence, the

proof checker is a type checker, which can be made to be quite small and simple, and hence, feasible for a human to check its correctness. Unlike the LCF-style approach, the type-checking approach produces portable proofs that can be independently reverified by external type checkers. However, this benefit entails a trade-off: the resulting proofs are generally much larger, leading to slower proof-checking performance relative to the smaller and more efficiently verified proofs produced by LCF-style systems. Notable proof assistants that use the type-checking approach include Rocq [25], Lean [129] and Agda [30]. While these proof assistants are based on the type-checking approach, they also incorporate elements of the LCF-style approach, such as a small trusted kernel and the ability to admit axioms into the trusted base.

2.2.5 Proof Automation

Many of proof assistant also include tools or have access to third-party tools that support features that make formal verification easier, in particular, proof automation [27, 153]. These tools can range from those that do not require much human interaction, such as SMT solvers [128], to those that do, such as interactive theorem provers [25, 137]. However, those that require more human interaction can typically prove more complex properties than those that require less human interaction, and are generally more trustworthy due to their smaller TCB (trusted computing base).

Highly automated verification tools also exist. These tools make verification easy because they only require a figurative push of a button to work. Tools in this category include model checkers [19, 189] and simulators [79, 104], SAT and SMT solvers [128] and first-order logic provers [181]. Model checkers and simulators, as mentioned earlier, are usually used to validate specifications, but these can also be used to "quick check" if a proposition is true by finding counterexamples. Boolean satisfiability problems can be solved using SAT solvers, with SMT (Satisfiability Modulo Theory) solvers improving on the proof capabilities by incorporating theories, such as arithmetic. First-order logic provers, as the name suggests, solve first-order logic propositions. The SAT and SMT solvers and first-order logic provers are typically used by other proof assistants [28] or code verifiers [37, 108, 109] to prove subgoals in larger proofs. Highly automated verification tools have large TCBs, as they are just like any other unverified program, meaning that the soundness of the proofs they produce is not guaranteed. However, proof assistants, like Isabelle/HOL, can still use the highly automated tools and ensure proof soundness without trusting them, by reconstructing the proofs within their own trusted ecosystem [27].

Other verification tools require users to provide hints or annotations to guide the verification. The most notable group of tools in this category are code verifiers. Code verifiers are designed to verify properties, such as correctness for programs in specific languages, usually imperative languages. The code is annotated with preconditions and postconditions, and is then verified using static analysis techniques and formal verification approaches, typically based on Hoare-style logics. Additional annotations may be required, such ghost state and code, assertions, and especially loop invariants and variants if the code contains loops. Since the proof generation is generally intricately combined with proof checking, these tools have large TCBs, requiring the whole tool to be trusted. Examples of these tools include VCC [37] and Frama-C [89] for C programs, Verifast [80] for Java and C programs, and SPARK [21] for Ada programs.

Interactive theorem provers [20, 23, 25, 30, 72, 129, 137, 176] require the most human interaction. These are able to verify systems with richer specifications, such as the seL4 microkernel [91], the Four Color Theorem [62] and the CompCert C compiler [110]. Moreover, many of these tools are more trustworthy, with only the mechanical proof checker component needing to be trusted. This is because all proofs, whether they are manually constructed by the user, automatically generated by tools that are part of the interactive theorem prover, or from an external third party, are checked by the mechanical proof checker component. Hence, if other components or external tools have bugs, the only issue that can arise is a failed proof attempt.

Due to the level of human participation required by interactive theorem provers, proof construction requires much more effort [154], and hence is more costly. To combat this, especially in the case of program verification, many interactive theorem provers connect with tools that automate some of the proof construction. This is especially useful for program verification, in particular for imperative programs, as one the major bottlenecks is handling the mismatch between the functional language of the theorem prover and the imperative language of the program. This mismatch results in imperative programs being deeply embedded, which naturally makes proving semantic properties, like correctness, difficult. Approaches used to combat this are code extraction, which follows a top-down approach of taking a verified algorithm embedding and translating it into an executable program; program abstraction or refinement, which takes a bottom-up approach that abstracts deep embeddings to shallower forms while maintaining refinement guarantees; and verified or certifying compilation, which is a mix of top-down and bottom-up approaches and uses a compiler that either has its compilation proved correct or produces a proof each time it compiles a program to demonstrate that the compilation was correct.

If an existing verified algorithm embedding exists, a top-down approach of translating this to a correct-by-construction executable program may be suitable. This approach,

called code extraction, is supported by many interactive theorem provers, such as Isabelle/HOL [70] and Rocq [11]. By using code extraction tools, executable programs for formally verified algorithms, with optimizations for improved efficiency, can be obtained by concretizing the mathematical types and definitions [99–101, 103]. Ideally, these tools should guarantee that the executable programs are correct with respect to the verified algorithms. However, many of these tools only provide these guarantees under the assumption that some of the translation phases are correct. There is ongoing work in this area to overcome this issue [3, 46, 59, 77, 130, 132, 156, 157].

If the goal is to verify an existing program, or the program needs to be tailored for a particular use case, a bottom-up approach may be more suitable. Tools that use this approach lift the deep embeddings of the programs to a more abstract embedding, usually a mixed shallow and deep embedding that is easier to verify. Importantly, these tools prove refinements between these embedding, thus guaranteeing that any semantic properties that hold for the abstract embedding also hold for the deep embedding of the program. An example of a tool that uses this method is AutoCorres [66, 67], which abstracts deep embeddings of C programs, which are obtained using Norrish'C-parser [187], to monadic shallow embeddings in Isabelle/HOL, whilst proving refinement between the two embeddings. Another example that can be used to verify the correctness of Haskell programs in Rocq is hs-to-coq [33].

Using a verified or certifying compiler [152] is another approach that can reduce the cost of program verification. The core idea is that programs encoded in the source language should be more abstract, and hence, easier to verify than the compiled program, which is encoded in a target language that is less abstract. With a verified or certifying compiler, the compiled program is guaranteed to be a refinement of the source program, meaning that any property proved about the source program is also true for the compiled program. Verified compilers are, as their names suggest, have their compilation phases formally verified, meaning that the translation from source language to target language is guaranteed to be correct for all valid programs. Examples of verified compilers are CompCert C [110] and CakeML [97]. Certifying compilers, on the other hand, use translation validation [158] to produce a proof each time they are run, with the proof demonstrating that the particular compilation is correct. Examples of certifying compilers are Cogent [148] and a compiler from C to binary [174]. Certifying compilers are not guaranteed to compile all syntactically legal programs, unlike verified compilers; however, they are typically easier to maintain since modifying the compiler does not necessitate a new proof, which verified compilers would require. Other compilers may use a mixed approach with some verified compiler phases and some phases using translation validation.

Most program verification focuses on verifying the correctness of a program that is written in a single language. However, large scale software developments consist of multiple programs written in different languages interoperating with each other. While it is possible to verify each individual program separately using the above methods, this does not necessarily mean that the composition of these programs also results in their correctness being composed. This is because different languages may have different static guarantees, and their interaction might render them invalid. Moreover, different languages may have different levels of abstraction and expressiveness, complicating matters even more. Some verified and certifying compilers support the composition of programs written in different languages, but with restrictions, such as only linking with languages with similar static guarantees and levels of abstraction and expressiveness [184], or only linking with languages whose semantics are a subset of the source language [83, 95, 136].

2.2.6 Summary and Relation to the Thesis

This section described how formal verification can be used to ensure the correctness of systems, such as mathematical theorems, algorithms or programs. Correctness is achieved through the use of rigorous formal proofs showing that the system satisfies its formal specification, a description of what the system should do, encoded in formal logic. The proofs are checked by a mechanical proof checker, which is a trusted program that ensures that each proof step can be inferred from the previous, and hence from the assumptions of the specification. While formal verification can be expensive due to its complexity, several tools exist that reduce this cost, especially for formal verification of programs, such as through code extraction and verified or certifying compilation.

Formal verification techniques have been successfully used to verify many algorithms and data structures [14, 138]. In particular, they have been used to verify key pattern preprocessing substring search algorithms [32, 57, 73, 100, 127, 155], a simple suffix array construction algorithm [29, 34, 52], and sorting algorithms that are components of many suffix array and BWT construction algorithms [24, 51, 68, 102, 177]. In addition to this, Bird and Mu [26] developed a Haskell implementation of the BWT and through the use of equational reasoning, albeit informally, implemented its inverse. However, no linear-time construction algorithms for the suffix array or the BWT has been formally verified. This is remedied by this thesis in its primary investigation on formally verified indexed pattern search algorithms.

Formal verification techniques have also been used to verify programs, with noteworthy examples being the seL4 microkernel [91] and the CompCert C compiler [110]. Crucial to these successes and many others are tools, such as abstraction tools [33, 66], code

extractors [59, 100], and verified and certifying compilers [97, 110, 148, 174] that reduce the cost of formal verification of programs. However, many of these tools limit the verification to programs written in a single language, as the composition of verified programs written in different languages does not necessarily mean that their verifications can also be composed. While some approaches do support the verification of programs with cross-language or multi-language components [83, 95, 136, 184], these are limited to languages that either have similar static guarantees and levels of expressiveness and abstraction, or where one language is a subset of another. What is missing is the case where a program is written in multiple languages that have different static guarantees and levels of abstraction and expressiveness. This is remedied by this thesis in its secondary investigation.

2.3 Thesis Direction

The primary investigation of this thesis is to formally verify the correctness of indexed pattern search algorithms in Isabelle/HOL [137]. The first component, which is presented in Chapter 3, contains the formal verification of the SA-IS algorithm [140], a linear-time suffix array construction algorithm. The second component, which is presented in Chapter 4, contains the formalization of the BWT [35] and its proof of invertibility. The last component, which is presented in Chapter 5, contains the verification two indexed pattern search algorithms: binary search [92, Chapter 6.2.1] that uses the suffix array as a text index, and backward search [54] that uses the BWT as a text index. By combining all three components, we obtain a complete pipeline of formally verified indexed pattern search algorithms, beginning with the formally verified text index construction, and ending with formally verified search algorithms.

The secondary investigation of this thesis addresses a related but orthogonal problem of reducing the formal verification costs for programs. Specifically for programs composed of parts written in different languages with different static guarantees and levels of expressiveness and abstraction. In this project, we frame our investigation around Cogent [148], a restricted, functional language, and C, a low-level, imperative language. Unlike much of the existing work, the composition between Cogent and C is a composition between languages with different levels of abstraction and static guarantees. Here, the focus is on demonstrating how the foreign function interface (FFI) of Cogent allows C types and functions to be used in Cogent programs without invalidating the static guarantees inherent to Cogent and also how the correctness of Cogent functions and C functions can be composed to obtain correctness of the whole Cogent-C system.

Below, we provide a brief overview of the key formal verification tools used in the two investigations. We begin with Isabelle/HOL which is the proof assistant that is used in both investigations. This is followed by Cogent, which is the core component of the secondary investigation. Finally, we end this chapter with a description of AutoCorres, which is a Hoare logic based tool that is used in the secondary investigation to verify properties about C programs.

2.3.1 Isabelle/HOL

Isabelle/HOL [137] is an interactive theorem prover for classical higher-order logic based on Church's simply-typed lambda calculus [36]. It is an LCF-style proof assistant [63], meaning that 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. Proof soundness is guaranteed so 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 [71, 82], 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 [123], OCaml [124], Haskell [81] or Scala [149] programs via Isabelle/HOL's code extraction facilities [70]. 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.

2.3.2 Cogent

Cogent [147] is a restricted purely functional language that is designed to simplify the development of verified operating system components. It has a certifying compiler [147, 165] that takes a Cogent program and outputs a shallow embedding of the program in Isabelle/HOL [137], a C program, and a formal refinement proof (also in Isabelle/HOL) that connects the shallow embedding with the C program. Cogent's main restrictions are the purposeful lack of recursion or loops, which ensures totality, and its *uniqueness type system*. Cogent's uniqueness type system ensures that uniquely typed objects only have one mutable reference that must be used exactly once, or multiple read-only references, resulting in several benefits, such as destructive updates and guarantees memory safety.

Given that the restricted target domains of Cogent still contain some amounts of iteration and other features prohibited by Cogent, such as iteration over data structures like buffers, Cogent supports a principled foreign function interface (FFI). This allows parts of a program to be implemented in C without invalidating the static guarantees provided by Cogent. To use the principled FFI, engineers provide data structures and their associated operations, in a special dialect of C, and import them into Cogent, including formal reasoning. This special C code, called *template C*, can refer to Cogent data types and functions, and is translated into standard C along with the Cogent program by the Cogent compiler. As long as the C components respect Cogent's foreign function interface — that is, are correct, memory-safe and respect the uniqueness invariant — the Cogent framework guarantees that correctness properties proved on high-level specs also apply to the compiler output.

2.3.3 AutoCorres

AutoCorres [66, 67] is a tool that automatically abstracts deep embeddings of C programs to monadic shallow embeddings in Isabelle/HOL. Crucially, AutoCorres generates proofs that certify that the monadic shallow embeddings are refined by the deep embeddings. The deep embeddings of the C programs are obtained using Norrish's C parser [187], which translates the C programs to SIMPL programs [170, 171], a sequential imperative programming language model in Isabelle/HOL. Key constructs of imperative languages, such as loops, branching, assignments, sequential operations, non-determinism and exceptions, are deeply embedded, while simple computations, such as arithmetic, are shallowly embedded. AutoCorres maps the key constructs to equivalent expressions in Isabelle/HOL, such as sequential operations to the monadic bind operation.

The monadic shallow embedding is more amenable to formal verification as C types are abstracted to Isabelle/HOL types, local variables to bound variables, and the heap is

mapped to a collection of heaps, one for each type, and each heap has a set of valid pointers associated with it [179], where a pointer is valid for a given heap if it points to an object in that heap. Properties about the monadic shallow embedding, and hence the associated C code deep embedding, are proved using Hoare and separation logic. AutoCorres offers optional abstractions that make the monadic shallow embedding more abstract, such as lifting finite words to natural numbers.

Chapter 3

Formally Verified Suffix Array Construction

In the previous chapter, we provided a broad overview of the substring search problem, and the different ways to solve it. One approach that is especially well-suited to solve the substring search problem is indexed pattern search using the suffix arrays [117]. 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. Such algorithms, however, are conceptually complex, and implementing these algorithms require a deep understanding of their underlying theory.

In this chapter[1], we begin our investigation on formally verified indexed pattern search by verifying a sophisticated suffix array construction algorithm called the SA-IS algorithm [140]. We choose the SA-IS algorithm because it is both asymptotically fast, with a worst-case time complexity of $\mathcal{O}(n)$, and also fast in practice, especially on highly repetitive texts [18]. Moreover, the algorithm shares several features with other suffix array construction algorithms [58, 64, 94, 113, 113, 139], which enhances its appeal from a reusability standpoint. To verify the SA-IS algorithm, we use formal verification techniques, also described in the previous chapter, to develop an implementation of the algorithm in Isabelle/HOL [137] and formally verify that it is equivalent to a mathematical functional correctness specification of suffix arrays, a task requiring the formalization

[1]This chapter is based on the paper: L. Cheung, A. Moffat, and C. Rizkallah. Formally Verified Suffix Array Construction. In *Journal of Automated Reasoning*, 69(3), no. pages 21, 2025. doi: 10.1007/s10817-025-09735-8.

of a wide range of underlying properties of strings and suffixes[2]. We also use Isabelle's code extraction facilities to extract our Isabelle/HOL implementation of the SA-IS algorithm to an executable Haskell implementation of SA-IS, which albeit is inefficient due to using lists and natural numbers rather than arrays and machine words, to demonstrate that our verified Isabelle/HOL implementation of SA-IS can be refined to an executable implementation in its current form.

3.1 Introduction

The *suffix array* [117] is an indexing data structure that stores all 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 [69, Chapter 5]. 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 text, for texts drawn from constant-sized alphabets [84, 94, 140, 141].

The suffix array is widely used in substring matching problems because a pattern of length m can be found within a text of length n by searching for all suffixes of the text that begin with the pattern. The streamlined structure of the suffix arrays simplifies this task, as all suffixes that begin with the same prefix are all located in the suffix array in a single contiguous block. Finding such a block can be achieved in $\mathcal{O}(m \log n)$ time, using simple search methods such as binary search [92, Chapter 6]. With additional data structures, this can be reduced to $\mathcal{O}(m)$ time [1]. In addition to substring search, the suffix array is also used to construct the *Burrows-Wheeler transform* (BWT) [35], as the BWT can be constructed from the suffix array in linear-time. Since the suffix array can be constructed in linear-time, the overall time cost for constructing the BWT is also linear. The BWT is an invertible data transformation that is widely used in data compression, such as `bzip2` [173], and in compressed text search [55], most notably in bioinformatics where it is used for DNA analysis [105, 106, 111, 112].

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 comparison-based sorting, an example of which is shown in Figure 3.1. This straightforward approach is only suitable for moderate-length texts and for texts that do not contain long subsequence repetitions, as it has a worst-case time complexity

[2]The proofs are published in: L. Cheung and C. Rizkallah. Formally Verified Suffix Array Construction. *Archive of Formal Proofs*, September 2024. ISSN 2150-914x. https://isa-afp.org/entries/SuffixArray.html. Formal proof development.

of $\mathcal{O}(n^2 \log n)$ for texts of length n. Such running times become infeasible once n reaches approximately 10^6 characters. Important texts, such as genomes, do not satisfy these requirements, and require more efficient suffix array construction algorithms.

In this chapter, we present a formal verification of the *suffix array construction by induced sorting* (SA-IS) algorithm [140, 141] in Isabelle/HOL [137]. 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 chapter 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 this chapter delivers on those goals. Section 3.2 provides definitions and the necessary background material, and lays the foundation for the new work. Section 3.3 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 3.4 to 3.6, via a sequence of definitions, lemmas, and theorems. Section 3.7 discusses the insights gained during the construction of the proofs. Finally, Section 3.8 summarizes this part of our presentation.

3.2 Problem Description

This section first introduces key concepts relevant to suffix array construction in Section 3.2.1. Building on those ideas, we then give a description of suffix arrays and their construction in Section 3.2.2, with in-depth treatment available in the work of Crochemore et al. [41, Chapter 4] if desired.

```
1: function simple(S)
2:     suffixes ← map (λi. S[i...]) [0...< |S|]
3:     suffixes ← sort_{<lex} suffixes
4:     SA ← map (suffix-id S) suffixes
5:     return SA
6: end function
```

FIGURE 3.1: Pseudocode for a simple suffix array construction algorithm, with S the input sequence and SA the computed suffix array.

3.2.1 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 3.1 (Lexicographical Order, $<_{lex}$)**.**

$$[] <_{lex} (y \# ys) \quad := \mathsf{true} \tag{3.1a}$$

$$(x \# xs) <_{lex} [] \quad := \mathsf{false} \tag{3.1b}$$

$$[] <_{lex} [] \quad := \mathsf{false} \tag{3.1c}$$

$$(x \# xs) <_{lex} (y \# ys) \quad := (x \leq y) \wedge (x = y \longrightarrow xs <_{lex} ys)\,. \tag{3.1d}$$

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 $S = \mathtt{aabdabb}$, then $S[2\ldots] = \mathtt{bdabb}$; and $S[6\ldots] = \mathtt{b}$. 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*, denoted by `$`, that marks the end of the sequence and 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 3.2 (Valid Sequence)**.**

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

where @ concatenates two sequences, and set turns a list into a set.

While this domain restriction might appear 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.

3.2.2 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 $0 \leq i < j < |S|$, we have $S[SA[i] \ldots] <_{lex} S[SA[j] \ldots]$. Formally, this axiomatic characterization of a suffix array is defined as follows:

Definition 3.3 (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

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

where $xs \sim ys$ means that xs and ys are permutations of each other and $\textbf{sorted}_{<_{lex}}$ is a predicate that checks if a list is sorted with respect to 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 $SA = [0, 4, 1, 6, 5, 2, 3]$, and this corresponds to the ordered suffixes of S, which are [aabdabb, abb, abdabb, b, bb, bdabb, 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 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 an instance 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 3.1, 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, which is needed by a comparison-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. [160] provide an extensive survey of suffix array construction algorithms, categorizing each according to its strategy. The SA-IS algorithm [140, 141] that is described in Section 3.3 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 [58, 78, 94]. Details of the SA-IS algorithm, the main focus of this chapter, are provided in the next section.

3.3 Calculating Suffix Arrays

The SA-IS algorithm [140, 141] 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 the suffix properties that the SA-IS algorithm relies on and then describe the algorithm itself.

3.3.1 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 3.4 (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 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 of Definition 3.4 means that $S[7 \ldots] = \$$ is an S-type. Figure 3.2 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 3.4, with the following being one of the most vital:

Theorem 3.5 (Lemma 2 from Ko and Aluru [94])**.** *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, $S[i \ldots] <_{lex} S[j \ldots]$.*

The intuition behind this that when comparing an L-type suffix with an S-type, which starts with the same symbol, there will be a shared prefix followed by a symbol where the mismatch occurs. For the L-type suffix, the mismatching symbol must be strictly smaller than the preceding symbol as this is what made the suffix an L-type. For the S-type suffix, the mismatching symbol must be strictly greater than the preceding symbol as this is what made the suffix an S-type. Since both of the preceding symbols are equal, as it was part of the shared prefix, the mismatching symbol of the L-type must be strictly

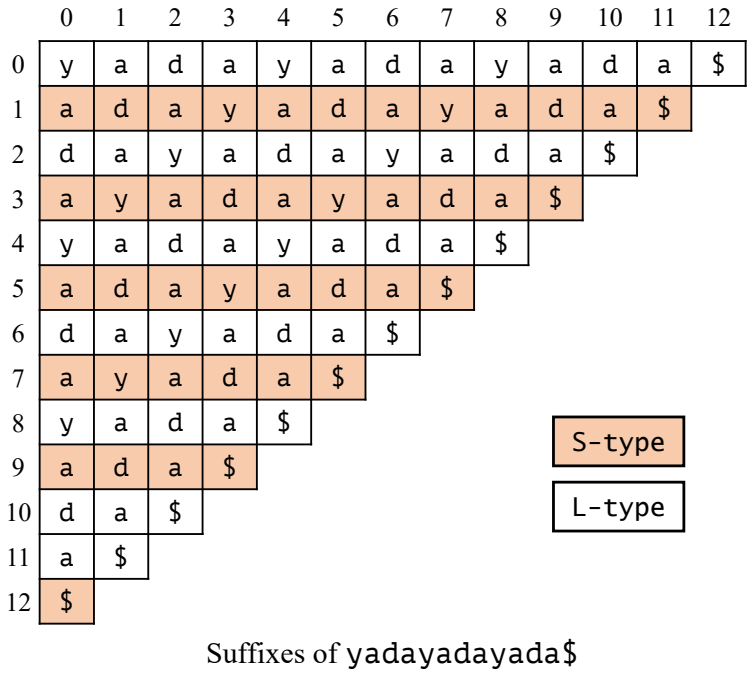


FIGURE 3.2: The suffix types of the sequence S = `yadayadayada$`. Shaded values correspond to S-type suffixes, each of which is lexicographically less than the suffix that occurs immediately after it. Unshaded values are L-type suffixes, each of which is lexicographically greater than the suffix that occurs immediately after it.

less than the mismatching symbol of the S-type, and hence the L-type suffix is less than the S-type suffix.

Theorem 3.5 means that it is possible to partition suffixes in a suffix array both by their starting symbol, similar to a single pass of bucket sort, and by their corresponding suffix type; with all the L-type suffixes being at the beginning of each possible start symbol's partition, and all the S-type suffixes being at the end of that same partition. For example, consider the six suffixes in Figure 3.2 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 3.5, it immediately follows that the suffix at position 11 is less than the other suffixes that start with `a` with respect to $<_{lex}$. 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 backwards to the beginning, applying these mutually exclusive rules:

Definition 3.6 (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 3.1 and 3.4, and Rule 4 follows from the following:

Theorem 3.7 (Lemma 1 from Ko and Aluru [94])**.** *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 [94]. 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 3.8 (Definition 2.1 from Nong et al. [140])**.** For a valid sequence S and $i > 0$, the suffix $S[i\ldots]$ is an LMS-type (leftmost S-type) iff $S[i\ldots]$ is an S-type and $S[i-1\ldots]$ is an L-type.

Using the earlier example, where $S =$ `yadayadayada$`, we know that $S[9\ldots] =$ `ada$` is an LMS-type suffix, since $S[9\ldots]$ is an S-type suffix and $S[8\ldots] =$ `yada$` is an L-type suffix. In fact, all S-type suffixes of this particular S are LMS-type suffixes.

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" (details provided shortly), 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 3.9 (LMS-prefix)**.** For a valid sequence S and $0 \leq i < |S|$, let $i \leq j < |S|$ be the next LMS-type suffix location in S at position i or beyond, or $|S|-1$ if no such LMS-type suffix occurs. Then the subsequence $S[i\ldots<j]$ is the i^{th} LMS-prefix.

Definition 3.10 (LMS-substring, Definition 2.2 from Nong et al. [140])**.** For a valid sequence S and $0 \leq i < |S|$, let $i < j < |S|$ be the next LMS-type suffix location in S strictly beyond i, or $|S|-1$ if no such LMS-type suffix 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 3.2, 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 3.11 (Substring Ordering, Definition 2.3 from Nong et al. [140])**.** Let S be a valid sequence and suppose that the comparison is either 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 with respect 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.

3.3.2 The SA-IS Algorithm

The SA-IS algorithm is presented in Figure 3.3, adapted from Nong et al. [140, 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.

3.3.2.1 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:  function SA-IS(S)
2:      ST : [0 ... < n] of {L-type, S-type}
3:      B : [0 ... < |Σ|] of natural numbers (ℕ)
4:      P : [0 ... < n'] of ℕ
5:      S' : [0 ... < n'] of ℕ
6:      SA' : [0 ... < n'] of ℕ
7:      SA: [0 ... < n] of ℕ
8:      B ← compute the buckets for S for each symbol in Σ
9:      ST ← compute the suffix type for each of the suffixes of S
10:     n' ← count how many LMS-type suffixes are in ST
11:     P ← extract all the locations of the LMS-type suffixes from ST
12:     SA ← induced sort the n' LMS-substrings using P, ST and B
13:     S' ← scan SA, naming each LMS-substring of an LMS-type suffix by its rank
14:     if each symbol in S' is unique then
15:         SA' ← set sorted LMS-types obtained by scanning SA
16:     else
17:         SA' ← SA-IS(S')
18:         SA' ← map each suffix S' back to suffixes in S, keeping the order of SA'
19:     end if
20:     SA ← induced sort suffixes from SA' using ST and B
21:     return SA
22: end function
```

FIGURE 3.3: The SA-IS algorithm, adapted from Nong et al. [140, 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.

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 3.11.

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, denoted by S'_1.

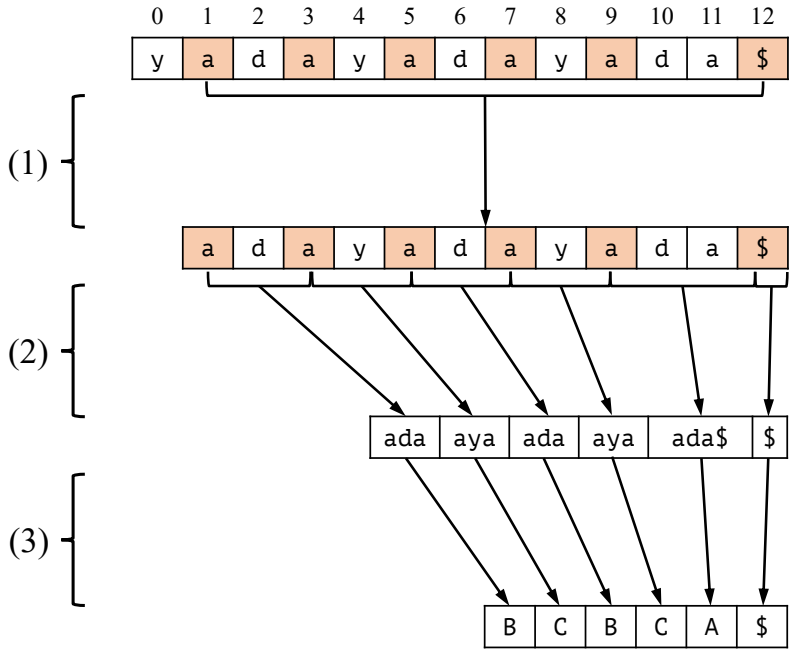


FIGURE 3.4: The sequence $S =$ `yadayadayada$` is reduced to a shorter surrogate $S' =$ `BCBCA$` by: (1) taking the first LMS-type suffix $S_1 =$ `adayadayada$`; (2) mapping it to a sequence of LMS-substrings of LMS-type suffixes; and (3) mapping these LMS-substrings to their symbols in a new alphabet $\Sigma' = \{\texttt{\$}, \texttt{A}, \texttt{B}, \texttt{C}\}$, where `$` $\mapsto$ `$`, `ada$` $\mapsto$ `A`, `ada` $\mapsto$ `B` and `aya` $\mapsto$ `C`. Locations of the LMS-type suffixes are shaded.

3. Map each of the LMS-substrings in S'_1 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 possible 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 3.4 illustrates the reduction process applied to the sequence $S =$ `yadayadayada$` shown earlier in Figure 3.2. The distinct LMS-substrings in sorted order are `$`, `ada$`, `ada$` and `aya`, and each is mapped to a symbol in a new alphabet Σ'. In the example, the mapping is `$` $\mapsto$ `$`, `ada$` $\mapsto$ `A`, `ada` $\mapsto$ `B` and `aya` $\mapsto$ `C` respectively; with $\Sigma' = \{\texttt{\$}, \texttt{A}, \texttt{B}, \texttt{C}\}$. Once Σ' has 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.

3.3.2.2 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 the LMS-substrings of the sequence in connection with the ordering in Definition 3.11 to produce the approximate suffix array, which is used in the sequence reduction.

The induced sorting procedure can be in turn divided into three functions. The first function, denoted by insert_lms, inserts the LMS-type suffixes into the array in sorted order, initializing the order the suffixes will be sorted with respect to. The second function, denoted by induce_l, induced sorts the L-types based on the order of the LMS-types. The third function, 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 functions.

All three functions 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 3.5, 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 3.11. 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

```
1:  function induce_l(S, ST, B, SA)
2:      i ← 0
3:      while i < |S| do
4:          j ← SA[i] − 1
5:          if ST[j] is an L-type then
6:              SA[B[S[j]]] ← j
7:              B[S[j]] ← B[S[j]] + 1
8:          end if
9:          i ← i + 1
10:     end while
11:     return SA
12: end function
```

FIGURE 3.5: The induced sorting function for the L-types, where S is 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 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.

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 of insert_lms and induce_s. The B array also indicates where the next free location in the suffix array is for each bucket, and this is used to determine where the next insert should occur. The B array is updated accordingly by the induced sorting functions.

The first function 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 (with respect to lexicographical order). When induced sorting LMS-substrings, insert_lms inserts LMS-prefixes of LMS-type suffixes into the approximate suffix array in sorted order (with respect to Definition 3.11). 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 that suffix, thus meaning that the LMS-prefixes starting with the same symbol in the input array will always be listed in sorted order.

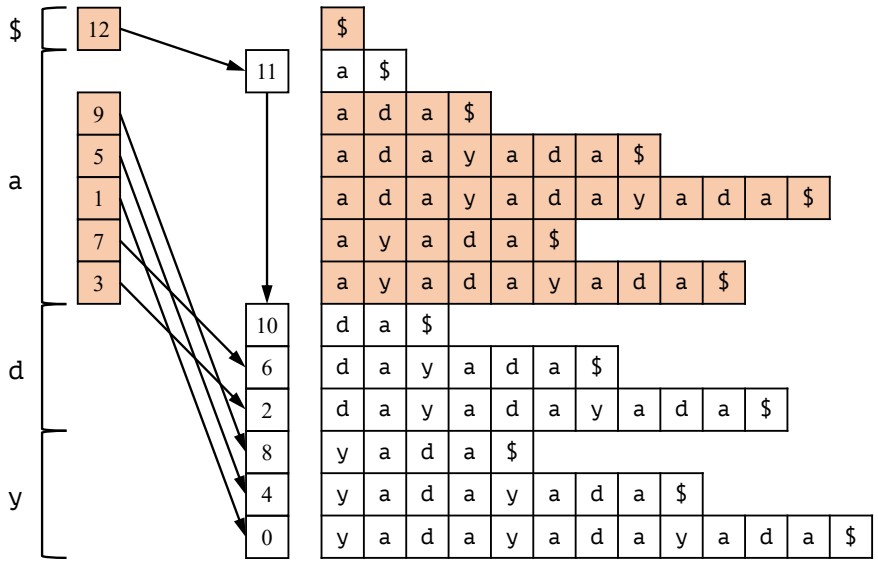


FIGURE 3.6: Induced sorting the L-type suffixes from the LMS-type suffixes of the sequence S = `yadayadayada$`. The procedure occurs from top to bottom, starting from $S[12\ldots]$, then progressing to $S[11\ldots]$, $S[9\ldots]$, and so on, finishing with $S[0\ldots]$. The LMS-type suffixes are shaded, the arrows indicate where an L-type suffix is inserted and from which suffix this was from, and the labeled brackets on the left signify the buckets, managed by the array B.

3.3.2.3 Induced Sorting — L-types

The second function 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 function, shown in Figure 3.5, 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 $c \in \Sigma$, $B[c] = \|\{k \mid k < |S| \wedge S[k] < c\}\|$. After an L-type is inserted into a bucket, the pointer corresponding to that bucket is incremented (line 7), pointing to the next free position.

Intuitively, the function 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 3.11 have the property that the sequence $x \# xs$ is less than the sequence $x \# ys$ iff xs is less than ys.

```
1:  function induce_s(S, ST, B, SA)
2:      i ← |S| − 1
3:      while i > 0 do
4:          j ← SA[i] − 1
5:          if ST[j] is an S-type then
6:              SA[B[S[j] − 1]] ← j
7:              B[S[j]] ← B[S[j]] − 1
8:          end if
9:          i ← i − 1
10:     end while
11:     return SA
12: end function
```

FIGURE 3.7: The induced sorting function for the 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.

An example of the induced sorting for the L-type suffixes is shown in Figure 3.6 for the sequence $S =$ `yadayadayada$`. On the left are the LMS-type suffixes that have been inserted into their buckets in lexicographical order, as a result of the previous function for inserting LMS-type suffixes. To the right of these are the L-type suffixes that are inserted 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 happens because $S[11\ldots] =$ `a$` is both lexicographically smaller and is encountered before $S[7\ldots] =$ `ayada$`.

3.3.2.4 Induced Sorting — S-types

The third function induce_s, takes the output of induce_l and inserts all the S-types into the suffix 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 function has finished executing. If the L-type suffixes in the output of induce_l are lexicographically sorted, the all the S-type suffixes of the sequence will be in sorted order, after the induce_s finishes executing, and hence the output of induce_s will be the suffix array.

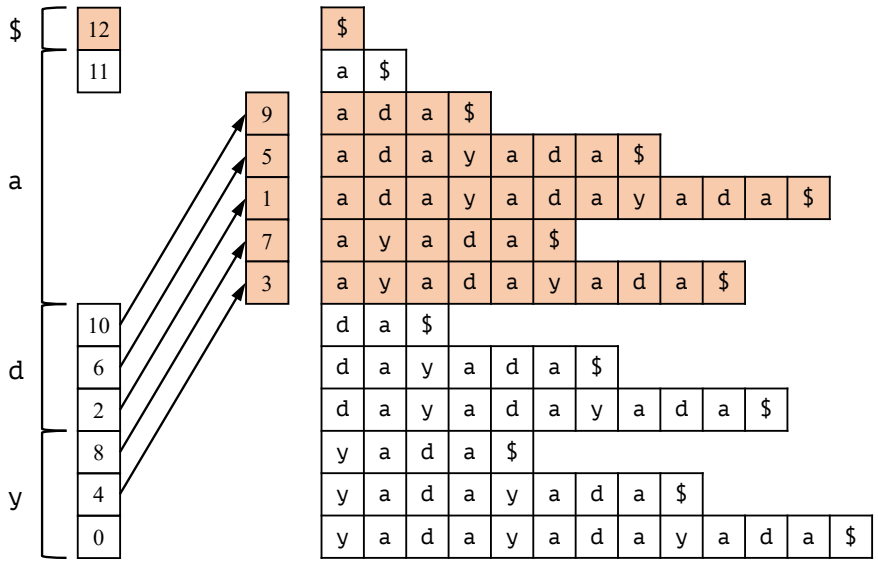


FIGURE 3.8: Induced sorting the S-type suffixes from the L-type suffixes of the sequence S = yadayadayada$. The procedure proceeds from bottom to top, starting from $S[0\ldots]$, then to $S[4\ldots]$, $S[8\ldots]$, and so on, finishing with $S[12\ldots]$. The LMS-type suffixes are shaded, the arrows indicate where an S-type suffix is inserted and which suffix it came from, and the labeled brackets on the side scope the buckets.

The induce_s function, shown in Figure 3.7, is very similar to the induce_l function. However, it instead iterates over the array backwards 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 function, they are initialized to the end of each 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 (line 7) rather than incremented (as was the case in Figure 3.5).

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 function. It is the LMS-type insertion function that inserts this into its bucket, and the induce_l and the induce_s functions do not modify its placement.

Figure 3.8 demonstrates the induced sorting function 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 function for L-types. The array is scanned from the largest to smallest lexicographically, from bottom to top in Figure 3.8, 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

$S[8\ldots] = \texttt{yada\$}$, and so $S[7\ldots] = \texttt{ayada\$}$ is inserted. This continues until the start of the array is reached. As already noted, the placement of the last suffix $S[12\ldots] = \texttt{\$}$ is neither modified by the induced sorting of the L-types nor by the induced sorting of the S-types. This is indicated by being on the left in both Figures 3.6 and 3.8, where being on the left signifies not being inserted by the function. Its position is always at the start of the suffix array.

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

3.3.3 Complexity Analysis

Given a sequence S of length n, whose elements are drawn from a constant-sized alphabet Σ, the SA-IS algorithm runs in linear-time and linear-space. This is because the length of the reduced sequence S' is always guaranteed to be at most $\lfloor n/2 \rfloor$ elements only, and the suffix type computation, induced sorting and the reduced sequence construction all run in linear-time and linear-space. Note that the suffix types computation clearly runs in linear time and space, as it can be computed with a single backwards scan of S using Definition 3.6 at each iteration, and the suffix types are stored in an array of size n.

The maximum length of S' is $\lfloor n/2 \rfloor$ because each element of S' corresponds to an LMS-substring of an LMS-type suffix of S. From the definition of an LMS-type suffix (Definition 3.8), an LMS-type suffix is an S-type suffix that is immediately preceded by an L-type suffix. Since L-type and S-type suffixes are mutually exclusive types, the maximum number of LMS-type suffixes only occurs when the suffixes of S have alternating L-types and S-types, with the last suffix always being an S-type suffix due it being the sequence only containing the sentinel \$. This results in there being $\lceil n/2 \rceil$ S-type suffixes, where the rounding up is due to the case where n is an odd number and the first suffix is an S-type. However, the first suffix of S can never be an LMS-type as no suffix occurs before it, hence the maximum number of LMS-type suffixes is $\lfloor n/2 \rfloor$.

The induced sorting runs in linear time and space over the length of the sequence S because each of the induced sorting functions and the bucket pointer array construction, which is constructed by computing the frequencies of each symbol in S and then calculating the cumulative sums of the frequencies, are all linear-time operations for constant-sized alphabets. The insert_lms function only iterates over all the LMS-type suffixes of which there are only $\lfloor n/2 \rfloor$ many. Both induce_l and induce_s functions only iterate over the suffix array once each. Hence, the induced sorting functions run in $\Theta(n)$

time. Similarly, the frequency computation only iterates over S, and hence runs in $\Theta(n)$. The cumulative sum operation, however, iterates over Σ, which means that it runs in $\Theta(\|\Sigma\|)$. However, Σ becomes Σ' at each recursive call to the SA-IS algorithm, and this is bounded by $\lfloor n/2 \rfloor$. This is because each symbol in Σ' corresponds to a distinct LMS-substring of a LMS-type suffix of S of which there can be at most $\lfloor n/2 \rfloor$ many. Hence, the bucket pointer construction in the first call of the SA-IS algorithm takes $\Theta(n + \|\Sigma\|)$ and every subsequent construction in the recursion takes $\Theta(n)$ time. Given that $\|\Sigma\|$ is a constant, the construction then takes $\Theta(n)$. Thus, the time complexity is $\Theta(n)$ for the entire induced sorting. The space required for the entire induced sorting is the space needed to store the suffix array, the bucket pointer array and the suffix types array, which is $\Theta(n + \|\Sigma\|)$, however, since $\|\Sigma\|$ is a constant for the first call and is bounded by n in every subsequent call, the space complexity is $\Theta(n)$.

In the reduced sequence construction, it compares LMS-substrings of LMS-type suffixes with each other. Each of the LMS-substrings is compared at most twice, resulting in a maximum of $2k$ sequence comparisons, where k is the number of LMS-substrings of LMS-type suffixes. However, the sum of the lengths of these LMS-substrings is at most $n + k$, as adjacent LMS-substrings of LMS-type suffixes can only overlap with each other on their first and last elements. That is, the first and last elements of the i^{th} LMS-substring of an LMS-type suffix overlaps with the last and first elements of the $i - 1^{\text{th}}$ and $i + 1^{\text{th}}$ LMS-substrings of LMS-type suffixes, respectively. Hence, the $2k$ sequence comparisons in reality is just two scans of $n + k$ symbols. Since the maximum value of k is $\lfloor n/2 \rfloor$. The time it takes is $\Theta(n)$. The space required to construct the reduced sequence is clearly an array of size $\lfloor n/2 \rfloor$, and the space to store the mapping from an LMS-substring to a new symbol in the reduced alphabet Σ' is at most n, as this would be an array that is used to map LMS-substring locations to alphabet symbols, where the array indices are treated as LMS-substring locations. Thus, the space complexity for the reduced sequence construction is $\Theta(n)$.

Given that the reduced sequence $|S'|$ is at most $\lfloor n/2 \rfloor$, and the suffix type computation, reduced sequence construction and induced sorting all run in linear-time and linear-space, 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 constant-sized alphabets.

3.3.4 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 [58, 64, 78, 94, 113, 139].

This section has described the SA-IS algorithm and provided intuitive explanations for its various functions. These informal descriptions have been primarily based on rhetoric rather 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 remainder of this chapter.

3.4 An Outline of the Formal Development

This section provides an overview of our formalization and total correctness proof of the SA-IS algorithm in the Isabelle/HOL theorem prover. Later sections then provide a comprehensive proof of correctness, and are followed by a summary that lists key insights and challenges.

3.4.1 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 define given that index and suffix locations can be modeled as natural numbers (starting from 0) 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 3.2 and 3.3 (on pages 45 and 46 respectively),

```
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 )"

```

FIGURE 3.9: The Isabelle/HOL embedding of the SA-IS algorithm.

which are just transliterations of these definitions, to define the soundness of suffix array construction as:

Definition 3.12 (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

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

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 3.12 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 3.9, is largely a transliteration of the algorithm description, shown in Figure 3.3. 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 3.9, that are used to store the locations of LMS-type suffixes

```
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"
```

FIGURE 3.10: 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.

```
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))"
```

FIGURE 3.11: The step function that implements one iteration of the induce_l function, that is, the loop body of Figure 3.5.

after they have been sorted with respect to the substring ordering and with respect to 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, which is explained in more detail in Chapter 6. The loop combinator repeat, shown in Figure 3.10, simply repeatedly applies a function f n many times to an accumulator *acc* with an observer argument *obsv*. 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 3.11.

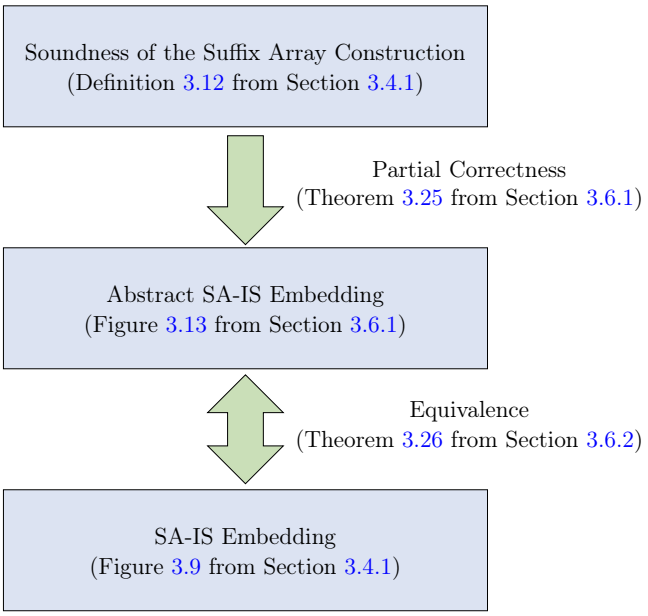


FIGURE 3.12: A visualization of the approach taken to prove soundness with respect to the formal specification (Definition 3.12 from Section 3.4.1) for the SA-IS algorithm. Boxes represent specifications and embeddings and arrows represent proofs, with double-sided arrows meaning equivalence.

3.4.2 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 3.5 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 3.3.1.

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 respect to the formal specification. These are presented in more detail in Section 3.6. 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 3.12. The first step is the definition and proof of soundness with respect to Definition 3.12 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 sais_α, to simplify the verification. The embeddings simplify the verification because it forgoes the efficient suffix type

pre-computation (Definition 3.6) 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 sais_α for valid lists. The proof is not particularly complex, relying mostly on induction on the arguments of sais/sais_α, term rewriting, and on the correctness of the suffix type pre-computation (Definition 3.6), 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 3.3.1, and hence, was largely an exercise in translating these arguments into Isabelle/HOL syntax.

As mentioned in Section 3.2.1, algorithms, such as SA-IS, assume that the input sequence is valid. We demonstrate in Section 3.6.5 that the validity assumption can be removed by providing a lifting function that given any alphabet and sequence 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.

3.4.3 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 chapter, which is to verify the correctness of the SA-IS algorithm.

3.5 Formalizing Underlying Properties

To prove the correctness of the SA-IS algorithm, the underlying theory about sequences and suffixes from Sections 3.2.1 and 3.3.1, 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 3.2 (page 45).

Formalizing suffix properties, however, is more challenging. This is because both the suffix type and the subsequence comparison functions (Definitions 3.4 and 3.11 on pages 48 and 51, respectively), as defined by Nong et al. [140], 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 3.11) 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 3.11. 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 3.11 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 3.11 is a ternary operator is to replace the lexicographical comparison in the suffix type definition and the subsequence comparison (Definition 3.11) with the following comparison function.

Definition 3.13 (Alternative Lexicographical Order)**.**

$$[] <_{lex'} (y \# ys) \quad := \mathsf{false} \tag{3.2a}$$

$$(x \# xs) <_{lex'} [] \quad := \mathsf{true} \tag{3.2b}$$

$$[] <_{lex'} [] \quad := \mathsf{false} \tag{3.2c}$$

$$(x \# xs) <_{lex'} (y \# ys) \quad := (x \leq y) \wedge (x = y \longrightarrow xs <_{lex'} ys)\,. \tag{3.2d}$$

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 3.11).

Definition 3.13 ($<_{lex'}$) is somewhat similar to 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 `aaa` being greater than `aaaaa`. Interestingly, both lexicographical comparison functions produce the same results 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 3.4, and so the suffix type can be defined using $<_{lex'}$, removing the valid list requirement and the special case for $S = \$$.

Definition 3.14 (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. This is because 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, including Theorems 3.5 and 3.7. 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 function (Definition 3.11) is defined for, that is, returns a true or false result, $<_{lex'}$ is equivalent to it, and hence the subsequence comparison (Definition 3.11) 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 3.5 and Theorem 3.7, eliminates the validity requirements. Furthermore, for Theorem 3.5, the two lists need not be suffixes of the same list. The generalized theorem of Theorem 3.5 and an outline of its proof is shown below.

Theorem 3.15 (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 $S = as \mathbin{@} xs \mathbin{@} b \# bs$ and $T = as \mathbin{@} ys \mathbin{@} c \# cs$, 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. As $a > b$ and $a < c$, then $b < c$, and so $S <_{lex'} T$.
- 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 3.16 (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 3.15 and 3.16, it follows that LMS-substrings of L-type suffixes are less than LMS-substrings of S-type suffixes with respect to $<_{lex'}$, for LMS-substrings starting with the same symbol, as LMS-substrings have the same list type as their corresponding suffix (Theorem 3.16), and L-type lists are less than S-type lists for lists starting with the same symbol (Theorem 3.15) with respect to $<_{lex'}$. This result is exactly what Definition 3.11 wished to enforce, and hence can be replaced with $<_{lex'}$.

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

3.6 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 3.4.2, with Section 3.6.1 describing the soundness proof of the abstract embedding sais_α; Section 3.6.2 describing the equivalence proof between the sais_α and the embedding of the SA-IS algorithm, sais; Section 3.6.3 describing

```
function sais_α :: "nat list ⇒ nat list"
  where
"sais_α [] = []" |
"sais_α [x] = [0]" |
"sais_α (a # b # xs) =
  (let
     S = a # b # xs;
     –< Scan and extract LMS-type >
     P = extract_lms_α S [0 ..< length S];
     –< Induced sort the LMS-substrings >
     SA = induce_α id S P ;
     –< Extract and name the sorted LMS-substrings of LMS-type suffixes >
     Q = extract_lms_α S SA ;
     names = rename_mapping_α 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 (sais_α S'))
   –< Induced sort the suffixes >
   in induce_α id S R )"
```

FIGURE 3.13: The abstract Isabelle/HOL embedding of the SA-IS algorithm.

the termination proof; and Section 3.6.4 describing the composition of these proofs to obtain total correctness of the SA-IS algorithm. Finally, this section is completed by Section 3.6.5, which is a small extension to generalize the SA-IS algorithm to non-valid sequences.

3.6.1 Formally Verifying the SA-IS Abstract Embedding

The abstract embedding sais_α, shown in Figure 3.13, is similar to the concrete embedding shown in Figure 3.9. However, the suffix types are no longer pre-computed and instead, suffix type queries, which occur in the functions $\mathsf{extract\_lms}_\alpha$, induce_α and $\mathsf{rename\_mapping}_\alpha$, are computed using the definition whenever required. Moreover, additional array indexing checks are added to induce_α and its functions, as these have intricate array updates. All of these changes make the verification easier by shrinking theorem statements and pruning cases.

The soundness of sais_α is proved in two steps. First, sais_α is shown to produce a permutation of the suffix locations, that is:

$$\textbf{valid}\, S \longrightarrow (\mathsf{sais}_\alpha\, S) \sim [0 \mathinner{..<} |S|]\,. \tag{3.3}$$

Then, using this fact, we show that the suffix locations in the permutation are ordered according to the lexicographical ordering of the suffixes,

$$\textbf{valid}\, S \longrightarrow \textbf{sorted}_{<_{lex}}\, (\mathsf{map}\, (\lambda i.\, S[i \ldots])\, (\mathsf{sais}_\alpha\, S))\,. \tag{3.4}$$

The proofs of Equations (3.3) and (3.4) depend on the correctness of each function of sais_α, which are summarized as follows:

- The $\mathsf{extract\_lms}_\alpha$ 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 induce_α 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 produces a list of suffix locations that is a permutation of $[0 \ldots < |S|]$ and these are ordered according to the LMS-substring ordering with respect to $\leq_{lex'}$. Moreover, if the LMS-type suffix locations it is given are ordered according to suffix ordering with respect to $\leq_{lex'}$, then the list of suffix locations will also have the same ordering. Note that induce_α is just the sequential application of the functions $\mathsf{insert\_lms}_\alpha$, which initially inserts the LMS-types at locations in the list according to their starting symbol, $\mathsf{induce\_l}_\alpha$, which induced sorts the L-types based on the order that the LMS-types, and $\mathsf{induce\_s}_\alpha$, which induced sorts the S-types based on the order of the L-types.
- The $\mathsf{rename\_mapping}_\alpha$ function is correct if it produces the mapping from LMS-type suffix locations to the rank of its corresponding 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'. Moreover, 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 $\mathsf{extract\_lms}_\alpha$, rename_string and order_lms have straightforward definitions, as these are just calls to filter and/or map, and functional correctness specifications, and hence, have correspondingly simple correctness proofs. Consequently, these are not presented here for brevity. The functions induce_α and $\mathsf{rename\_mapping}_\alpha$ have

more intricate functional correctness specifications with induce_α also having a more intricate definition, and so, have more complex correctness proofs. These are described below. Note that the complete proofs are available online[3].

For the induce_α function, we proved three separate theorems. One is for the permutation component, and two are for different sorting components. For the permutation component, we prove the following:

Theorem 3.17 (Induced Sorting Permutation)**.** *If a valid list S has more than one element, then:*

$$\begin{aligned}&\textbf{distinct}\ xs\ \wedge\ \mathsf{set}\ xs = \{i \in \mathbb{N} \mid \textbf{is\_lms}\ S\ i\} \longrightarrow \\ &(\mathsf{induce}_\alpha\ S\ xs) \sim [0 \ldots < |S|]\,,\end{aligned}$$

where **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 reduces to proving that each induced sorting function, $\mathsf{insert\_lms}_\alpha$, $\mathsf{induce\_l}_\alpha$ and $\mathsf{induce\_s}_\alpha$, correctly inserts 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. We achieve this by showing that each of the functions establishes and maintains invariants, such as the distinctness of inserted items, and that items are only inserted in their intended bucket regions. From these, we show that no duplicates are inserted and that the suffix locations are inserted in the correct places in the list.

For the $\mathsf{insert\_lms}_\alpha$ function, we define a version that uses additional ghost variables, in this case two lists, where one list tracks all the LMS-type suffixes that have been inserted and the other lists tracks all the LMS-type suffixes that still need to be inserted. We then prove, as an invariant, that the concatenation of these ghost lists contains all the LMS-type suffix locations. We also prove that after $\mathsf{insert\_lms}_\alpha$ has completed, the ghost list for what still needs to be inserted is now empty, thus implying that all LMS-type suffix locations have been inserted. Finally, the ghost variable and non-ghost variable versions are shown to be equivalent.

For the $\mathsf{induce\_l}_\alpha$ and $\mathsf{induce\_s}_\alpha$ functions, we prove that these maintain invariants about what suffix locations have been inserted. Specifically, 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

[3] https://isa-afp.org/entries/SuffixArray.html

been iterated over. These and the other invariants are used in a proof by contradiction, that shows that all L-type and S-type suffix locations must have been inserted.

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

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

Theorem 3.18 (Induced Sorting LMS-substrings)**.** *If a valid list S has more than one element, then:*

$$\begin{aligned}&\mathbf{distinct}\, P \;\wedge\; \mathsf{set}\, xs = \{i \in \mathbb{N} \mid \mathbf{is\_lms}\, S\, i\} \longrightarrow \\ &\mathbf{sorted}_{\leq_{lex'}}\, (\mathsf{map}\, (\lambda i.\, \mathsf{lms\_slice}_\alpha\, S\, i)\, (\mathsf{induce}_\alpha\, S\, P))\,,\end{aligned}$$

where $\mathsf{lms\_slice}_\alpha\, S\, i$ *returns the* LMS-substring *at position i in S.*

Proof. The proof reduces to proving that each function of induce_α maintains an invariant about inserting suffix locations in sorted order, as described in Sections 3.3.2.2 to 3.3.2.4. We prove these using the invariants utilized in Theorem 3.17, and we also use Theorems 3.15 and 3.16 to show that the ordering is preserved between L-type and S-type LMS-substrings. □

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

Theorem 3.19 (Induced Sorting Suffixes)**.** *If a valid list S has more than one element, then:*

$$\begin{aligned}&\mathbf{distinct}\, R \;\wedge\; \mathsf{set}\, R = \{i \in \mathbb{N} \mid \mathbf{is\_lms}\, S\, i\} \;\wedge\; \mathbf{sorted}_{\leq_{lex'}}\, (\mathsf{map}\, (\lambda i.\, S[i \ldots])\, R) \longrightarrow \\ &\mathbf{sorted}_{\leq_{lex'}}\, (\mathsf{map}\, (\lambda i.\, S[i \ldots])\, (\mathsf{induce}_\alpha\, S\, R))\,.\end{aligned}$$

To prove the correctness of the $\mathsf{rename\_mapping}_\alpha$ function, we show that it constructs the mapping from LMS-substrings of LMS-type suffixes to a new alphabet that preserves the LMS-substring ordering with respect to $\leq_{lex'}$. In our formalization, each LMS-substring is mapped to a natural number, which we refer to as its $\mathsf{rank}_{<_{lex'}}$, that represents how many LMS-substrings of LMS-type suffixes are smaller than it with respect to $<_{lex'}$. Mathematically, the $\mathsf{rank}_<$ function for an arbitrary ordering $<$ is defined as follows:

Definition 3.20 (Rank)**.** Given a comparison function $<$ and a set X, the rank of a value x is defined as follows:

$$\mathsf{rank}_<\, X\, x = \|\{y \mid y \in X \wedge y < x\}\|\,.$$

If X is finite and $<$ is transitive and irreflexive, then the function $\mathsf{rank}_<$ will be monotonic.

The $\mathsf{rename\_mapping}_\alpha$ 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 $\mathsf{rename\_mapping}_\alpha$ is then as follows:

Theorem 3.21 (Correctness of the New Alphabet Mapping)**.**

$$\begin{aligned}&\mathbf{distinct}\, Q \;\wedge\; \mathbf{sorted}_{\leq_{lex'}}(\mathsf{map}\,(\lambda j.\, \mathsf{lms\_slice}_\alpha\, S\, j)\, Q) \;\wedge\; i \in \mathsf{set}\, Q \;\wedge\; i < |S| \longrightarrow \\ &(\mathsf{rename\_mapping}_\alpha\, S\, Q)[i] = \\ &\quad \mathsf{rank}_{<_{lex'}}(\{\mathsf{lms\_slice}_\alpha\, S\, j \mid j \in \mathsf{set}\, Q\})\,(\mathsf{lms\_slice}_\alpha\, S\, i)\,,\end{aligned}$$

where $\mathsf{lms\_slice}_\alpha$ *computes the* LMS-substrings.

Proof. We prove the above by generalizing the $\mathsf{rename\_mapping}_\alpha$ function so that it initializes the accumulator to some arbitrary value k. We then 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 $\mathsf{rename\_mapping}_\alpha$ and Definition 3.20. In the inductive case, where $Q = a \# b \# Q'$, we have two inductive hypotheses. One covers the case when the LMS-substrings at a and b are equal, and the other handles the case when the LMS-substrings at a and b are not equal. Hence, we prove the inductive step 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 $\mathsf{rename\_mapping}_\alpha$ will not be incremented after comparing the two LMS-substrings, leading to both a and b mapping to k according to the definition of $\mathsf{rename\_mapping}_\alpha$. The rest of the proof for this case follows from the first inductive hypothesis.

In the second case, since the 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 $\mathsf{rename\_mapping}_\alpha$. Moreover, we know that $\mathsf{lms\_slice}_\alpha\, S\, a <_{lex'} \mathsf{lms\_slice}_\alpha\, S\, b$, as this follows from the sorted assumption, and the fact that the LMS-substrings at a and b are not equal. Using the fact that Definition 3.20 is monotonic, we then have that the ranks of $\mathsf{lms\_slice}_\alpha\, S\, a$ and $\mathsf{lms\_slice}_\alpha\, S\, b$ will be equal 0 and 1 respectively. The rest of the proof for this case then follows from the second inductive hypothesis. □

Besides the correctness theorems of the functions, the correctness of sais_α 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 3.3.2.1. 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 3.22 (LMS-substring Ordering Implies Suffix Ordering)**.** *For i and j less than $|S|$,*

$$\mathsf{lms\_slice}_\alpha\, S\, i <_{lex'} \mathsf{lms\_slice}_\alpha\, S\, j \longrightarrow S[i\ldots] <_{lex'} S[j\ldots]\,.$$

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

$$\exists b\ c\ as\ bs\ cs.\ \mathsf{lms\_slice}_\alpha\, S\, i = as \mathbin{@} b \# bs \wedge \mathsf{lms\_slice}_\alpha\, S\, j = as \mathbin{@} c \# cs \wedge b < c\,,$$

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

$$\exists a\ as.\ \mathsf{lms\_slice}_\alpha\, S\, i = \mathsf{lms\_slice}_\alpha\, S\, j \mathbin{@} a \# as\,,$$

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

The proof of the first is straightforward 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

$$\mathsf{lms\_slice}_\alpha\, S\, j = bs \mathbin{@} [S[k], S[k+1]]\,,$$

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,

$$\mathsf{lms\_slice}_\alpha\, S\, i = bs \mathbin{@} [S[k], S[k+1], a] \mathbin{@} as\,.$$

Moreover, there exists some xs and ys such that

$$S[i\ldots] = \mathsf{lms\_slice}_\alpha\, S\, i \mathbin{@} xs,\ \text{and}$$

$$S[j\ldots] = \mathsf{lms\_slice}_\alpha\, S\, j \mathbin{@} S[k+1\ldots]\,.$$

The problem thus 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

$$\mathsf{lms\_slice}_\alpha\, S\, i \neq \mathsf{lms\_slice}_\alpha\, S\, j \mathbin{@} a \# as\,,$$

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. Using Theorem 3.15, we then have that $S_i <_{lex'} S_j$. □

With the correctness theorems of the set of functions, and knowing that the LMS-type suffixes of S map to suffixes of the list S', preserving the ordering with respect to $<_{lex'}$, we can then prove that sais_α produces a permutation by induction on the list S, that is:

Theorem 3.23 (sais_α Produces a Permutation)**.**

$$\mathbf{valid}\, S \longrightarrow (\mathsf{sais}_\alpha\, S) \sim [0 \ldots < |S|]\,.$$

Proof. The proofs for its base cases, which are $S = []$ and $S = [x]$ for some arbitrary x, are straightforward, as both of these are degenerate cases for permutations.

In the inductive case where $S = a \# b \# xs$, we use Theorems 3.17, 3.18 and 3.21, the correctness theorems of $\mathsf{extract\_lms}_\alpha$ 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 use Theorem 3.17 to show that sais_α produces a permutation of $[0 \ldots < |S|]$. If the elements of S' are not distinct, we use the inductive hypothesis

$$\mathbf{valid}\, S' \longrightarrow (\mathsf{sais}_\alpha\, S') \sim [0 \ldots < |S'|]$$

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. Theorem 3.17 is then used to complete the proof. □

A similar proof approach is used to show that sais_α sorts suffixes, that is:

Theorem 3.24 (sais_α Sorts Suffixes)**.**

$$\mathbf{valid}\, S \longrightarrow \mathbf{sorted}_{<_{lex}}\, (\mathsf{map}\, (\lambda i.\, S[i \ldots])\, (\mathsf{sais}_\alpha\, S)).$$

Proof. We first prove by induction on the list S that

$$\textbf{valid}\, S \longrightarrow \textbf{sorted}_{<_{lex'}}\, (\mathsf{map}\, (\lambda i.\, S[i \ldots])\, (\mathsf{sais}_\alpha\, S))\,.$$

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

For the inductive case, when $S = a \# b \# xs$, we follow a similar approach to Theorem 3.23, showing that the construction of the reduced list S' is correct, and handling the two cases when S' is distinct and when it is not. However, we now use Theorem 3.19 and the inductive hypothesis, which is now

$$\textbf{valid}\, S' \longrightarrow \textbf{sorted}_{<_{lex'}}\, (\mathsf{map}\, (\lambda i.\, S[i \ldots])\, (\mathsf{sais}_\alpha\, S'))\,,$$

to show that the suffixes are sorted with respect to $\leq_{lex'}$. Note that we also use Theorem 3.23 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 with respect to $<_{lex'}$ if S is valid, we then use the fact that $<_{lex'}$ and $<_{lex}$ are equivalent for valid lists to show the suffixes are sorted with respect to $<_{lex}$. □

By combining Theorems 3.23 and 3.24, we obtain partial correctness of sais_α.

Theorem 3.25 (Partial Correctness of sais_α)**.**

$$\textbf{valid}\, S \longrightarrow \textbf{sa}\, (\mathsf{sais}_\alpha\, S)\,.$$

3.6.2 Proving Equivalence of SA-IS and Abstract Embedding

Recall that the sais_α function is an abstract embedding of the SA-IS algorithm, where the suffix types are computed each time rather than using a pre-computed value like in the algorithm. Moreover, additional array indexing checks are added in the induced sorting functions. We now show that the concrete embedding (sais) of the SA-IS algorithm, which pre-computes 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 pre-computed suffix types, which are computed by get_types at the start of each call to sais, in the functions induce, its sub-functions, extract_lms, and rename_mapping results in sais producing the correct suffix array. That

is, sais is equivalent to sais_α. These proofs are straightforward and follow from the correctness theorem of get_types. The get_types function is simply the implementation of Definition 3.6, and its proof is the formalization of the intuitive arguments presented in Section 3.3.1.

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 more challenging, as the 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 3.26 (Equivalence of sais_α and sais)**.**

$$\textbf{valid}\, S \longrightarrow \mathsf{sais}\, S = \mathsf{sais}_\alpha\, S\,.$$

3.6.3 Termination

Both sais_α and sais require manual termination proofs, as Isabelle's termination checker is not powerful enough to automatically determine if these terminate. These proofs are not difficult as termination for both follows from the fact that the reduced list S', that is, the new list whose suffixes correspond to the LMS-type suffixes of the original list S, is always guaranteed to be strictly smaller than S — in fact, it is less than or equal to half the length of S. 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. Consequently, this means that if S is of length n, then there can only be at most $\lfloor n/2 \rfloor$ many LMS-type suffixes. From this, it follows that sais (and sais_α) terminate.

Theorem 3.27 (Termination of sais)**.**

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

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. Termination is proved by showing that the domain predicate is always true for all inputs, regardless of the validity of the input.

3.6.4 Verifying the Total Correctness of the SA-IS Algorithm

Using Theorem 3.26 to replace sais_α with sais in Theorem 3.25, and combining this with Theorem 3.27, we obtain the total correctness of sais.

Theorem 3.28 (Total Correctness of sais for Valid Lists of Naturals)**.**

$$(\mathbf{terminates}\,\mathsf{sais}) \;\wedge\; (\forall S.\,\mathbf{valid}\,S \longrightarrow \mathbf{sa}S\,(\mathsf{sais}\,S))\,.$$

3.6.5 Verifying the Generalized SA-IS Algorithm

Recall from Section 3.2.1 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, that is, it will return an empty array, as 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 then 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 3.29 (Generalizing Suffix Array Construction)**.**

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

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 an order preserving mapping α from a finite alphabet with a total order to the natural numbers. 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 3.29), 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 3.30 (Total Correctness Generalizes)**.**

$$\begin{aligned}&\textbf{terminates}\,\mathsf{f} \longrightarrow \textbf{terminates}\,(\mathsf{lift}\ f) \quad \wedge \\ &(\forall S.\,\textbf{valid}\ S \longrightarrow \textbf{sa}S\,(\mathsf{f}\ S)) \longrightarrow (\forall S.\,\textbf{sa}S\,(\mathsf{lift}\,\mathsf{f}\ S))\,.\end{aligned}$$

By instantiating f with sais in Theorem 3.30, and using Theorem 3.28, the correctness theorem for a generalized SA-IS algorithm can then be obtained.

Corollary 3.31 (Total Correctness of Generalized SA-IS)**.**

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

3.7 Discussion

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

3.7.1 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 [120, 164]. Nevertheless, an estimate of proof complexity can be determined by considering the number of lines needed for each line of code in the algorithm, and the time taken to complete it [120].

The simple algorithm, outlined in Figure 3.1, is approximately 20 lines when embedded in Isabelle/HOL, and its verification is approximately 180 lines of proof; giving a lines

TABLE 3.1: 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 3.2: Number of lines of algorithm definition, and in the proofs of correctness and additional theorems.

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

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 3.2.

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, sais_α, is composed of over 18,000 lines of proof, plus approximately 4000 further lines for various underlying properties (including, for example, properties about sorting and bijections); also shown in Table 3.2. The equivalence to the concrete Isabelle/HOL implementation, sais, which resembles Figure 3.3 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. Furthermore, 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 were 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 function of the SA-IS algorithm independent of the other functions.

3.7.2 Proof Comparison

Our work formalizes the SA-IS algorithm that was developed by Nong et al. [140], with the formalized theorems and proofs resembling 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 3.7.2.1, is about how the suffix type is defined; the second, described in Section 3.7.2.2, is about how the subsequence comparison is defined; and the third, described in Section 3.7.2.3, is about how the induced sorting function was shown to be correct.

3.7.2.1 Suffix Types

As described in Section 3.5, 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, nor would it have resolved 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. The aim being to cover 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 result in a conflicting suffix type. For example, for the sequence $S = \mathtt{aa}$, $S[0\ldots]$ should be an L-type, as $\mathtt{a} <_{lex} \mathtt{aa}$, but $S[0\ldots]$ should also be an S-type as $S[0] = S[1]$ and $S[1\ldots]$, which is the last suffix, is an S-type, according to case three of Definition 3.4. While this is not a major issue as this is an invalid sequence and using the lifting function, Definition 3.29, 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 3.5.

3.7.2.2 Subsequence Comparison

The subsequence comparison, described in Definition 3.11, is a partial function that takes three arguments: a valid sequence, and two suffix positions. As stated earlier in Section 3.5, 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 3.11 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 3.16. The valid sequence and two suffix positions could then 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 3.16 and the corresponding theorem for LMS-prefixes, would mean that we could just use $<_{lex'}$ in conjunction with Theorem 3.15 for the subsequence comparison. Hence, there is no major incentive to do the extra work of defining a total and binary version of Definition 3.11. Moreover, we argue that $<_{lex'}$ is more elegant, as it provides a better explanation of why the induced sorting function can be used to sort LMS-substrings.

3.7.2.3 Induced Sorting

The correctness proof of the induced sorting function in our formalization differs from the natural language justification provided by Nong et al. [140]. 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 formal presentation and a high degree of rigor. The natural language justification provided by Nong et al. [140], 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 function 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 function sorts the array. However, the informal argument does not provide sufficient detail, if any at all, to show that all suffixes are inserted in their correct positions without any duplicates being introduced. Despite perhaps seeming obvious, this part is the largest and most complex part of the entire Isabelle/HOL verification effort. It accounts for almost 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 suffix array. All of this required much proof exploration to determine suitable invariants, and then further effort to demonstrate their validity.

3.7.3 Verification Approach

As described in Section 3.4, 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 [101] 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 [190] could then be used to handle proofs about asymptotic complexity. Note that an alternative approach is to manually implement the SA-IS algorithm and then verify that it is correct with respect to our formalization, using bottom-up verification tools [33, 66, 67, 148].

3.7.4 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 3.3.1 that the SA-IS algorithm classifies suffixes and 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 [94], which is the predecessor of the SA-IS algorithm, uses a classification that is almost identical, except that it 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. While these differences would require some modifications, this would be minimal, and large portions of our formalization of suffix types could be reused unchanged. Several other algorithms [64, 113, 139], 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 completely reusable as is for the formal verification of these algorithms. Note that other algorithms also classify suffixes into different types [58, 78], however, while their classification of suffixes shares some features, there are still fundamental differences that would most likely require major modifications of our formalization if used.

Induced sorting, detailed in Sections 3.3.2.2 to 3.3.2.4, is an approach used by many suffix array construction algorithms with this being a hallmark of all algorithms in the induced-copying category [160]. The KA algorithm [94] uses the same procedure, shown earlier in Figure 3.8, to induced sort S-type suffixes and uses a slightly altered procedure for induced sorting 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, however, since the S-type suffixes that are not LMS-type suffixes do not affect the order of L-type suffixes, the only major change would most likely be an additional theorem for non-interference. The algorithms based

on the SA-IS algorithm [64, 113, 139] 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 [78] and DivSufSort [58] algorithms, that use different suffix types could reuse our formalization, however, the sorting argument would have to be modified to account for the change in suffix types.

Besides suffix array construction algorithms, our formalization can be reused for the formalization of applications that use suffix arrays. These applications include data compression [35, 144] and string algorithms [2, 87]. For example, recall from Section 3.1 that the BWT [35] 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. By formally verifying the construction of the BWT from the suffix array, which we will see in the next chapter, we can combine this verification with our verified SA-IS Isabelle/HOL implementation to obtain a formally verified and linear-time construction of the BWT.

3.8 Summary

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, thus leading to a deeper understanding of these properties and the SA-IS algorithm.

Chapter 4

Formalized Burrows-Wheeler Transform

Having established a formally verified, linear-time construction of the suffix array via the SA-IS algorithm [140], presented in the previous chapter, we now turn our attention to the *Burrows-Wheeler transform* (BWT) [35], a transformation that has a close relationship with the suffix array. The BWT is an invertible lossless transformation that permutes a sequence into an alternate sequence of the same length that frequently contains long locally homogeneous regions. As a result of being locally homogeneous, the regions tend to consist of only a few distinct symbols and may sometimes include long runs of same-symbol repetitions. As a consequence, the BWT is used in a wide range of tools, particularly those related to data compression, such as `bzip2` [173]. Recall from Section 2.1.3.4 that the BWT is also a crucial component of text indexes, which are suitable for compression [54]. Such text indexes are the basis for many highly sophisticated bioinformatics tools, such as DNA sequencers [105, 106, 111, 112], where an efficient encoding of genomic sequences is essential. As such, ensuring the correctness of the BWT is critical.

In this chapter[1], we continue our investigation into formally verified indexed pattern search by presenting the first formal verification of both the BWT and its inverse. Building on the suffix array formalization presented in the previous chapter, we provide mechanized proofs[2] of correctness, invertibility, and termination for both transformations in

[1]This chapter is based on the paper: L. Cheung, A. Moffat, and C. Rizkallah. Formalized Burrows-Wheeler Transform. In *Proc. Certified Programs and Proofs*, pages 13-26. ACM, 2025. doi: 10.1145/3703595.3705883.

[2]The proofs are published in: L. Cheung and C. Rizkallah. Formalized Burrows-Wheeler transform. *Archive of Formal Proofs*, January 2025. ISSN 2150-914x. https://isa-afp.org/entries/BurrowsWheeler.html. Formal proof development.

Isabelle/HOL [137]. By doing so, we address the gap in the formal verification of compression algorithms and indexed pattern search algorithms that utilize text indexes based on the BWT, ensuring that the BWT can be safely deployed in critical applications, such as genomic data analysis. This work thereby also provides the necessary foundation for verifying the various algorithms for compression and text search that operate on BWT-transformed sequences.

4.1 Introduction

The Burrows-Wheeler transform (BWT) [35] has been the source of much attention, from the time of its conception in 1994, commencing with initial disbelief that it could possibly make sense; and now, thirty years later, having progressed to widespread adoption in the fields of data compression and indexed pattern search.

The importance of the BWT in data compression is due to two key features: localized subsequences and invertibility. First, the BWT transforms typical text into globally non-homogeneous sequences that are the concatenation of numerous smaller subsequences each of which is locally homogeneous and of low localized entropy. The boundaries of these localized subsequences can be identified implicitly via adaptive transformations such as move-to-front, and need not be explicit. The fact that the subsequences are of low-localized entropy means that a subsequent entropy coding step achieves high compression rates within each such subsequence. Simply sorting the input sequence would lead to symbol-based runs of greatest length of course, minimizing the localized entropy. But that rearrangement would not be invertible.

The second critical feature that makes the BWT indispensable is that it has an inverse, BWT^{-1}, that recovers the original sequence. These two features — and the fact that efficient implementations of both BWT and BWT^{-1} have been developed — are why the BWT is widely used in lossless data compression applications such as `bzip2` [173].

The BWT is also a core component for powerful text indexing techniques [54, 55], some of which also support text search queries that run in $\mathcal{O}(m)$ time, where m is the length of the pattern. Such data structures are extensively used in the bioinformatics community for DNA sequencing and alignment tasks [105, 106, 111, 112]. The BWT is useful in text search applications due to its one-to-one correspondence with the suffix array [117].

The one-to-one correspondence between the BWT and the suffix array also means that the BWT can be computed from a suffix array with a single linear scan. Moreover, suffix arrays themselves can be constructed in linear-time [84, 94, 140, 141]. In combination, these two results mean that the BWT for sequence S of length n can be computed in

$\mathcal{O}(n)$ time, using a linear-time suffix array construction algorithm, such as the SA-IS algorithm [140] detailed in the previous chapter.

Due to its widespread adoption in data compression and pattern search applications, the BWT is critically important. Hence, it is highly desirable that the BWT's properties, such as correctness, are formally verified. The work described in this chapter, the formalization and functional correctness proof of the BWT and its inverse transform BWT^{-1}, is the first of its kind, and provides the assurance required for operational confidence. Moreover, it provides a deeper understanding of the BWT's underlying properties that makes it so useful for data compression and pattern search. All the formalization and proofs are mechanized in Isabelle/HOL [137].

4.2 Background

This section provides an overview of the BWT, its inverse transformation, and its correspondence with suffix arrays. While this overview aims to be sufficiently self-contained for understanding our work, readers interested in a more detailed description of the BWT can refer to the work of Navarro [135, Chapter 11], further details of suffix arrays can be found in the work of Crochemore et al. [41, Chapter 4].

Similar to the previous chapter, we assume throughout that sequences are over an ordered alphabet, denoted by Σ, that has a special *minimum element* or *sentinel*, denoted as `$`, that occurs exactly once in any input sequence, and only in the last position. For example, in the programming language C, the null byte that ends each string has the integer value zero. Sequences that satisfy this convention are referred to as valid sequences (see Definition 3.2 in the previous chapter for more detail).

4.2.1 The Burrows-Wheeler Transform

The BWT is a data transformation that produces an invertible permutation of a valid input sequence. The resulting permutation tends to be locally homogeneous but globally heterogeneous, making it ideal for further processing by adaptive data compression techniques.

The canonical form of the BWT, denoted as `bwt` and first presented by Burrows and Wheeler [35], is shown in Algorithm 4.1. In overall terms, the construction can be thought of as starting with a valid n-sequence S; then generating all n distinct rotations of S; then sorting the rotations lexicographically; and then extracting the last column

Algorithm 4.1 BWT Construction (general idea)

```
1: function bwt(S)
2:     M ← a matrix of all the distinct rotations of S
3:     M′ ← sort the rows of M lexicographically
4:     L ← extract the last column of the sorted matrix
5:     return L
6: end function
```

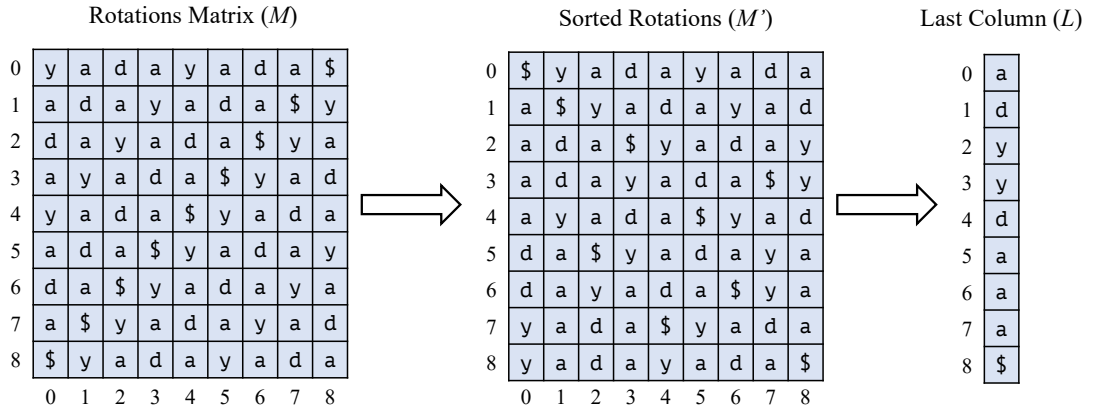


FIGURE 4.1: The construction of the BWT for the sequence $S =$ `yadayada$`. From left to right: the matrix M of rotations of S; the matrix M' of sorted rotations of S; and the last column L of M', the BWT of S.

of the reordered matrix of strings. That final column also contains n symbols and is the BWT-transformed sequence.

An example BWT computation is shown in Figure 4.1 for the sequence $S =$ `yadayada$`, with the BWT-transformed sequence $L =$ `adyydaaa$` emerging on the right. Even at this very limited scale, the BWT tends to form groups of similar symbols. That tendency is amplified as the input sequence becomes longer. For example, when the first paragraph of this chapter (approximately 1 kB) is run through the BWT, shown in Figure 4.2, one part of the transformed sequence appears as `tttttTttttttttttt` (starting at offset 560), and then just a few hundred bytes away transitions to `iaaiooooooorooia` (starting at offset 705). In fact, from offset 703 to offset 783, the sequence mostly consists of vowels. Similar types of subsequences can be found throughout Figure 4.2.

To see why these clusters arise in BWT-transformed text, consider the character subsequence `ainly`. The suffix sorting process will bring every instance of that five-character pattern together, and the corresponding BWT characters — always one to the left of the corresponding suffix — must thus be derived from the words (assuming correctly spelled English input text across a typical literate vocabulary) `ungainly`, `plainly`, `mainly`, `certainly`, and `vainly`. That is, in the locus established by the suffix `ainly` the BWT sequence will be composed of the letters `glmtv`, with the proportions depending on the

```
   0: .m,,,n...eeeees.em..sm2)nssoydss,nSsnoesshsxxhhrefdssalraaoeyesaownt,s
  70: gaesatlftyyyddsttenTTTsTe,ggseerdetgtstsenefn,t,eeeseelagefaeAcyy,,see
 140: etshfhnn4tf,oaegn,ffhyddnfsraswsyy,,,ap#T561]hds]snesr]sAserlssl]]y213
 210: 1141.[#1##1[[p#1.[7.150[301NS#######(##$-D#I###WWWWW#BBBBB-#####50234#
 280: t######i##tttici#cccm##ss####rrrrmhe##l##p#h#bhhhhdnclmmml#Hmrr###aiam
 350: m#iieooinnnnniuuuu#ui#nn##########ineueieeeennee#uinno#hhhhhhrhmhlcmdl
 420: wcghdhsshthhrrcmnRSrhtsttih#rrbrreuuuuu#tgglg#insstunnrsssssrlthvptcvt
 490: xmcxrnl#rr#rmprfddttooooooooo#euufiff##s#nss##nnnnnnnoo#ieelnccccttcc#
 560: tttttTttttttttttsWww#spgt##tvcttmtthfttwfcrhtt###t##lovedra##btstttts
 630: gtgtvhz####s#lshduwrrtffaloaaueebb#esa#baaaa####c#oooauacntllhrlrohr#r
 700: rr#yyiaaiooooooorooiaaaorooiooreeeeeieiaeiiioeetiioiioiaeoe#iuoooaaao
 770: ooeeeoeeeeiiastttllc#########mmibboorsnhhccciiiiiiipll#iiicicccttsffg
 840: fffffchll#ieenrii##eom#mm#aeeeeeeuooeea#tttrreaar#####fppp#peuocaooooo
 910: euefraaoueeact#araAAeiileauiiieennsuAncsilluewmn##ooues##n##nnninssans
 980: l##eeeeeoieiin######en-#naaxxlsnacaaisnaana#tplu##gi################i#
1050 nrisar-esccaaniaencn########sa#ssSsrrlqqqqqqssscsrostBooo#m#n#aeeo####
1120 #oiieeeellnllalllaassb
```

FIGURE 4.2: The BWT of the first paragraph of this chapter with spaces replaced with # symbols.

frequencies of those various words, and with the ordering of the letters determined by what comes after the `ainly` suffix, including the following spaces or punctuation characters. As an even more extreme example, in the sorted-suffix context established by the five-character sequence `ghtly`, the words `rightly`, `nightly`, `tightly`, `brightly`, `lightly`, and `unsightly` (plus similar words) mean that the BWT sequence will likely be a run containing nothing but `i`'s. In the closely related (and thus nearby in the BWT sequence) suffix `ghtle`, the BWT symbols will also likely be dominated by `i` (because of `weightless`, `flightless`, and so on) but might also include sporadic instances of the letter `u`, from `thoughtless`.

In data compression applications, further transformations, such as *move-to-front*, are applied to the BWT sequences, with each highly localized section of the BWT text giving rise to long runs of small integers. The final step applied to the stream of integers is then entropy coding [125].

Finally in this subsection, note that Figure 4.1 is somewhat misleading, since it suggests that the entire square matrix M' is constructed, which would required $\mathcal{O}(n^2)$ space and thus time. In fact, it is not necessary to compute every element in M' in order to determine L, a saving that will become apparent shortly.

	F								*L*
0	$\$_8$	y_0	a_1	d_2	a_3	y_4	a_5	d_6	a_7
1	a_7	$\$_8$	y_0	a_1	d_2	a_3	y_4	a_5	d_6
2	a_5	d_6	a_7	$\$_8$	y_0	a_1	d_2	a_3	y_4
3	a_1	d_2	a_3	y_4	a_5	d_6	a_7	$\$_8$	y_0
4	a_3	y_4	a_5	d_6	a_7	$\$_8$	y_0	a_1	d_2
5	d_6	a_7	$\$_8$	y_0	a_1	d_2	a_3	y_4	a_5
6	d_2	a_3	y_4	a_5	d_6	a_7	$\$_8$	y_0	a_1
7	y_4	a_5	d_6	a_7	$\$_8$	y_0	a_1	d_2	a_3
8	y_0	a_1	d_2	a_3	y_4	a_5	d_6	a_7	$\$_8$
	0	1	2	3	4	5	6	7	8

FIGURE 4.3: The sorted rotations matrix M' of the sequence $S =$ `yadayada$` (see Figure 4.1) with each symbol labeled by its position in the sequence. The first and last columns are labeled as F and L, respectively.

4.2.2 The Inverse Transform

The fact that the BWT is invertible is a consequence of the following three sorted rotations matrix properties.

Proposition 4.1 (Sorted Rotations Matrix Properties)**.** *The first column F and last of column L of the sorted rotations matrix M' are connected in that:*

1. *The first column of the sorted rotations matrix, F, is obtained by sorting the BWT-transformed sequence, L. A simple counting sort is sufficient because the alphabet Σ is known, and is (usually) small.*

2. *For all i, the symbol $L[i]$ occurs immediately to the left of the symbol $F[i]$ in the original sequence S, except in the case when $L[i]$ is the sentinel symbol* `$`.

3. *The relative order for all equal-value symbols is the same in F and in L.*

These three properties can be observed in Figure 4.3, which shows the sorted rotations matrix M' for the input sequence $S =$ `yadayada$`, with each symbol now annotated with its original position in S via a subscript. For example, when $i = 1$ we have $F[i] = \mathtt{a}_7$ and $L[i] = \mathtt{d}_6$, in accordance with property (2). Similarly, and as asserted by property (3), consider the order in which the `a` symbols occur. In both F and L the order is $\mathtt{a}_7$, $\mathtt{a}_5$, $\mathtt{a}_1$ and then $\mathtt{a}_3$.

TABLE 4.1: The values arising from the rank function for the BWT-transformed sequence $L = \texttt{adyydaaa\$}$.

rank L v i		i									
		0	1	2	3	4	5	6	7	8	9
v	$	0	0	0	0	0	0	0	0	0	1
	a	0	1	1	1	1	1	2	3	4	4
	d	0	0	1	1	1	2	2	2	2	2
	y	0	0	0	1	2	2	2	2	2	2

TABLE 4.2: The values arising from the select function for the sorted sequence $F = \texttt{\$aaaaddyy}$.

select F v i		i			
		0	1	2	3
v	$	**0**	-	-	-
	a	**1**	2	3	4
	d	**5**	6	-	-
	y	**7**	8	-	-

To understand *why* property (3) must always hold, note that the ordering in each same-symbol grouping in column F is determined by the suffixes that follow after that symbol, exactly because all of the rotations in that group *do* start with the same symbol. Moreover, when each of those "same initial symbol" rotations is shifted by one so that the column F item goes to column L, the row ordering is again determined by the suffixes that follow those shifted symbols, with the first element of each of those suffixes now the dominant ordering basis in column F. That is, the tie-breaking rule for sorting the rotations that have the same first symbol is the same as the overall sorting rule for sorting the rotations. Hence, all pairs of same-symbol items in column F must appear in the same relative order in column L, and vice versa.

Using Proposition 4.1, the inverse transform, BWT^{-1}, can be computed using the following two query functions on sequences.

- The rank function, when given a sequence xs, a value v and an index i, returns the number of times v occurs in xs up to but not including the i^{th} position.
- The select function, when given a sequence xs, a value v and a natural number i, returns the position in xs at which the i^{th} occurrence of v appears, where counting starts from 0.

To implement the BWT^{-1} the rank function is used on the BWT-transformed sequence, and the select function is applied to the sorted sequence. Examples of these functions for the example sequence presented in Figure 4.1 are shown in Tables 4.1 and 4.2, respectively.

Queries to the rank function for the BWT-transformed sequence L can take linear-time to compute in the worst-case as the entire sequence may need to be scanned. However, all queries can be pre-computed with a single linear-time computation, assuming that the alphabet size is constant. To do so, it suffices to iterate over L once for each symbol in the alphabet, tracking how frequently each symbol occurs. Thus, each single query to the rank function, after pre-computation, will only take constant time.

Queries to the select function for sorted sequences similarly take linear-time to compute in the worst-case. However, all queries can likewise be pre-computed with a single linear-time computation assuming that the alphabet size is constant. This is because only the value for the 0^{th} occurrence is needed, as the i^{th} occurrence (if it exists) can be obtained by adding i to the position to the 0^{th} occurrence; that is, $\mathsf{select}\ F\ v\ i = (\mathsf{select}\ F\ v\ 0) + i$. The 0^{th} occurrences can be obtained by computing the frequencies of all the symbols and summing them up accordingly using a process similar to counting sort. Those initial values are shown in **bold** in Table 4.2. Thus, a single query to the select function, after pre-computation, will only take constant time.

Algorithm 4.2 Inverse BWT Transformation

```
1:  function bwt⁻¹(L)
2:      S : the sequence being constructed
3:      F ← sort L
4:      n ← |L| − 1
5:      v ← $
6:      S[n] ← v
7:      i ← scan L to find where the sentinel symbol $ occurs
8:      while n ≠ 0 do
9:          n ← n − 1
10:         j ← rank L v i
11:         k ← select F v 0
12:         i ← k + j
13:         v ← L[i]
14:         S[n] ← v
15:     end while
16:     return S
17: end function
```

Using the rank and select functions, which are usually pre-computed to have $\mathcal{O}(1)$ query times, assuming that the linear array space required to store them is not an obstacle, the BWT^{-1}, denoted as bwt^{-1}, can be computed, regenerating the original sequence S in right-to-left (that is, reverse) order, as shown in Algorithm 4.2. Broadly speaking, the inverse works by mapping symbols in L to their corresponding locations in F, with this translation often referred to as the *LF-mapping*. The LF-mapping utilizes the fact that the i^{th} occurrence of a symbol v in L corresponds to the i^{th} occurrence of v in F, drawing on property (3) of Proposition 4.1. Property (2) of Proposition 4.1 then allows

TABLE 4.3: Inverting BWT-transformed sequence $L =$ `adyydaaa$` to regenerate the original sequence $S =$ `yadayada$`, extracting symbols in order from $S[8]$ back to $S[0]$.

v	i	$j = \mathsf{rank}\ L\ v\ i$	$k = \mathsf{select}\ F\ v\ 0$	$v = L[k + j]$
-	-	-	-	`$`
`$`	8	0	0	`a`
`a`	0	0	1	`d`
`d`	1	0	5	`a`
`a`	5	1	1	`y`
`y`	2	0	7	`a`
`a`	7	3	1	`d`
`d`	4	1	5	`a`
`a`	6	2	1	`y`

identification of the predecessor of that symbol, thus allowing the array $S[]$ to be filled from right to left.

That is, the LF-mapping established by the calls to the functions rank and select, that is, $\mathsf{rank}\ L\ v\ i$ and $\mathsf{select}\ F\ v\ 0$, respectively, in conjunction with property (3) of Proposition 4.1 ensures that paired symbols between L and F with the same value and the same rank for that value are correctly identified; and then property (2) of Proposition 4.1 allows the symbol that directly precedes the current one to be found. Overall, the original sequence is recovered in right-to-left order, following the chain of predecessor relationships between column F and column L; an iteration that starts by locating the sentinel `$` into $S[n]$ (line 7 of Algorithm 4.2) and ends once $S[0]$ has been reached as a predecessor and its value has been assigned.

For example, the first `a` symbol in both L and F is denoted $\mathtt{a}_7$, as shown in Figure 4.3. This means that in the original sequence this particular `a` is located at position 7. Moreover, using property (2), the symbol that occurs before this can be obtained by looking up L at the position where the first `a` symbol occurs in F. A complete computation of the inverse BWT is shown for the $L =$ `adyydaaa$` in Table 4.3.

4.2.3 One-to-One Correspondence with Suffix Arrays

As mentioned earlier, the BWT has a one-to-one correspondence with the suffix array, meaning that the BWT can be used to solve string matching problems, and can be computed in linear-time. The correspondence between the two becomes clear when comparing the sorted rotations implicitly generated during the construction of the BWT and the sorted suffixes that are generated during the construction of the suffix array. Figure 4.4 shows this correspondence for the same example sequence S.

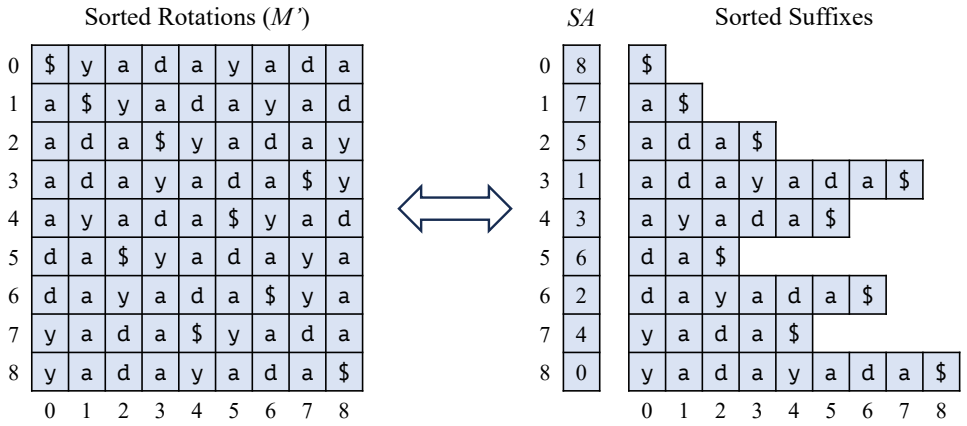


FIGURE 4.4: The correspondence between the sorted rotations matrix and the sorted suffix list for sequence $S =$ `yadayada$`. On the left of the sorted list of suffixes is the suffix array SA, which contains the suffix starting positions in sorted order.

Each suffix is a prefix of exactly one rotation. Moreover, given that a suffix is represented by its starting location within the sequence, a rotation can similarly be represented by the number of left shifts that are needed to obtain it from the sequence. Hence, the mapping between rotations and suffixes is such that the i^{th} rotation maps to the i^{th} suffix and vice versa. For example, the first rotation `adayada$y` maps to the first suffix `adayada$` in Figure 4.4.

Importantly, the mapping between rotations and suffixes is monotonic and preserves the lexicographical ordering. This is because the comparison only ever compares up to the first occurring sentinel symbol `$`, as the sentinel is both the smallest symbol in the alphabet and also occurs only once in the input sequence, meaning that there will be no such case where two distinct rotations will have `$` in the same location. Therefore, the comparison is equivalent to comparing the corresponding suffixes, and hence, lexicographical ordering is preserved. For example, the first rotation `adayada$y` is less than the third rotation `ayada$yada`, and likewise the first suffix `adayada$` is less than the third suffix `ayada$`.

Due to the monotonic one-to-one mapping between rotations and suffixes, the sorted rotations matrix and the sorted suffixes list also have a one-to-one mapping. Hence, the suffix array has a one-to-one mapping with the sorted rotations matrix, as demonstrated by Figure 4.4. The BWT, which is the last column L of the sorted rotations matrix M', can thus be obtained from the suffix array by using the fact that the symbol at the i^{th} position of L occurs in S immediately preceding the symbol at the i^{th} position of F. And F can be obtained by taking the first symbol of each suffix in the sorted suffix list.

Formally, this is expressed as follows:

Definition 4.2 (BWT and Suffix Array Correspondence)**.** Given a sequence S of length n and its suffix array SA, the i^{th} value in the sequence $L = \mathsf{bwt}\, S$, that is, the result of applying the BWT to S, is:

$$L[i] = \begin{cases} S[n-1] & SA[i] = 0\,, \\ S[SA[i]-1] & \text{otherwise.} \end{cases}$$

Note the special case that arises when the suffix array value is equal to 0, corresponding to the original input sequence S. In this case, the value of L is the last symbol of the original sequence, which is always the sentinel.

As an example of the suffix array to BWT mapping, consider Figure 4.4, again applied to the sequence $S = \texttt{yadayada\$}$. With $SA[0]$ equal to the value 8, Definition 4.2 correctly anticipates that the 0^{th} element of the BWT of S is $S[8-1] = \texttt{a}$. Meanwhile, $SA[8]$ is 0, and hence the last element of the BWT of S must be $L[8] = S[8] = \texttt{\$}$.

Note that, as was also the case in connection with the description of the BWT, the figures shown here are illustrative rather than operational. In particular, a suffix array does not require the use of $\mathcal{O}(n^2)$ space to store n explicit suffix sequences (which is how Figure 4.4 might be interpreted); rather, it is the array SA that is stored, together with a single copy of the underpinning sequence S.

4.3 BWT Formalization and Verification

The HOL implementations of the BWT, shown in Algorithm 4.1, and of its inverse, shown in Algorithm 4.2, have been formalized in Isabelle/HOL. This builds on the formalization of suffix arrays, presented in Chapter 3. The BWT formalization and verification also includes additional properties about the correspondence between the BWT and suffix array, and the rank and select functions, which are required by the inverse. The total size of the formalization is 3,844 lines of Isabelle/HOL. Out of this total, 1,787 lines are for the formalization of rank, select, and additional counting properties, 596 lines are for general properties on suffix arrays and bounded incrementing sequences, and the remaining 1,461 lines for the formalization of the BWT and its inverse.

4.3.1 Preliminaries

Before presenting the formalization of the BWT and its inverse, it is important to recall that the BWT requires sequences have their elements drawn from an ordered alphabet

that has a minimum symbol. In the formalization, these sequences are represented using Isabelle/HOL's polymorphic lists, whose elements are of any type that has a total order and a minimum element. In Isabelle/HOL this is natively expressed by stating that the element type is of sort `linorder` and `order_bot`, respectively. Moreover, the formalization also utilizes two key constructs. These are the **valid** predicate and the suffix array construction function `sa` from the suffix array formalization described in Chapter 3. We briefly revisit these definitions here.

The **valid** predicate, shown in Definition 3.2, is used to ensure that lists are terminated by the minimum element in the alphabet and that this minimum element does not occur anywhere else in the list. That is, a list S, whose elements belong to an alphabet that has a total order and contains a minimum element, is valid if it satisfies the following:

$$\textbf{valid}\, S \longleftrightarrow \exists T.\ (S = (T\ @\ [\$])) \wedge (\$ \notin (\mathsf{set}\, T))\,.$$

The suffix array construction function `sa`, as the name suggests, constructs the suffix array of the list it is provided. That is, the function `sa` is a suffix array construction function, according to Definition 3.3, if and only if for all lists S, whose elements belong to a total order, the following is true:

$$(\mathsf{sa}\ S) \sim [0 \ldots < |S|] \ \wedge \textbf{sorted}_{(<)}\,(\mathsf{map}\,(S[\ldots])\,(\mathsf{sa}\ S))$$

where $x \sim y$ means that x is a permutation of y, $[i \ldots < j]$ is the list from i up to but not including j, and $\textbf{sorted}_{(<)}\, x$ means that x is sorted with respect to the strictly less than comparison operator $(<)$.

In the formalization, we do not utilize a specific suffix array construction function but rather use the Isabelle/HOL locale, called *Suffix-Array-General*, where `sa` is the parameter to the locale and the above, that is Definition 3.3, is the locale assumption. Then the formalization of the BWT occurs within the locale context, and thus, has access to definitions and theorems within locale. Note that the formalization of suffix arrays also contains the formal verification of two suffix array construction algorithms: a straightforward forward one, as well as a linear-time algorithm called SA-IS [140]. While our work benefits from these results as they validate this characterization of a suffix array construction, our formalization only directly relies on the locale itself, and hence, only requires one to understand the characterization of a suffix array construction function, that is Definition 3.3.

4.3.2 Formalizing the BWT

In Sections 4.2.1 and 4.2.3, two different formulations of the BWT were presented, one using the canonical description, which would not be efficient to directly implement, and one using a suffix array formulation. These are both formalized and shown to be equivalent. Note that suffix array formulation results in an efficient implementation, albeit with additional practical space costs, as the SA-IS algorithm, detailed in Chapter 3, and other linear time and space suffix array construction algorithms [64, 113, 139] still require the entire suffix array to be stored. While the suffix array is still the same length as its corresponding BWT sequence, an element of the suffix array, which is a numeric value, would need to be at least $\log n$ many bits in size for a sequence of length n. For example, to compute the BWT of a string of length 1024 that consists of ASCII characters, each 8 bits in size, the suffix array that would be required would be of length 1024, but each element would need to be at least 10 bits in size.

The canonical form of the BWT, shown below, is essentially a transliteration of the pseudocode in Algorithm 4.1:

Definition 4.3 (Canonical BWT)**.** For all S, such that **valid** S holds, the canonical BWT definition is:

$$\mathsf{bwt}\ S = \mathsf{map}\ \mathsf{last}\ (\mathsf{sort}_{<_{lex}}\ (\mathsf{map}\ (\lambda i.\ \mathsf{rotate}\ i\ S)\ [0 \ldots < |S|]))\,.$$

It uses the in-built function rotate to generate rotations, $\mathsf{sort}_{<_{lex}}$ for sorting, and last to extract the last elements from the sorted list of rotations. A list S of length n has n rotations, and it is thus sufficient to generate rotations for the values 0 to $n-1$.

The definition of the BWT in terms of the suffix array is similar to Definition 4.2. But instead of a piecewise function, it uses the modulo (`mod`) operator for natural numbers:

Definition 4.4 (Suffix Array Version of the BWT)**.** For all lists S such that **valid** S, and suffix array construction function sa satisfying Definition 3.3, the BWT is computed as follows:

$$\mathsf{bwt\text{-}sa}\ S = \mathsf{map}\ (\lambda i.\ S[(i + |S| - 1)\ \mathsf{mod}\ |S|])\ (\mathsf{sa}\ S)\,.$$

Since indices are represented as natural numbers, the length of the list needs to be added when doing the modulo calculation. This is because, in the calculation above, one is subtracted from the index, which could potentially lead to an invalid result if not handled correctly. Adding the length of the sequence prevents this case from happening, and at the same time preserves the result of the modulo calculation.

To show that Definitions 4.3 and 4.4 are equivalent, the rotations and suffixes are first shown to have the same lexicographical ordering.

Theorem 4.5 (Same Suffix and Rotation Order)**.** *For a list S, if* **valid** S *holds and i, j are less than $|S|$, then*

$$S[i \ldots] < S[j \ldots] \longleftrightarrow \mathsf{rotate}\, i\, S < \mathsf{rotate}\, j\, S\,.$$

The proof of this is straightforward and reduces to showing that

1. the i^{th} suffix is a prefix of the i^{th} rotation and
2. the lexicographical comparison between rotations only ever compares up to the first (and only) occurrence of the minimum element `$`.

Both of these follow from the definition of a rotation and of lexicographical order.

Theorem 4.5 implies that generating the rotations from the suffix array is equivalent to generating and then sorting the rotations. With this and the fact that taking the last element of the i^{th} rotation of a sequence S is equal to

$$S[(i + |S| - 1) \bmod |S|]$$

the equivalence of the different formulations of the BWT is obtained; as stated by the following theorem.

Theorem 4.6 (BWT and Suffix Array Correspondence)**.** *For a list S,*

$$\textbf{valid}\, S \longrightarrow \mathsf{bwt}\, S = \mathsf{bwt\text{-}sa}\, S\,.$$

4.3.3 Rank and Select

Before moving to the inverse transform, we briefly describe the formalization of the rank and select functions and some key properties that are necessary to understand the rest of the formalization. The goal of this formalization is to define the semantics and properties of rank and select. Efficient implementations are outside the scope of this work, but we note that numerous implementations have been developed [61] and formally verifying such efficient implementations deserves its own separate investigation. The definitions for rank and select, as well as some of their properties, are in part based on the work of Affeldt et al. [6]. However, we also formalize several additional properties that turned out to be necessary including Theorems 4.10 and 4.11.

Recall that $\mathsf{rank}\ xs\ v\ i$ returns the number of occurrences of the value v in the list xs up to (but not including) the i^{th} position in xs. This can be computed using Isabelle/HOL's count-list function, which counts how many times a value occurs in a list, and the take function, which returns the prefix up to the i^{th} location:

Definition 4.7 (Rank)**.**

$$\mathsf{rank}\ xs\ v\ i = \mathsf{count\text{-}list}\ (\mathsf{take}\ i\ xs)\ v\ .$$

The $\mathsf{select}\ xs\ v\ i$ function returns the location of the i^{th} occurrence of a value v in the list xs. We define this using a recursive function that iterates over the list, decrementing i each time the current element equals v. When i is equal to zero and the current value is equal to v, that location is returned. If such a scenario does not occur, then the return value is $|xs|$:

Definition 4.8 (Select)**.**

$$\begin{aligned}
&\mathsf{select}\ []\ v\ i = 0\\
&\mathsf{select}\ (a\ \#\ xs)\ v\ 0 = (\text{if}\ v = a\ \text{then}\ 0\ \text{else}\ \mathsf{Suc}\ (\mathsf{select}\ xs\ v\ 0))\\
&\mathsf{select}\ (a\ \#\ xs)\ v\ (\mathsf{Suc}\ i) =\\
&\quad (\text{if}\ v = a\ \text{then}\ \mathsf{Suc}\ (\mathsf{select}\ xs\ v\ i)\\
&\quad\ \text{else}\ \mathsf{Suc}\ (\mathsf{select}\ xs\ v\ (\mathsf{Suc}\ i)))\ .
\end{aligned}$$

The select function is more complex than the rank function, and requires a correctness theorem.

Theorem 4.9 (Correctness of Select)**.**

$$\begin{aligned}
&\mathsf{select}\ xs\ v\ i \leq |xs| \wedge\\
&(\mathsf{select}\ xs\ v\ i < |xs| \longrightarrow \mathsf{rank}\ xs\ v\ (\mathsf{select}\ xs\ v\ i) = i) \wedge\\
&(\mathsf{select}\ xs\ v\ i = |xs| \longrightarrow \mathsf{count\text{-}list}\ xs\ v \leq i)\ .
\end{aligned}$$

Essentially, this guarantees that if select returns a valid index, then it corresponds to the i^{th} occurrence of v in xs. Otherwise, it must mean that there is no i^{th} occurrence v in xs. The above is proven by induction on the list.

The description of the BWT^{-1} provided earlier stated that if the list is sorted, then only $\mathsf{select}\ xs\ v\ 0$ needs to be known. We prove this by induction on the list:

Theorem 4.10 (Select Sorted Equivalence)**.** *If xs is sorted and* $i <$ count-list $xs\, v$ *then*

$$\mathsf{select}\ xs\, v\, i = \|\{k.\ k < |xs| \wedge xs[k] < v\}\| + i\,,$$

where $\|s\|$ *returns the cardinality of a given set* s*.*

An alternative formulation for rank is now possible, capturing the synergy with Theorem 4.10:

Theorem 4.11 (Rank Equivalence)**.**

$$\mathsf{rank}\ xs\, v\, i = \|\{k.\ k < |xs| \wedge k < i \wedge xs[k] = v\}\|\,.$$

Proving this theorem reduces to showing that

$$\mathsf{count\text{-}list}\ xs\, v = \|\{k.\ k < |xs| \wedge xs[k] = v\}\|$$

This is proven by straightforward induction on the list. Then the rest of the proof of Theorem 4.11 follows from this fact along with Definition 4.7.

4.3.4 Sorted Rotations Matrix Properties

The LF-mapping is a core component of BWT^{-1}. Recall that this maps positions in the BWT-transformed sequence L (the last column of the sorted rotations matrix M') to the corresponding positions in the sorted sequence F, the first column in the sorted rotations matrix. The mapping demonstrates that the i^{th} occurrence of v in L is also the i^{th} occurrence of v in F.

The LF-mapping is a consequence of Proposition 4.1, and so, these properties need to be shown to be true. This is achieved by using the correspondence with suffix arrays. Specifically, the BWT permutation specifies how the indices of the original sequence are permuted to produce the BWT, which is defined in terms of the suffix array as follows:

Definition 4.12 (BWT Permutation)**.** For a list S, where **valid** S holds, and suffix array construction function sa, the BWT permutation is:

$$\mathsf{bwt\text{-}perm}\ S = \mathsf{map}\ (\lambda i.\ (i + |S| - 1)\ \mathsf{mod}\ |S|)\ (\mathsf{sa}\ S)\,.$$

Property (2) of Proposition 4.1 immediately follows. In addition, property (1) is also trivial to prove, as the suffix array can be viewed as the permutation mapping of F, the

first column of the sorted matrix. Hence for a valid list S,

$$F = \mathsf{map}\,(\lambda i.\, S[i])\,(\mathsf{sa}\; S)\,,$$

which is sorted because the suffix array represents the sorted suffixes S, as per Definition 3.3.

Property (3) of Proposition 4.1, shown below, is also straightforward to prove, and again follows from property (2) of Proposition 4.1, Definition 3.3, and the definition of lexicographical order.

Theorem 4.13 (Suffix Relative Order Preservation)**.** *Given a valid list S, if $i < j$, $j < |S|$ then*

$$S[(\mathsf{bwt\text{-}perm}\; S)[i]] = S[(\mathsf{bwt\text{-}perm}\; S)[j]] \longrightarrow$$
$$S[((\mathsf{bwt\text{-}perm}\; S)[i])\ldots] < S[((\mathsf{bwt\text{-}perm}\; S)[j])\ldots]\,.$$

Note that in the formalization, only properties (1) and (3) are explicitly proven, as property (2) is immediately obtained by term rewriting.

4.3.5 Verifying the Correctness of the Inverse

The Isabelle/HOL embedding of the BWT^{-1}, bwt^{-1}, closely resembles Algorithm 4.2 presented in Section 4.2. It uses a loop, in this case a recursive function $\mathsf{bwt}_\mathsf{n}^{-1}$, that repeatedly applies the LF-mapping, $\mathsf{lf\text{-}map}$, to find the next symbol in the original sequence in reverse order. These are defined as follows:

Definition 4.14 (Inverse BWT)**.**

$$\mathsf{lf\text{-}map}\; ss\; bs\; i = (\mathsf{select}\; ss\; bs[i]\; 0) + (\mathsf{rank}\; bs\; bs[i]\; i)$$

$$\mathsf{bwt}_\mathsf{n}^{-1}\; 0\; ss\; bs\; i = []$$
$$\mathsf{bwt}_\mathsf{n}^{-1}\; (\mathsf{Suc}\; n)\; ss\; bs\; i = (\mathsf{bwt}_\mathsf{n}^{-1}\; n\; ss\; bs\; (\mathsf{lf\text{-}map}\; ss\; bs\; i))\;\; @\; [bs[i]]$$

$$\mathsf{bwt}^{-1}\; bs = \mathsf{bwt}_\mathsf{n}^{-1}\; |bs|\; (\mathsf{sort}_{\leq}\; bs)\; bs\; (\mathsf{select}\; bs\; \bot\; 0)\,,$$

where @ i s the concatenation operator for two lists.

For bwt^{-1} to be correct, it needs to satisfy:

$$\mathbf{valid}\; S \longrightarrow \mathsf{bwt}^{-1}\; (\mathsf{bwt}\; S) = S\,,$$

This is achieved by verifying the correctness of $\mathsf{lf\text{-}map}$ and $\mathsf{bwt}_\mathsf{n}^{-1}$. These in turn are proven correct by proving the correctness of $\mathsf{lf\text{-}map}_\alpha$, an abstract version of $\mathsf{lf\text{-}map}$, and bwt_α^{-1}, an abstract version of the inverse BWT permutation, respectively, and then verifying that these abstract versions are equivalent to the concrete ones. We call $\mathsf{lf\text{-}map}_\alpha$ and bwt_α^{-1} abstract as these have access to additional information about the original sequence and its suffix array. When these are made concrete, all information about the original sequence and its suffix array is removed, and only information that can be derived from the BWT-transformed sequence itself may be assumed.

The abstract LF-mapping, $\mathsf{lf\text{-}map}_\alpha$, defined below, takes the original list S, and a position i in L, the last column of the sorted rotations matrix and BWT-transformed sequence of S. It returns the location in F (first column of the sorted rotations matrix) of the value $L[i]$. That is, given the k^{th} occurrence of some value v in L, it returns the k^{th} occurrence of v in F.

Definition 4.15 (Abstract LF-mapping)**.** Given a list S, such that **valid** S, then

$$\mathsf{lf\text{-}map}_\alpha\ S\ i = \mathsf{select}\ (\mathsf{sort}_\leq\ S)\ (\mathsf{bwt\text{-}sa}\ S)[i]\ (\mathsf{rank}\ (\mathsf{bwt\text{-}sa}\ S)\ S[i]\ i)\,.$$

The abstract BWT^{-1} permutation, defined below, is essentially the repeated application the abstract LF-mapping. Note that this is referred to as abstract, as it too uses the original S rather than the BWT-transformed sequence.

Definition 4.16 (Inverse BWT Permutation)**.**

$$\begin{aligned}&\mathsf{bwt}_\alpha^{-1}\ 0\ S\ i = []\\&\mathsf{bwt}_\alpha^{-1}\ (\mathsf{Suc}\ n)\ S\ i = \mathsf{bwt}_\alpha^{-1}\ n\ S\ (\mathsf{lf\text{-}map}_\alpha\ S\ i)\ \ @\ [i]\,.\end{aligned}$$

Strictly speaking, the above returns a sub-list of a rotation of the BWT^{-1} permutation, as it depends on the initial value of i and the number of times $\mathsf{lf\text{-}map}_\alpha$ is applied. If the initial starting value is the location of the minimum value $\perp$ in the BWT-transformed sequence, and it iterates for the length of the sequence, then it will return the BWT^{-1} permutation.

The refinement proofs from the abstract definitions to the concrete ones, shown in Definition 4.14, are omitted from the presentation as they are straightforward and are largely proved using standard term rewriting. For example, the refinement from $\mathsf{lf\text{-}map}_\alpha$ to $\mathsf{lf\text{-}map}$ follows from the fact that for a sequence S,

$$\mathsf{sort}_\leq\ S = \mathsf{sort}_\leq\ (\mathsf{bwt\text{-}sa}\ S)\,,$$

and from Theorem 4.10,

$$\mathsf{select}\ (\mathsf{sort}_{\leq}\ S)\ v\ i = \mathsf{select}\ (\mathsf{sort}_{\leq}\ S)\ v\ 0 + i\,.$$

A similar argument is also used for the refinement from abstract to concrete of the BWT^{-1} permutation.

To show that Definition 4.15 is correct, the k^{th} occurrences of v in L and F need to be shown to have come from the same location in S. This can be shown using Theorem 4.13 and is formalized as follows:

Theorem 4.17 (Same Rank)**.** *For valid list S, if i and j are less than $|S|$, and $S[(\mathsf{sa}\ S)[i]]$ and $S[(\mathsf{bwt\text{-}perm}\ S)[j]]$ are equal to v then*

$$\mathsf{rank}\ \mathsf{sort}_{\leq}\ S\ v\ i = \mathsf{rank}\ \mathsf{bwt\text{-}sa}\ S\ v\ j \longrightarrow (\mathsf{sa}\ S)[i] = (\mathsf{bwt\text{-}perm}\ S)[j]\,.$$

The correctness of the abstract LF-mapping then follows:

Theorem 4.18 (Abstract LF-Mapping Correctness)**.** *For a valid list S and $i < |S|$, then*

$$(\mathsf{bwt\text{-}perm}\ S)[\mathsf{lf\text{-}map}_{\alpha}\ S\ i] = ((\mathsf{bwt\text{-}perm}\ S)[i] + |S| - 1)\ \mathsf{mod}\ |S|\,.$$

Theorem 4.18 states that $\mathsf{lf\text{-}map}_{\alpha}$ finds the next value in the BWT^{-1} permutation in reverse order. Hence, repeated applications of $\mathsf{lf\text{-}map}_{\alpha}$ by $\mathsf{bwt}_{\alpha}^{-1}$ should produce the whole BWT^{-1} permutation. One slight complication is that depending on what initial starting index is used, the inverse BWT permutation may be rotated.

To solve this, the following is useful:

Theorem 4.19 (Abstract Inverse BWT Permutation Rotated Sub-list)**.** *Suppose that S is a list such that* **valid** *S, $i < |S|$, and $n \leq |S|$, then*

$$\exists k\ ys\ zs.\ \mathsf{rotate}\ k\ [0\,..\!<\,|S|] = ys\ @\ xs\ @\ zs\,,$$

where

$$xs = \mathsf{map}\ (\lambda j.\ (\mathsf{bwt\text{-}perm}\ S)[j])\ (\mathsf{bwt}_{\alpha}^{-1}\ n\ S\ i)\,.$$

This follows from the fact that repeated applications of $\mathsf{lf\text{-}map}_{\alpha}$ produces a sequence that when passed to $\mathsf{bwt\text{-}perm}$, becomes a bounded incrementing sequence, which is a sequence that increases by one, eventually wrapping back to zero after reaching the maximum $|S| - 1$. While there are several additional proof steps involved, the core component follows from Theorem 4.18.

From Definition 4.16, it becomes clear that a list of length n will be produced when the number of iterations is n, with n being the first argument. Moreover, if the initial value of i is equal to the location of $\perp$ in the BWT-transformed sequence L, that is, $i = \mathsf{select}\ L \perp 0$, the following holds:

$$(\mathsf{bwt\text{-}perm}\ S)[(\mathsf{bwt}_\alpha^{-1}\ |S|\ S\ i)[|S| - 1]] = |S| - 1\,.$$

Combining that with Theorems 4.6 and 4.19 and the refinement theorems, the correctness of BWT^{-1} can then be obtained.

Theorem 4.20 (Correctness of the Inverse BWT)**.** *For a valid list S,*

$$\mathsf{bwt}^{-1}\ (\mathsf{bwt}\ S) = S\,.$$

We reiterate that the complete proofs are available online[3].

4.4 Discussion

In this section, we present our reasoning for formalizing the BWT using suffix arrays, note some difficulties that were encountered, and consider how to instantiate Definitions 4.4 and 4.14 into efficient implementations.

4.4.1 The BWT in Terms of Suffix Arrays

In our formalization of the BWT and its inverse, we utilized the one-to-one correspondence between the BWT and suffix array. While it is possible to specify the BWT without using suffix arrays, we found that doing so was more complicated. This is because suffix arrays succinctly express positional information, notably as captured by Definition 3.3, which is useful for the verification of the inverse. For example, the BWT permutation, Definition 4.12, is expressed in terms of the suffix array.

Another benefit of using suffix arrays is that they are also useful in a computational sense, allowing the BWT for a sequence to be calculated in a simple manner. Furthermore, if we have a formally verified suffix array construction algorithm, such as the SA-IS algorithm that we verified in Chapter 3, we can easily plug that into our specification to obtain a formally verified BWT construction algorithm. An algorithm for computing the BWT will then inherit the same time complexity as the suffix array construction algorithm, since any suffix array construction algorithm must require time that is $\mathcal{O}(n)$ or greater.

[3] https://isa-afp.org/entries/BurrowsWheeler.html

4.4.2 Formalization Challenges

One of the major difficulties encountered was capturing the logic of why the BWT is invertible. The key challenge here was finding a suitable formal description for Proposition 4.1, specifically the third property. That is, encoding that the i^{th} occurrence of a symbol x in the BWT of a text is the i^{th} occurrence of the symbol x in the sorted text. The difficulty here is that a symbol x is not distinct from another symbol x. While our initial description of this property augments each symbol by its position in the text, we decided to reformulate the property in terms of suffixes, as we could reuse the suffix array formalization, presented in Chapter 3.

In terms of theorem proving, while no major obstacles were encountered when creating the proofs, there was little existing theory on properties about suffix arrays, finding the k^{th} occurrence of some value in a list, and for bounded incrementing sequences. Hence, many of these were formalized in this work, becoming a significant proportion of the formalization. Indeed, over half the total of the lines of Isabelle/HOL were for formalizing these properties. Theorems that were the intersection of two or more of these areas were particularly difficult to prove, such as Theorems 4.17 and 4.19. We trust these will be able to be reused by others.

4.4.3 Limitations and Future Work

Our formalization of the BWT and its inverse is not intended to produce efficient implementations. Rather, it is meant to be a mathematical formalization, specifying logical properties about the BWT. Efficient implementations can be constructed from our formalization using approaches described in Section 3.7.3.

The embedding of BWT that is computed via a suffix array, that is, Definition 4.4, can be made into a linear-time implementation, given a linear-time suffix array construction function, such as the SA-IS algorithm that we verified in Chapter 3. While such an implementation would be fast, it does have a larger memory overhead, as the whole suffix array needs to be constructed and stored before the BWT sequence can be generated. This might not be suitable for very large inputs.

The inverse transformation, BWT^{-1}, presented in Definition 4.14, requires further refinement into a linear-time implementation. At present, the rank and select functions are represented rather naïvely. However, this could be resolved by providing an efficient precomputation and verifying that it is a refinement of our straightforward implementation, as is done in practiced data compression systems [173].

The select function could be replaced with a data structure such as that shown in Table 4.2, that can be computed by an approach similar to counting sort. The partner rank function could be computed by calculating the cumulative frequencies for each symbol in the text. For constant-sized alphabets, this would still provide a linear-time computation. However, this approach has an alphabet-sized memory overhead. Reductions are possible but would affect the running time, and hence change the overall running time of the inverse BWT. One possible way to overcome this obstacle is to formalize more advanced rank-and-select data structures [134].

4.5 Summary

We have presented the first formal verification of both the BWT and its inverse. The formalization consists of a mechanized proof of correctness, invertibility and termination for both transformations, using Isabelle/HOL. We have thus addressed an important gap in the formal verification of data compression and indexed pattern search algorithms, ensuring that the BWT can be safely deployed in critical applications. In particular, this work provides the necessary foundation that is required for verifying the various algorithms for compression and text search that operate on BWT-transformed sequences.

Chapter 5

Formally Verified Indexed Pattern Search

Chapters 3 and 4 addressed the formal verification of text index construction, a crucial component of verified indexed pattern search. Chapter 3 presented the verification of the linear-time SA-IS algorithm for suffix array construction, while Chapter 4 presented a formalization of the Burrows-Wheeler Transform (BWT). This chapter[1] completes the formal verification pipeline for indexed pattern search by focusing on the search phase with an emphasis on search algorithms that use suffix arrays and the BWT as text indexes. The central contribution is a machine-checked correctness proof of the backward search algorithm [54], which enables efficient pattern matching over a text using only the BWT of that text, supported by auxiliary data structures. To establish this result, we first formalize the substring search problem and verify the binary search algorithm [92, Chapter 6.2.1] over suffix arrays, the latter of which also serves as a point of comparison.

5.1 Introduction

Recall from Section 2.1 that indexed pattern search is an efficient approach for solving pattern search related tasks, and is particularly good at solving the substring search problem. The substring search problem involves finding the occurrences of a sequence of symbols, called a *pattern*, in another sequence of symbols, called a *text*, that is typically longer than the pattern. Indexed pattern search involves pre-constructing text indexes, which are data structures that augment the text with additional information. Text indexes enable search for arbitrary patterns in the text with worst-case time complexities

[1] This chapter is unpublished material that is in the process of being prepared for submission. The complete formalization is located in Appendix A.

that, ideally, scale linearly in terms of the pattern length. While the cost of constructing a text index varies depending on what particular text index is constructed, the text index of a static text only needs to be constructed once, thus amortizing the construction cost over all subsequent search operations for that text.

The suffix array [117], discussed in detail in Chapter 3, is one particularly useful text index for indexed pattern search. This is because the substring search problem can be reformulated as finding all suffixes of the text that have the pattern as a prefix. Moreover, the suffix array stores all the suffixes of the text, which are represented by their locations in the text, in lexicographical order. Naturally, this results in all suffixes that start with the same prefix being located in one contiguous region in the suffix array, resulting in all occurrences of a pattern in a text being located in such a contiguous region in the text's suffix array. The streamlined structure of the suffix array thus makes it particularly amenable to efficient search using simple algorithms.

The BWT [35] of a text, discussed in detail in Chapter 4, is another useful text index for indexed pattern search. This is because the BWT has a one-to-one correspondence with the suffix array, and so operations on the suffix array, such as search, can be simulated on the BWT. When using the suffix array of the text for indexed pattern search, both the suffix array and text are essential components. However, when using the BWT of the text, only the BWT of the text is required, while the auxiliary data structures only make the search process more efficient. Due to this and the fact that the BWT is used in data compression [173], the BWT is often used for particularly large texts, such as genome sequences [40, 105, 106, 111, 112]. Moreover, the auxiliary data structures, which are computable from the BWT, are also used to compute the inverse of the BWT, as described in Chapter 4, and can be compressed, albeit with reduced efficiency for search and inverse operations.

Given that the suffix array and the BWT are important text indexes for indexed pattern search, it highly desirable that both their construction and use in search algorithms are formally verified. The former was addressed in Chapters 3 and 4 for suffix arrays and the BWT, respectively. The work of this chapter addresses the latter and presents the formal verification two search algorithms: binary search [92, Chapter 6.2.1] using suffix arrays, and backward search [54] using the BWT. The main result of this chapter is the verification of the backward search algorithm with the verification of binary search being used to help establish this result. The formalization of these algorithms are mechanized in Isabelle/HOL [137] and builds on the formalization of suffix arrays and the BWT detailed in Chapters 3 and 4, respectively.

The remainder of this chapter is structured as follows. Section 5.2 provides an overview of the substring search problem and how binary search with suffix arrays and backward

search with the BWT solve this problem. Sections 5.3 and 5.4 present the formalization of the substring search problem in its canonical form and an alternate formulation in terms of a search interval over the suffix array, respectively. Sections 5.5 and 5.6 present the verification of indexed pattern search by means of binary search with suffix arrays and backwards search using the BWT, respectively. Section 5.8 concludes this chapter.

5.2 Background

This section begins with a summary of the substring search problem, followed by a description of how suffix arrays are used as text indexes by algorithms like binary search, and concludes with a description of the backward search algorithm, which uses the BWT as a text index. While this overview aims to be sufficiently self-contained for understanding this work, readers interested in more detailed descriptions of the suffix array, the BWT and the backward search algorithm, and the binary search algorithm can refer to the works of Crochemore et al. [41, Chapter 4], Navarro [135, Chapter 11] and Knuth [92, Chapter 6], respectively.

5.2.1 Substring Search Problem

Recall from Chapter 2.1 that the substring search problem in its most basic form entails finding an occurrence of a pattern P of length m in a text S of length n, where the pattern and text are sequences over a common alphabet Σ. Besides using the brute force approach, where each substring of S of length m is compared with P, there are two main approaches available: preprocessing the pattern, and preprocessing the text. The latter is what we refer to as indexed pattern search.

The aim of preprocessing the pattern P is to eliminate as many redundant string comparisons with substrings of S. Algorithms that use this approach have worst-case time complexities that are at least $\Omega(m + n)$ and in most cases have worst-case space complexities of $\mathcal{O}(m)$. This is because an m length pattern is scanned at least once during preprocessing and only stores information related to the pattern. Moreover, an n length text must be scanned at least once during the search for pathological cases, such as when the pattern is not a substring of the text or when finding all occurrences of a pattern in a text.

Notable algorithms that preprocess the pattern are the Knuth-Morris-Pratt (KMP) algorithm [93], the Boyer-Moore (BM) algorithm [31], and the Rabin-Karp algorithm [85]. The KMP algorithm uses a failure function, which requires $\Theta(m)$ time and space to build,

to avoid redundant string comparisons after a symbol mismatch, resulting in a worst-case search time cost of $\Theta(n)$. The BM algorithm uses bad character and good suffix heuristics, which requires $\Theta(m)$ time and $\Theta(m + |\Sigma|)$ space to build, to likewise avoid redundant string comparisons. While in practice bad character and good suffix heuristics speeds up the substring search, the theoretical worst-case search time cost is still $\mathcal{O}(mn)$. The Rabin-Karp algorithm hashes the pattern, which requires $\mathcal{O}(m)$ time and $\mathcal{O}(1)$ space to compute, and compares this with the hashes of the text's substrings to quickly identify potential matches. While in practice hashing helps to eliminate many redundant substring comparisons, the theoretical worst-case search time cost is still $\mathcal{O}(mn)$.

The aim of indexed pattern search is to use additional space proportional to the text S to construct an index that organizes S so that substrings that match any arbitrary pattern can be easily located. In the context of the substring search problem, there are many different types of text indexes, including trie-like data structures [172, Chapter 5.2], like the suffix tree [69, Chapter 5], and the suffix array [41, Chapter 4]. Based on which index structure is used, the search time cost and index building cost varies, however in most cases, it is possible to search for all occurrences of an arbitrary pattern using such an index in $\mathcal{O}(m)$ time, and to build the index in $\mathcal{O}(n)$ time.

Both pattern search approaches have their advantages and disadvantages. As stated above, pattern preprocessing approaches generally have space requirements that are proportional to the pattern length, while indexed pattern search approaches have space requirements that are proportional to the text length, where pattern length is usually significantly shorter than the text length. Pattern preprocessing generally only takes time proportional to the pattern length, while constructing the text index takes time proportional to the text. However, worst-cast search time costs are always at least linear in terms of the text length for pattern preprocessing approaches, while indexed pattern search can have worst-case search time costs that are linear in terms of the pattern length. Moreover, when the texts are static, the pattern preprocessing information must be recomputed for each new pattern, while the text index can be reused unchanged for any arbitrary pattern.

5.2.2 The Suffix Array as a Text Index

Recall from Chapter 3 that the suffix array organizes the locations of suffixes of a text S in such a way that the locations are ordered with respect to the lexicographical ordering of the suffixes. That is, all suffixes beginning with the same prefix appear in one contiguous region in the suffix array. Hence, searching for a pattern P in S is equivalent to finding

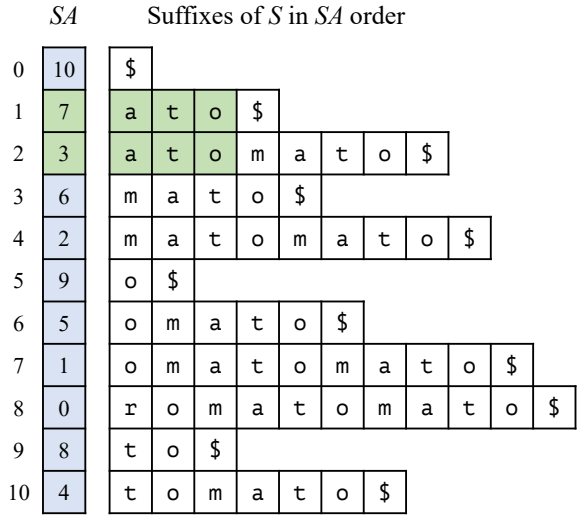


FIGURE 5.1: The suffix array (SA) of the sequence $S = \texttt{romatomato\$}$ with the occurrences of the pattern $P = \texttt{ato}$ shaded green.

a suffix in a region in the suffix array where all suffixes begin with P. Moreover, finding all occurrences of P in S can be achieved by finding the start and end of such a region.

For example, consider the suffix array of the sequence $S = \texttt{romatomato\$}$ shown in Figure 5.1. Notice that all suffixes with the same prefix are grouped together, and so, if we search for the pattern $P = \texttt{ato}$ in S, we can see that the suffixes that begin with P lie in the interval of the suffix array $\{1 \ldots < 3\}$, where $\{x \ldots < y\}$ can be viewed as the set of all natural numbers between x (inclusive) and y (exclusisve). Looking up the suffix array within this interval, we observe that the suffixes at positions 7 and 3 both start with `ato`.

For the suffix array to be useful for substring search, the process of finding regions of suffixes beginning with the pattern must not only be feasible but also efficient. Since the suffix array stores suffixes in lexicographical order, several approaches are available. A straightforward approach is to use binary search, which can find the region in $\mathcal{O}(m \log n)$, where the $\mathcal{O}(m)$ factor is caused by the comparisons with the m length pattern. With auxiliary data structures, such as the LCP array [117], which stores the length of the longest common prefix between two adjacent suffixes in the suffix array, this time cost becomes linear in terms of the pattern length, that is, $\mathcal{O}(m)$.

5.2.3 Binary Search for Indexed Pattern Search

Standard binary search is an algorithm that determines if a value v is in a sorted sequence xs and if it is in xs, locates one position at which it occurs. The core idea of binary search is that since xs is sorted, the search range over xs can be iteratively shrunk by

Algorithm 5.1 Binary Search (leftmost)

```
1:  function binary-search-l(xs , v)
2:      l ← 0
3:      r ← |xs|
4:      while l < r do
5:          m ← floor((l + r)/2)
6:          if xs[m] < v then
7:              l ← m + 1
8:          else
9:              r ← m
10:         end if
11:     end while
12:     return l
13: end function
```

Algorithm 5.2 Binary Search (rightmost)

```
1:  function binary-search-r(xs , v)
2:      l ← 0
3:      r ← |xs|
4:      while l < r do
5:          m ← floor((l + r)/2)
6:          if xs[m] > v then
7:              r ← m
8:          else
9:              l ← m + 1
10:         end if
11:     end while
12:     return r
13: end function
```

half, comparing at each iteration the value currently in the middle of the search range e with v. When v matches e or when the search range becomes empty, the algorithm terminates. If v is less than e, then the lower half, which contains the values in xs that are less than e, becomes the new search range. If v is greater than e then the upper half, which contains the values in xs that are greater than e, becomes the new search range.

With careful modification, binary search can be altered so that it finds the first instance in the array xs that matches the search value v, and similar modifications for finding the first position after last instance matching v. These two versions are shown in Algorithms 5.1 and 5.2. Strictly speaking, the version that finds the first instance matching v returns the location of the first value in xs that is greater than or equal to v. Similarly, the version that finds the last instance matching v returns the location of the first value in xs that is strictly greater than v.

When searching for patterns in a text using a suffix array, the two versions of binary search are used to find the interval over the suffix array containing all matching instances.

TABLE 5.1: Binary search example for the pattern $P = \texttt{ato}$ on the sequence $S = \texttt{romatomato\$}$.

Iteration	Leftmost				Rightmost			
	l	r	m	$S[SA[m]\ldots]$	l	r	m	$S[SA[m]\ldots]$
-	0	11	-	-	0	11	-	-
0	0	11	5	`o$`	0	11	5	`o$`
1	0	5	2	`atomato$`	0	5	2	`atomato$`
2	0	2	1	`ato$`	3	5	4	`matomato$`
3	0	1	0	`$`	3	4	3	`mato$`
4	**1**	1	-	-	3	**3**	-	-

Algorithm 5.3 Backward Search

```
1:  function backward-search(L, P)
2:      F ← sort_≤ L
3:      m ← |P|
4:      l ← 0
5:      r ← |L|
6:      while m > 0 ∧ l < r do
7:          v ← P[m − 1]
8:          k ← select F v 0
9:          l ← k + rank L v l
10:         r ← k + rank L v r
11:         m ← m − 1
12:     end while
13:     return {i ... < j}
14: end function
```

To see this in action, consider the example, mentioned earlier in Section 5.2.2, of finding all locations of the pattern $P = \texttt{ato}$ in the sequence $S = \texttt{romatomato\$}$ using its suffix array, shown in Figure 5.1. Table 5.1 demonstrates the two binary search algorithms, Algorithms 5.1 and 5.2, being used to find the lower and upper bounds of the interval, respectively. The resulting interval is $\{1 \ldots < 3\}$, which matches interval mentioned in Section 5.2.2 as required. Note that if the pattern P does not occur in the sequence S, then the interval returned will be $\{x \ldots < x\}$, where x is the first location in the suffix array of S, SA, such that $P <_{lex} S[SA[x] \ldots < SA[x] + |P|]$.

5.2.4 The Backward Search Algorithm

Recall from Chapter 4, the BWT has a one-to-one correspondence with the suffix array. Hence, using a suitable mapping, the BWT can function as a text index for indexed pattern search. *Backward search* [54] is an algorithm that uses such a mapping, that is, the LF-mapping described in Chapter 4, to search for an arbitrary pattern P in the sequence S, using only P and the BWT of S. With some auxiliary data structures computed from the BWT of S, the search time cost can be reduced to be $\mathcal{O}(m)$ for an m

length pattern [54]. Like the binary search approach on the suffix array, backward search returns the same interval over the suffix array, containing all the matching instances.

The backward search algorithm, shown in Algorithm 5.3, is similar to the inverse BWT, shown in (Algorithm 4.2). Both of these use the LF-mapping, with the backward search algorithm using it to construct the pattern P in reverse order from the BWT of S, attempting to find the first and last instances. The core idea behind the algorithm is that the i^{th} occurrence of a symbol v in the BWT of S both belongs to and is the first symbol of the i^{th} suffix starting with v in the list of sorted suffixes of S. Moreover, the i^{th} symbol in the BWT of S is also the symbol that occurs before the i^{th} suffix in the sorted suffix list. Both ideas are fundamental properties related to the BWT, detailed in Proposition 4.1, with the former being precisely what the LF-mapping computes.

Given a text S of length n and a pattern P of length m, the backward search algorithm begins by initializing a search interval from $l = 0$ up to but not including $r = n$. It then iterates over P in reverse order, that is starting at position $m - 1$, refining the interval by using the LF-mapping to look up where the first and last symbols in the interval that match the current symbol of P occur. If the interval becomes empty before P is completely processed, then P is not a subsequence of S and the algorithm terminates. Otherwise, the interval returned after P is processed contains all the occurrences of P in S. An example of the backward search algorithm in action is shown in Figure 5.2 and Table 5.2 for the sequence $S =$ `romatomato$` and pattern $P =$ `ato`.

Note that the interval is over the suffix array, and so, the backward search algorithm by itself can only determine the number of occurrences of P in S. Additional computations are required to determine their locations in S. The simplest approach is to invert the BWT of S starting at the locations indicated by the search range. This would compute the prefixes that occur directly before each occurrence of P in S, and the locations can then be determined by computing the lengths of these prefixes. Naturally, this approach would result in an additional worst-case time cost of $\mathcal{O}(n)$ for an n length text S and for each occurrence of P in S. This time cost can be reduced by using additional memory to store the prefix lengths at suitable intervals. The length of the intervals can be adjusted to balance time and space requirements.

5.3 Formalizing Substring Search

We formalize the substring search problem in Isabelle/HOL, using the in-built list type, with ordered element types, to represent both patterns and texts, and natural numbers

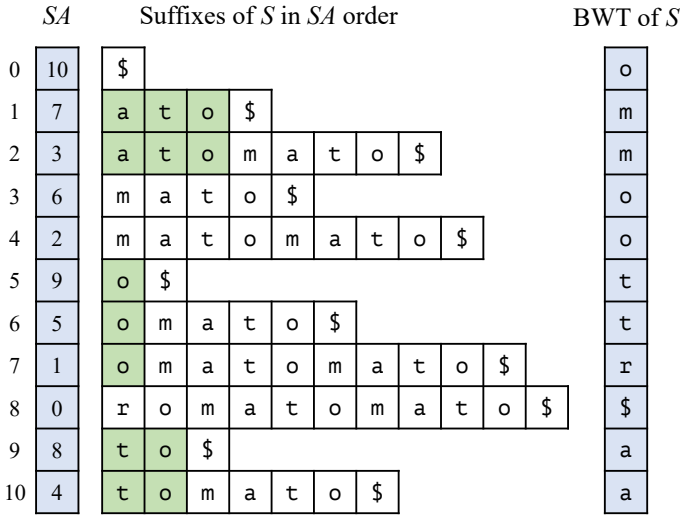


FIGURE 5.2: The suffix array (SA) and BWT of the sequence $S = \texttt{romatomato\$}$ with the pattern $P = \texttt{ato}$ being iteratively discovered in reverse order by the backward search algorithm. Green shading indicates the search range with the number of shaded columns indicating what iteration it belongs. For example, after the first iteration, the search range is $\{5 \ldots < 8\}$, and so, only the first column of the suffixes in that range are shaded.

TABLE 5.2: Backward search example for the pattern $P = \texttt{ato}$ and sequence $S = \texttt{romatomato\$}$, where m is the current position being looked in P and i and j is the start and end of the search interval.

m	$v = P[m-1]$	select $F\ v\ 0$	rank $L\ v\ l$	rank $L\ v\ r$	l	r
-	-	-	-	-	0	11
3	o	5	0	3	5	8
2	t	9	0	2	9	11
1	a	1	0	2	1	3
0	-	-	-	-	**1**	**3**

to represent locations in the sequences. The ordering between lists is classical lexicographical order ($<_{lex}$), and ordering between natural numbers is the standard numeric ordering ($<$).

The solution to a successful substring search query is defined as the set of all locations in a text at which the pattern occurs.

Definition 5.1 (Substring Search)**.** Given a text S and pattern P, the set of all locations in S where P occurs is defined as follows:

$$\mathsf{patsearch\text{-}set}\ S\ P = \{i \mid i < |S| \wedge S[i \ldots < i + |P|] = P\}\,,$$

where $S[x \ldots < y]$ is the subsequence of S starting at the location x and ending at the location y, exclusively, for some locations x and y. Note that if $y \geq |S|$, then $S[x \ldots < y] = S[x \ldots < |S|]$.

Note that in the above definition, if P is an empty sequence, then Definition 5.1 will return all the locations in S as possible matches, as the empty sequence occurs at every possible location in S.

5.4 Reformulation Using Suffix Arrays

The substring search problem, Definition 5.1, is reformulated in terms of the suffix array of the text S, by first defining an interval over a sorted list of sequences, which we denote as the *matching prefix interval*. A sequence lies within the matching prefix interval if and only if it begins with the pattern P. Instantiating the sorted list of sequences with the list of suffixes of S in sorted order, obtained from the suffix array of S, then results in an equivalent formulation of the substring search problem. Note that this part of the formalization and onwards builds on the results presented Chapter 3, specifically the axiomatic characterization of a suffix array construction function.

5.4.1 Matching Prefix Interval Over Sorted Lists

The matching prefix interval is defined using two functions. One function to compute the lower bound of the interval, which is the first sequence in a list of sequences that has a prefix of length up to $|P|$ that is greater than or equal to P. The other function to compute the upper bound, which is the first sequence in a list of sequences that has a prefix of length up to $|P|$ that is strictly greater than P.

The function that computes the lower bound of the matching prefix interval, defined below, constructs the set of locations belonging to sequences that begin with a prefix of length up to $|P|$ that is greater than or equal to $|P|$, and then takes the smallest of these. In the case that this set is empty, then the length of the list is returned instead as all sequences in the list must have prefixes that are strictly less than P.

Definition 5.2 (Pattern Start Location)**.** Given a list of sequences xs and pattern P. The location of the first sequence in xs with prefix greater than equal to P is:

$$\mathsf{patsearch\text{-}idx\text{-}start}\; xs\, P = \mathsf{Min}\left(\{i \mid i < |xs| \wedge P \leq_{lex} \mathsf{prefix}\,(xs[i])\;|P|\} \cup \{|xs|\}\right),$$

where $\mathsf{prefix}\, S\, x$ is the prefix of S of length x when $x < |S|$, and is simply S when $x \geq |S|$, for some sequence S and length x.

The location returned by patsearch-idx-start must point to the first sequence in the list of sequences xs that begins with the pattern P, if there exists a sequence in xs that begins

with P. In the case that no such sequence exists, then patsearch-idx-start returns the location of the first sequence that has a prefix of length up to P that is strictly greater than P. If all sequences in xs are less than P, then patsearch-idx-start returns the length of xs. Importantly, if the list of sequences xs is sorted, patsearch-idx-start serves as a partition point. Any sequence before patsearch-idx-start either has a prefix of length $|P|$ that is strictly less than P or the entire sequence is strictly less than P. In contrast, any sequence after patsearch-idx-start has a prefix of length up to $|P|$ that is greater than or equal to P.

The function that computes the upper bound of the matching prefix interval, shown below, constructs the set of all locations belonging to sequences that start with a prefix of length up to $|P|$ that is strictly greater than P, and takes the minimum In the case that the set is empty, then the length of the list is returned, as all sequences have prefixes less than or equal to P.

Definition 5.3 (Pattern End Location)**.** Given a list of sequences xs and pattern P. The location after the last sequence in xs with prefix less than equal to P is:

$$\text{patsearch-idx-end}\; xs\, P = \mathsf{Min}\left(\{i \mid i < |xs| \wedge P <_{lex} \text{prefix}\; (xs[i])\; |P|\} \cup \{|xs|\}\right).$$

The location returned by patsearch-idx-end points to the location after the last sequence in xs that begins with the pattern P. That is, the location of the first sequence with a prefix of length up to $|P|$ that is strictly greater than P. Note that if no sequence in xs has a prefix of length up to $|P|$ that is strictly greater than P, then patsearch-idx-end is $|xs|$. Importantly, if the list of sequences xs is sorted, patsearch-idx-end serves as a partition point. Any sequence before patsearch-idx-end either has a prefix of length $|P|$ that is less than or equal to P or the entire sequence is less than or equal to P. In contrast, any sequence after patsearch-idx-start has a prefix of length up to $|P|$ that is strictly greater than P.

When used in conjunction with a sorted list of sequences, Definitions 5.2 and 5.3 indicate the lower and upper bounds (l and r), respectively, of the interval containing sequences that begin with P. This is formalized below:

Theorem 5.4 (Matching Prefix Interval over Sorted Lists)**.** *Let xs be a list of sequences that are in sorted order, and P be a pattern. Then*

$$i < |xs| \wedge \text{prefix}\; (xs[i])\; |P| = P \longleftrightarrow$$
$$\text{patsearch-idx-start}\; xs\, P \leq i \wedge i < \text{patsearch-idx-end}\; xs\, P\,.$$

The proof of Theorem 5.4 follows from Definitions 5.2 and 5.3 and the fact that xs is sorted. From these, we then have that any sequence that occurs before any location j must be less than or equal to the sequence at j and any sequence occurring after j must be greater than or equal to the sequence at j. The core difficulty of the proof is handling the different proof obligations that arise from showing that $\mathsf{patsearch\text{-}idx\text{-}start}\, xs\, P \leq i$. Specifically, showing that the set of locations belonging to sequences starting with prefixes of length P that are greater than or equal to P is non-empty and that i is an inhabitant of that set. Similar obligations arise from showing that $i < \mathsf{patsearch\text{-}idx\text{-}end}\, xs\, P$. All the main proof obligations are separated into their own lemmas as this makes them shorter, clearer and more specific, and hence, easier to maintain and use.

We prove additional lemmas that equate the lower and upper bounds to the cardinality of sets. Specifically, we show that patsearch-idx-start equals to the cardinality of the set of locations that belong to sequences with prefixes that are strictly less than P, and that patsearch-idx-end equal to the cardinality of the set of locations that belong to sequence with prefixes that are less than or equal to P.

Lemma 5.5 (Pattern Start Location Cardinality Equivalence)**.** *Given a sorted list of sequences xs and pattern P, then*

$$\mathsf{patsearch\text{-}idx\text{-}start}\, xs\, P = \|\{k \mid k < |xs| \wedge \mathsf{prefix}\, (xs[k])\, |P| <_{lex} P\}\|\,,$$

where $\|A\|$ *is the cardinality of the set* A *for some set* A*.*

Lemma 5.6 (Pattern End Location Cardinality Equivalence)**.** *Given a sorted list of sequences xs and pattern P, then*

$$\mathsf{patsearch\text{-}idx\text{-}end}\, xs\, P = \|\{k \mid k < |xs| \wedge \mathsf{prefix}\, (xs[k])\, |P| \leq_{lex} P\}\|\,.$$

Lemmas 5.5 and 5.6 both follow from Theorem 5.4 (more precisely from lemmas about its components) where these are used to show that

$$\{0 \ldots < \mathsf{patsearch\text{-}idx\text{-}start}\, xs\, P\} = \{k \mid k < |xs| \wedge \mathsf{prefix}\, (xs[k])\, |P| <_{lex} P\}$$

and similarly for patsearch-idx-end. Lemmas 5.5 and 5.6 are particularly useful as we use these to show that binary search on suffix arrays and backward search on the BWT produce the same results as patsearch-idx-start and patsearch-idx-end. That is, they compute the start and end point of the matching prefix interval.

5.4.2 Substring Search and Suffix Array Interval Equivalence

By instantiating Definitions 5.2 and 5.3 with a list of sorted suffixes, obtainable from a suffix array, we then have the matching prefix interval over the suffix array. That is, the interval containing all suffixes that start with the pattern. We prove that the substring search problem and the suffix array interval problem are equivalent.

Theorem 5.7 (Substring Search and Suffix Array Interval Equivalence)**.** *Given a sequence S and a pattern P, then*

$$\mathsf{patsearch\text{-}set}\, S\, P = \{SA[i] \mid i \in \{\mathsf{patsearch\text{-}idx\text{-}start}\, xs\, P \ldots < \mathsf{patsearch\text{-}idx\text{-}end}\, xs\, P\}\},$$

where SA is the suffix array of S, $xs = \mathsf{map}\,(\lambda i.\, S[i\ldots])\, SA$ and $\{x \ldots < y\}$ is the set containing the numbers from x up to but not including y, for some locations x and y.

The above immediately follows from Theorem 5.4 and Definition 5.1.

With Theorem 5.7, we can now reformulate the substring search problem as simply computing patsearch-idx-start and patsearch-idx-end for $xs = \mathsf{map}\,(\lambda i.\, S[i\ldots])\, SA$, the list of sorted suffixes of S, and the pattern P.

5.5 Verified Binary Search with Suffix Arrays

The verification of binary search on suffix arrays, presented in this section, validates the canonical specification of substring search, detailed in Section 5.3, and demonstrates how the reformulation of the substring search as a matching prefix interval over the suffix array, detailed in Section 5.4, facilities the verification. In addition, the verification was instrumental in the development of Lemmas 5.5 and 5.6, as well as other smaller lemmas not mentioned in this chapter, that streamline the verification of binary search for indexed pattern search and the backwards search algorithm, detailed in Section 5.6.

To verify that binary search can indeed be used to search for patterns in a sequence with its suffix array, we first define Isabelle/HOL embeddings of the two binary search algorithms shown in Algorithms 5.1 and 5.2. These, shown below, have the same structure as their pseudocode counterparts, but have been generalized so that they also take additional values that are used to initialize the search range. This means that the search can be restricted to a designated search range over the list.

Definition 5.8 (Binary Search (leftmost))**.** Given a sorted list of values xs with respect to some comparison $<_\alpha$, a value v, locations l and r, with $0 \leq l$, $0 \leq r$, $l \leq |xs|$ and

$r \leq |xs|$, the Isabelle/HOL embedding of binary search that finds the leftmost instance is:

$$
\begin{aligned}
&l < r \longrightarrow \mathsf{binary\text{-}search\text{-}l}\ xs\ v\ l\ r = \\
&\quad (\text{let } m = (l + r)\ \mathsf{div}\ 2 \\
&\qquad \text{in } (\text{if } xs[m] <_\alpha v \text{ then } \mathsf{binary\text{-}search\text{-}l}\ xs\ v\ (\mathsf{Suc}\ m)\ r \text{ else } \mathsf{binary\text{-}search\text{-}l}\ xs\ v\ l\ m) \\
&\neg(l < r) \longrightarrow \mathsf{binary\text{-}search\text{-}l}\ xs\ v\ l\ r = l\,.
\end{aligned}
$$

Definition 5.9 (Binary Search (rightmost))**.** Given a sorted list of values xs with respect to some comparison $<_\alpha$, a value v, locations l and r, with $0 \leq l$, $0 \leq r$, $l \leq |xs|$ and $r \leq |xs|$, the Isabelle/HOL embedding of binary search that finds the rightmost instance is:

$$
\begin{aligned}
&l < r \longrightarrow \mathsf{binary\text{-}search\text{-}r}\ xs\ v\ l\ r = \\
&\quad (\text{let } m = (l + r)\ \mathsf{div}\ 2 \\
&\qquad \text{in } (\text{if } v <_\alpha xs[m] \text{ then } \mathsf{binary\text{-}search\text{-}r}\ xs\ v\ l\ m \text{ else } \mathsf{binary\text{-}search\text{-}r}\ xs\ v\ (\mathsf{Suc}\ m)\ r) \\
&\neg(l < r) \longrightarrow \mathsf{binary\text{-}search\text{-}r}\ xs\ v\ l\ r = r\,.
\end{aligned}
$$

With the Isabelle/HOL embeddings of the two versions of binary search defined, we show that these are correct. For binary-search-l, we show that searching for a value v returns a location k within the initial search range (l, r), where every element starting from l and before k is strictly less than v and every element starting from k and up to r is greater than or equal to v.

Theorem 5.10 (General Binary Search Correctness (leftmost))**.** *Given a sorted list xs with respect to some comparison* $<_\alpha$ *and locations* l *and* r*, such that* $0 \leq l$*,* $l \leq r$ *and* $r \leq |xs|$

$$
\begin{aligned}
&(\forall i.\, (l \leq i \wedge i < \mathsf{binary\text{-}search\text{-}l}\ xs\ v\ l\ r) \longrightarrow xs[i] <_\alpha v) \wedge \\
&(\forall i.\, (\mathsf{binary\text{-}search\text{-}l}\ xs\ v\ l\ r \leq i \wedge i < r) \longrightarrow v \leq_\alpha xs[i])\,.
\end{aligned}
$$

Similarly for binary-search-r, we show that it returns a location k within (l, r), where every element starting from l and before k is less than or equal to v and every element starting from k and up to r is strictly greater than v.

Theorem 5.11 (General Binary Search Correctness (rightmost))**.** *Given a sorted list xs with respect to some comparison* $<_\alpha$ *and locations* l *and* r*, such that* $0 \leq l$*,* $l \leq r$ *and*

$r \leq |xs|$

$$(\forall i.\, (l \leq i \wedge i < \mathsf{binary\text{-}search\text{-}r}\ xs\ v\ l\ r) \longrightarrow xs[i] \leq_\alpha v) \wedge$$
$$(\forall i.\, (\mathsf{binary\text{-}search\text{-}r}\ xs\ v\ l\ r \leq i \wedge i < r) \longrightarrow v <_\alpha xs[i])\,.$$

The proofs of the Theorems 5.10 and 5.11 are constructed by splitting the conjunction and proving each part separately by induction on the list *xs*.

To show that binary search with suffix arrays implements indexed pattern search, we prove the following correctness theorem.

Theorem 5.12 (Indexed Pattern Search using Binary Search and the Suffix Array)**.** *Given a sequence S, its suffix array SA and pattern P*

$$\mathsf{patsearch\text{-}set}\ S\ P =$$
$$\{SA[i] \mid i \in \{\mathsf{binary\text{-}search\text{-}l}\ xs\ P\ 0\ |xs| \ldots < \mathsf{binary\text{-}search\text{-}r}\ xs\ P\ 0\ |xs|\}\}\,,$$

where $xs = \mathsf{map}\,(\lambda i.\, \mathsf{prefix}\,(S[i\ldots])\,|P|)\,SA$, that is, the list of prefixes of the sorted suffixes.

Instantiating l and r with the 0 and $|xs|$ respectively in Theorems 5.10 and 5.11 results the traditional correctness theorems for binary search. Moreover, by instantiating *xs* with the list of prefixes of the sorted suffix, obtained from the suffix array, and utilizing Lemmas 5.5 and 5.6 we then have that binary-search-l computes patsearch-idx-start and binary-search-r computes patsearch-idx-end. Hence, with Theorems 5.4 and 5.7, we then have that binary search with the suffix array finds all instances of a pattern in a text, proving Theorem 5.12.

5.6 Verified Backward Search

Building on Chapter 4, using the BWT Isabelle/HOL embeddings and the formalized properties about the BWT and counting, we verify that backward search with the BWT provides another implementation of indexed pattern search. First, we define the Isabelle/HOL embedding of backward search, which is defined in three stages: a single symbol lookup over the BWT S (backward-lookup); an iterative application of the single symbol lookups with early termination that given the reverse of the pattern P finds the matching prefix interval over the suffix array of S (search-backward); and a wrapper that reverses the desired pattern, initializes the search range and applies search-backward

(backward-search). Note that in the following definitions, intervals of the form $[x, y)$ are modelled as tuples.

Definition 5.13 (Backward Search)**.** Let S be a sequence, $L = \mathsf{bwt}\, S$, and P be a pattern. Then following finds the matching prefix interval over the suffix array of S:

$$\mathsf{backward\text{-}lookup}\ L\ v\ i = (\mathsf{select}\ (\mathsf{sort}_{\leq}\ L)\ v\ 0) + \mathsf{rank}\ L\ v\ i$$

$$\mathsf{search\text{-}backward}\ L\ []\ (l, r) = (l, r)\ |$$
$$\mathsf{search\text{-}backward}\ L\ (p\ \#\ ps)\ (l, r) =$$
$$(\text{if}\ l < r\ \text{then}\ \mathsf{search\text{-}backward}\ L\ ps\ (\mathsf{backward\text{-}lookup}\ L\ p\ l, \mathsf{backward\text{-}lookup}\ L\ p\ r)$$
$$\text{else}\ (l, r))$$

$$\mathsf{backward\text{-}search}\ L\ P = \mathsf{search\text{-}backward}\ L\ (\mathsf{rev}\ P)\ (0, |L|)$$

In addition to the above, we also define two other embeddings to aid in the verification. The first is simply the repeated application of backward-lookup for a sequence of symbols (backward-lookups). The second is search-backward but without the early termination (search-backward′).

Definition 5.14 (Backward Search Intermediaries)**.** Let S be a sequence, and $L = \mathsf{bwt}\, S$, then

$$\mathsf{backward\text{-}lookups}\ L\ []\ i = i\ |$$
$$\mathsf{backward\text{-}lookups}\ L\ (p\ \#\ ps)\ i = \mathsf{backward\text{-}lookups}\ L\ ps\ (\mathsf{backward\text{-}lookup}\ L\ p\ i)$$

$$\mathsf{search\text{-}backward'}\ L\ []\ (l, r) = (l, r)$$
$$\mathsf{search\text{-}backward'}\ L\ (p\ \#\ ps)\ (l, r) =$$
$$\mathsf{search\text{-}backward'}\ L\ ps\ (\mathsf{backward\text{-}lookup}\ L\ p\ l, \mathsf{backward\text{-}lookup}\ L\ p\ r)$$

With these embeddings we then show backward-search is correct, as a chain of stages. First we show that backward-lookup finds the first and last locations of a sequence $p\ \#\ ps$ if given the first and last locations of ps in the suffix array of S, respectively. Using these results, we show that backward-lookups finds the first and last locations of a reversed pattern, depending on the initial search location. Then we show that the matching prefix interval for a reversed pattern is computed by search-backward′, and then show that search-backward is a refinement. Finally, we show that backward-search is correct

using this result. Of these stages, the correctness of backward-lookup was the most difficult, while the rest had simpler proofs that followed from induction on the pattern and term rewriting.

For backward-lookup, we prove three key theorems: one for the case where the pattern is not a subsequence, and one each for finding the lower and upper bounds of the matching prefix interval. In the former, we show that if a pattern ps is a subsequence of the sequence S and $p \# ps$ is not a subsequence, then backward-lookup returns a lower bound that is greater than or equal to the upper bound, producing an empty interval.

Theorem 5.15 (Backward Lookup Failure Case)**.** *Let* $L = \mathsf{bwt}\ S$ *for a valid sequence* S*. For a sound pattern* $p \# ps$*, that is, the pattern is either a valid sequence or does not contain the sentinel* $\bot$*, assume that* $\mathsf{patsearch\text{-}set}\ S\ ps \neq \emptyset$ *and* $\mathsf{patsearch\text{-}set}\ S\ (p \# ps) = \emptyset$*. Then,*

$$\mathsf{backward\text{-}lookup}\ L\ p\ r \leq \mathsf{backward\text{-}lookup}\ L\ p\ l\,,$$

where SA *is the suffix array of* S*,* $xs = \mathsf{map}\ (\lambda i.\ S[i\ldots])\ SA$ *is the sorted list of suffixes of* S*, and* $l = \mathsf{patsearch\text{-}idx\text{-}start}\ xs\ ps$ *and* $r = \mathsf{patsearch\text{-}idx\text{-}end}\ xs\ ps$ *are the lower and upper bounds for the matching prefix interval over* SA *for the pattern* ps*.*

The core idea of the proof is to show that the rank for p at l and r are equal, that is

$$\mathsf{rank}\ L\ p\ l = \mathsf{rank}\ L\ p\ r\,.$$

With this, it then follows from Definition 5.13 that

$$\mathsf{backward\text{-}lookup}\ L\ p\ l = \mathsf{backward\text{-}lookup}\ L\ p\ r\,,$$

proving Theorem 5.15. To show that the ranks are the same, the proof proceeds by taking cases on ps, that is, handling when it is empty and when it is not. When ps is empty, the assumptions imply that p cannot be in L, resulting in $\mathsf{rank}\ L\ p\ k = 0$ for all k, and hence, showing that the rank of p at l and r are equal. When ps is not empty, the proof shows that the ranks must be equal, otherwise, Theorem 5.7 implies that $p \# ps$ would be a subsequence of the original sequence S, which would be a contradiction.

For the case where the pattern $p \# ps$ is a subsequence, we show that if the locations of the first and last occurrences of ps in the suffix array are known, then backward-lookup finds the locations of the first and last occurrences of $p \# ps$. The two theorems are presented below.

Theorem 5.16 (Backward Lookup Find Search Range Start)**.** *Let* $L = \mathsf{bwt}\ S$ *for a valid sequence* S*. For a sound pattern* $p \# ps$*, that is, the pattern is either a valid*

sequence or does not contain the sentinel $\perp$*, assume that* patsearch-set $S\ ps \neq \emptyset$ *and* patsearch-set $S\ (p \# ps) \neq \emptyset$*. Then,*

$$\text{backward-lookup}\ L\ p\ l = \text{patsearch-idx-start}\ xs\ (p \# ps)\,,$$

where SA is the suffix array of S*,* $xs = \text{map}\ (\lambda i.\ S[i\ldots])\ SA$ *is the list of sorted suffixes of* S*, and* $l = \text{patsearch-idx-start}\ xs\ ps$ *is the lower bound of the matching prefix interval over SA for the pattern ps.*

Theorem 5.17 (Backward Lookup Find Search Range End)**.** *Let* $L = \text{bwt}\ S$ *for a valid sequence* S*. For a sound pattern* $p \# ps$*, that is, the pattern is either a valid sequence or does not contain the sentinel* $\perp$*, assume that* patsearch-set $S\ ps \neq \emptyset$ *and* patsearch-set $S\ (p \# ps) \neq \emptyset$*. Then,*

$$\text{backward-lookup}\ L\ p\ r = \text{patsearch-idx-start}\ xs\ (p \# ps)\,,$$

where SA is the suffix array of S*,* $xs = \text{map}\ (\lambda i.\ S[i\ldots])\ SA$ *is the list of sorted suffixes of* S*, and* $r = \text{patsearch-idx-end}\ xs\ ps$ *is the upper bound of the matching prefix interval over SA for the pattern ps.*

Theorems 5.16 and 5.17 have largely the same proof structure with the differences arising only due to the former using patsearch-idx-start and the latter patsearch-idx-end. The core argument of both proofs is that from Theorem 5.7, we know that every suffix in the suffix array that lies in the interval $[l, r)$ begins with ps and vice versa. Moreover, the predecessors of these suffixes that start with p must have the same ordering. Furthermore, the interval $[l, r)$ over L, which is the BWT of the sequence S, corresponds to the first symbol of these predecessor suffixes. And so, the first and last p in L between $\{l \ldots < r\}$ belong to the first and last suffix in the suffix array starting with $p \# ps$, respectively, which follows from Theorem 4.13. We then show that these locations are patsearch-idx-start and patsearch-idx-end for $p \# ps$ using Lemmas 5.5 and 5.6.

With Theorems 5.7 and 5.15 to 5.17 we can then show by induction that the matching prefix interval for a sequence S and pattern P can be computed by iteratively applying backward-lookup to a sequence via backward-lookups, when given the initial search range of $\{0 \ldots < |P|\}$. Note that we do not simply prove that backward-lookups computes patsearch-idx-start or patsearch-idx-end as this is not true in the case where P is not a subsequence of S. In that case, backward-lookups will produce an empty range, as stated by Theorem 5.15.

Theorem 5.18 (Backward Lookups Correctness)**.** *Given a valid sequence S and pattern P, let $SA = \mathsf{sa}\ S$ and $L = \mathsf{bwt}\ S$. Then*

$$\mathsf{patsearch\text{-}set}\ S\ P = \{SA[i] \mid i \in \{x \ldots < y\}\}\,,$$

where $x = \mathsf{backward\text{-}lookups}\ L\ (\mathsf{rev}\ P)\ 0$ and $y = \mathsf{backward\text{-}lookups}\ L\ (\mathsf{rev}\ P)\ |L|$.

Having proved that backward-lookups computes the matching prefix interval, we then show by induction on the reversed pattern that search-backward′ computes the same interval as backward-lookups.

Theorem 5.19 (Search Backward Correctness)**.** *Given sequence xs and ys and locations l and r, let $x = \mathsf{backward\text{-}lookups}\ xs\ ys\ l$ and $y = \mathsf{backward\text{-}lookups}\ xs\ ys\ r$. Then*

$$\mathsf{search\text{-}backward'}\ xs\ ys\ (l, r) = (x, y)\,.$$

To introduce the early termination behavior, shown in Algorithm 5.3, search-backward′ is refined to search-backward. Note that this is a refinement as the functions do not produce the same lower and upper bounds, when the reversed pattern is not a substring of the text. However, since these intervals are empty in this case, the exact values of the lower and upper bounds would not matter.

Theorem 5.20 (Search Backward With Early Termination Correctness)**.** *Given sequence xs and ys and locations l and r, then*

$$\{w \ldots < x\} = \{y \ldots < z\}\,,$$

where $[w, x)$ is the matching prefix interval computed by search-backward′ *$xs\ ys\ (l, r)$, and $[y, z)$ is the matching prefix interval computed by* search-backward *$xs\ ys\ (l, r)$.*

To prove the above, we induct on *ys*. The base case follows from simple term rewriting. The inductive case is solved by case analysis on the initial search range $[l, r)$, that is we handle when $l < r$ and when $\neg(l < r)$. The first case follows directly from the correctness theorem for search-backward′ (Theorem 5.19), and term rewriting. The second case requires that both search-backward and search-backward′ produce empty ranges. We show this by proving that rank is monotonic, and due to this, backward-lookup and backward-lookups must also be monotonic, resulting in empty search ranges.

Finally, we prove that for a text S and pattern P, backward-search implements indexed pattern search.

Theorem 5.21 (Backward Search Correctness)**.** *Given a valid sequence S and pattern P, let $SA = \mathsf{sa}\ S$, $L = \mathsf{bwt}\ S$, and $(x, y) = \mathsf{backward\text{-}search}\ L\ P$. Then*

$$\mathsf{patsearch\text{-}set}\ S\ P = \{SA[i] \mid i \in \{x \ldots < y\}\}\,.$$

The proof of Theorem 5.21 follows from Theorem 5.18, instantiating l and r with 0 and $|S|$, and using Theorems 5.19 and 5.20 and Definition 5.13 to unfold the definition of backward-search and simplify it to backward-lookups.

5.7 Limitations and Future Work

Like the verifications of the SA-IS algorithm and the BWT, presented in Chapters 3 and 4, our verification of the binary search algorithm using suffix arrays and the backward search algorithm using the BWT is designed to be a mathematical formalization not an efficient implementation. Hence, similar approaches, described in Section 3.7.3, can be used to produced efficient implementations.

In this formalization, we demonstrated how an algorithm like binary search can use the suffix array to find patterns. While this approach is simple, it does have a running time of $\mathcal{O}(m \log n)$ for a text of length n and pattern of length m. Other approaches exist that reduce this cost to $\mathcal{O}(m)$ by augmenting the suffix array with additional data structures. Verifying the construction of such data structures [87] and search algorithms [1] is a potential extension.

The backward search algorithm that we verified also has the same limitation as the BWT formalization as it uses naïvely represented rank and select function. Again, this could be resolved by providing an efficient pre-computation and verifying that it is a refinement of our straightforward implementation, and also by using more advanced rank-and-select data structures. Besides this, the backward search algorithm is also limited by the fact that it returns an interval over the suffix array. Earlier in Section 5.2.4, we stated that there exist approaches, besides the using the inverse BWT, that can be used to obtain the exact location of a match in terms of the original text [54]. Formally verifying such approaches would be a possible extension.

5.8 Summary

We have presented the formal verification of two indexed pattern search algorithms: binary search with suffix arrays, and backward search with the BWT. The formalization

consists of a specification of the substring search problem, an equivalent formulation of the specification in terms of search intervals over suffix arrays, and mechanized correctness proofs of binary search and backward search in Isabelle/HOL. We provide a basis with our formal specification of substring search that enables future verification of other pattern search algorithms, in particular algorithms that utilize suffix arrays and the BWT as text indexes. Moreover, our formally verified Isabelle/HOL implementations of binary search and backward search form the essential foundation for the verification of efficient executable implementation.

In Chapters 3 and 4, we presented the formal verification of the SA-IS algorithm, a linear-time algorithm for suffix array construction, and the construction and inverse of the BWT, respectively. Both of these addressed the formal verification of text index construction, a key facet of indexed pattern search. This chapter presented the formal verification of two algorithms, binary search and backwards search, that use the suffix array and the BWT as text indexes, and addressed the verified search component of indexed pattern search. This completes our investigation on the formal verification of indexed pattern search.

Chapter 6

Composing Verification of Foreign Functions with Cogent

While Chapters 3 to 5 focused on the formal verification of indexed pattern search — the primary contribution of this thesis — this chapter explores a distinct but important challenge in formal verification: verifying the correctness and safety of multi-language programs. We frame our investigation around *Cogent* [148]. Cogent is a restricted functional language designed to reduce the cost of developing verified systems code. Due to its sometimes-onerous restrictions, such as the lack of support for recursion and its strict uniqueness type system, Cogent provides an escape hatch in the form of a foreign function interface (FFI) to C code. This poses a problem when verifying Cogent programs, as imported C components do not enjoy the same level of static guarantees that Cogent does. Previous verification efforts for file systems implemented in Cogent [9] merely assumed that their C components were correct and that they preserved the invariants of Cogent's type system.

In this chapter[1], we instead prove Cogent's FFI proof obligations. We demonstrate how they smoothly compose with existing Cogent theorems, and result in a correctness theorem of the overall Cogent-C system. The Cogent FFI constraints ensure that key invariants of Cogent's type system are maintained even when calling C code. We verify reusable higher-order and polymorphic functions, including a generic loop combinator and array iterators, and demonstrate their application to several examples, including binary search and the BilbyFs file system [9]. We demonstrate the feasibility of verification of mixed Cogent-C systems, and provide some insight into verification of software comprised of code in multiple languages with differing levels of static guarantees.

[1]This chapter is based on the paper: L. Cheung, L. O'Connor, and C. Rizkallah. Overcoming Restraint: Composing Verification of Foreign Functions with Cogent. In *Proc. Certified Programs and Proofs*, pages 13–26. ACM, 2022. doi: 10.1145/3497775.3503686.

6.1 Introduction

Cogent [147] is a restricted purely functional language with a *certifying compiler* [147, 165] designed to ease the challenge of creating verified operating systems components [9]. It has a foreign function interface (FFI) that enables implementing parts of a system in C. Cogent's main restrictions are the purposeful lack of recursion or loops, which ensures totality, and its *uniqueness type system*, which enforces a *uniqueness invariant* that, among other benefits, guarantees memory safety.

Even in the restricted target domains of Cogent, real programs need to be able to support iteration, primarily over data structures such as buffers. This is achieved through Cogent's principled FFI: engineers provide data structures and their associated operations, including iterators, in a special dialect of C, and import them into Cogent, including in formal reasoning. This special C code, called *template C*, can refer to Cogent data types and functions, and is translated into standard C along with the Cogent program by the Cogent compiler. As long as the C components respect Cogent's foreign function interface — that is, are correct, memory-safe and respect the uniqueness invariant — the Cogent framework guarantees that correctness properties proved on high-level specs also apply to the compiler output.

Two real-world Linux file systems have been implemented in Cogent — `ext2` and BilbyFs — and key operations of BilbyFs have been verified [9]. This prior work demonstrates Cogent's suitability as a systems programming language and as a verification framework that reduces the cost of verification. The implementations of these file systems import an external C library of data structures, which include fixed-length arrays and iterators for implementing loops, as well as Cogent stubs for accessing a range of the Linux kernel's internal APIs. This library was carefully designed to ensure compatibility with Cogent's FFI constraints, but was previously left unverified. That is, only the Cogent parts of these file system operations were proven correct, and statements of the underlying C correctness and FFI constraints defining Cogent-C interoperability were left as assumptions.

To fully verify a system written in Cogent and C, one needs to provide manually-written abstractions of the C parts, and manually prove refinement through Cogent's FFI. The effort required for this manual verification remains substantial, but the reusability of these libraries allows this cost to be amortized across different systems.

In this work, we eliminate several of these assumptions by verifying the array implementation and key iterators used in the BilbyFs file system (Section 6.3), and discharging the conditions imposed by Cogent's FFI (Section 6.5). We also verify a generic-loop combinator (Section 6.4) and its application to binary search. This demonstrates that it

is possible and relatively straightforward for the C components of a real-world Cogent-C system to satisfy Cogent's FFI conditions. The compiler-generated refinement and preservation proofs compose with manual C proofs at each intermediate level up to Cogent's generated shallow embedding, resulting in verified Cogent-C programs.

As arrays and loops are extremely common in Cogent programming, our proofs are highly reusable for verification of any future Cogent-C system. In addition, our proofs connect C arrays to Isabelle/HOL lists and loops to an Isabelle/HOL repeat function that allows early termination. These proofs are reusable even beyond the context of Cogent in verification of C code, and were in fact used in the verification the SA-IS algorithm, detailed in Chapter 3. Our code and proofs are online[2].

Similar to many high-level languages, Cogent's foreign function interface connects a high-level language with strong static guarantees to an unsafe imperative language. This work provides our community with a case-study demonstrating how to equip such a foreign function interface with proof requirements such that those static guarantees are maintained for the overall system. In particular, our work supports, and is well-described by, Ahmed's [7] claim that:

"Compositional compiler correctness is, in essence, a language interoperability problem: for viable solutions in the long term, high-level languages must be equipped with principled foreign-function interfaces that specify safe interoperability between high-level and low-level components, and between more precisely and less precisely typed code."

Our approach to language interoperability does not rely on how the refinement theorems of the languages are obtained nor on whether they are manually or automatically proven. As such, we believe that this approach is likely reusable in the context of verified compilers.

6.2 The Cogent Language

The Cogent language [146–148] was originally designed for the implementation of systems components such as file systems [9]. It is a purely functional language, but it is compiled into efficient C code suitable for systems programming[3].

The Cogent compiler produces three artifacts: C code, a shallow embedding of the Cogent code in Isabelle/HOL [137], and a formal refinement proof relating the two [147, 165]. The refinement theorem and proof rely on several intermediate embeddings also generated by

[2] https://github.com/au-ts/cogent/releases/tag/cpp2022

[3] While Cogent is ideally suited for applications that involve minimal sharing and where efficiency matters, it is not specific to the systems domain.

the Cogent compiler, some related through language level proofs, and others through translation validation phases (Section 6.2.5). The compiler certificate guarantees that correctness theorems proven on top of the shallow embedding also hold for the generated C, which eases verification, and serves as the basis for further functional correctness proofs.

A key part of the compiler certificate depends on Cogent's *uniqueness type system*, which enforces that each mutable heap object has exactly one active pointer in scope at any point in time. This *uniqueness invariant* allows modelling imperative computations as pure functions: the allocations and repeated copying commonly found in functional programming can be replaced with destructive updates, and the need for garbage collection is eliminated, resulting in predictable and efficient code.

Well-typed Cogent programs have two interpretations: a purely functional *value semantics*, which has no notion of a heap and treats all objects as immutable values, and an imperative *update semantics*, describing the destructive mutation of heap objects. These two semantic interpretations correspond (Section 6.2.5), meaning that any correctness proofs about the value semantics also apply to the update semantics. As we will see, this correspondence further guarantees that well-typed Cogent programs are memory safe.

6.2.1 Language Design and Examples

Cogent has unit, numeric, and Boolean primitive types, as well as functions, sum types (variants) and product types (tuples and records). Users can declare additional *abstract* types in Cogent, and define them externally (in C). Abstract and record types may be *boxed*, that is, stored on the heap, in which case they are mutable and subject to the uniqueness restrictions of Cogent's type system. Cogent does not support closures, so partial application via currying is not common. Thus, functions of multiple arguments take a tuple or record of those arguments.

Figure 6.1 includes an example of Cogent signatures for an externally-defined array library interface, where array indices and length are unsigned 32-bit integers (`U32`).

Like ML, Cogent supports *parametric polymorphism* for top-level functions, and implements it via monomorphisation. For imported code, the compiler generates specialized C implementations from a polymorphic template, one for each concrete instantiation used in the Cogent code. Variables of polymorphic type are by default *linear*, which means they must be used exactly once [182]. Thus, a polymorphic type variable may be instantiated to any type, including types that contain pointers, while preserving the uniqueness invariant.

```
type Array a
length : (Array a)! → U32
get : ((Array a)!, U32, a!) → a!
put : (Array a, U32, a) → Array a

map : (a → a, Array a) → Array a
fold : ((a!, b) → b, b, (Array a)!) → b
```

```
add : (U32, U32) → U32
add (x, y) = x + y
sum : (Array U32) ! → U32
sum arr = fold (add, 0, arr)
```

FIGURE 6.1: A Cogent sum program that makes use of an abstract array type and operations.

As mentioned, types and functions provided in external C code are called *abstract* in Cogent. The Cogent compiler has infrastructure for linking the C implementations and the compiled Cogent code. Users write *template C* code, that can include embedded Cogent types and expressions via quasi-quotation, and the Cogent compiler translates the template C into ordinary C that it links with the C code generated from Cogent. To represent containers, abstract types may be given type parameters. These parameterized types, as well as polymorphic functions, are translated into a family of automatically generated C functions and types; one for each concrete type used in the Cogent program.

Though the *Array* type interface may appear purely functional, Cogent assumes that all abstract types are by default *linear*, ensuring that the uniqueness invariant applies to variables of type *Array*. Therefore, any implementation of the abstract put function is free to destructively update the provided array, without contradicting the purely functional semantics of Cogent.

When functions only need to read from a data structure, uniqueness types can complicate a program unnecessarily by requiring a programmer to thread through all state, even states that are unchanged. The !-operator helps to avoid this by converting *linear*, *writable* types to *read-only* types that can be freely shared or discarded. This is analogous to a *borrow* in Rust. The length and get[4] functions, presented in Figure 6.1, can *read* from the given array, but may not *write* to it.

As Cogent does not support recursion, iteration is expressed using abstract *higher-order functions*, providing basic traversal combinators such as map and fold for abstract types, as can be seen in Figure 6.1. Note that map is passed a function of type $a \to a$. As such, map is able to destructively overwrite the array with the result of the function applied to each element.

[4]Note that the get function requires an additional read-only argument ($a!$) as this is the default value that is returned when the function is given an index that is out of bounds.

```
length : (Array a)! → U32
mapAcc : ((a, b, c!) → (a, b), b, Array a)!, U32, U32, c!) → (Array a, b)
fold : ((a, b, c!) → b, b, (Array a)!, U32, U32, c!)
```

```
add : (U32, U32, ()) → U32
add (x, y, z) = x + y
sum : (Array U32)! → U32
sum arr = fold (add, 0, arr, 0, length arr, ())
```

FIGURE 6.2: The sum function now written using the interface from Cogent's C library.

While Cogent supports higher-order functions, it does not support nested lambda abstractions or closures, as these can require allocation if they capture variables. Thus, to invoke the map or fold functions, a separate top-level function must be defined, such as the function add in our example.

The array interface from Cogent's C library used in the implementation of the Cogent file systems, part of which is given in Figure 6.2, is more complex than that of Figure 6.1: The higher-order functions are given two additional index parameters to operate over only a subsection of an array; and instead of relying on closure captures, which are not available in Cogent, we provide alternative iterator functions, which carry an additional *observer* read-only input (of type $c!$). In addition, the function mapAcc is a generalized version of map, which allows threading an accumulating argument through the map function, similar to the same function in Haskell. We present the verification of these iterators in Section 6.3.

6.2.2 Dynamic Semantics

Cogent's big-step *value* semantics is defined through the judgment $V \vdash e \Downarrow_{\mathsf{v}} v$. This judgment states that the expression e under environment V evaluates to the value v. The environment V maps variables to their values. The imperative *update* semantics, which additionally may manipulate a mutable storeμ, is defined through the judgment $U \vdash e|\mu \Downarrow_{\mathsf{u}} u|\mu'$. This states that, starting with an initial store μ the expression e will evaluate under the environment U to a final store μ' and a result value u. Unlike the values in the value semantics, values in the update semantics may be *pointers* to locations in the store.

Both of these semantics are further parameterized by additional *functions* and types of *values* that are provided externally to Cogent, to model the semantics of abstract functions and types. More formally, the value semantics is parameterized by a function $\xi_{\mathsf{v}} : f_{id} \to (v \to v)$ and the update semantics by a function $\xi_{\mathsf{u}} : f_{id} \to (\mu \times u \to \mu \times u)$. Both of these are essentially an environment providing a pure HOL function on Cogent

values (and stores, for the update semantics) for each abstract function. The definitions of values in the value semantics (v) and update semantics (u) are also extended with parameters a_{v} and a_{u} respectively which represent values of abstract types.

Along with C code for all abstract functions and types, the user must also manually provide Isabelle/HOL abstractions of this C code to instantiate these environments.

To verify Cogent systems, three main proof obligations must be discharged: *type preservation*, which ensures the uniqueness invariant is maintained; the *frame requirements*, which ensures that memory safety is maintained; and *refinement*, which ensures that functional correctness theorems are preserved down to the C level via the provided abstractions. Cogent proves all three of these requirements automatically for Cogent code: both type preservation and the frame requirements are simple corollaries of the key semantic correspondence theorem (Theorem 6.5) that makes up part of the Cogent refinement chain. For linked C code, however, the user must discharge these obligations manually. We discuss our verification of these requirements for C code in Section 6.3.

6.2.3 Type System and Type Preservation

Cogent's *static semantics* are defined through a standard typing judgment $A; \Gamma \vdash e : \tau$, which states that e has type τ under context Γ, with an additional context A that tracks assumptions about the linearity of type variables in τ. To accommodate abstract types, we allow the type system to be extended with types $\mathtt{A}\ \overline{\tau}$ and $(\mathtt{A}\ \overline{\tau})!$, referring to linear abstract types and read-only abstract types respectively, where $\mathtt{A}$ is a type constructor parameterized by zero or more type parameters $\overline{\tau}$.

Dynamic values in the value semantics are typed by the simple judgment $v : \tau$, whereas update semantics values must be typed with the store μ to type the parts of the value that are stored there. Update semantics values are typed by the judgment $u|\mu : \tau\ [r * w]$, which additionally includes the *heap footprint*, consisting of the sets of read-only (r) and writable (w) pointers the value can contain. We use the same notation for value typing on environments.

This heap footprint annotation is crucial to ensuring that Cogent maintains its uniqueness invariant, as it places constraints on the footprints of subcomponents of a value to rule out aliasing of live pointers. Thus, our theorem of type preservation across evaluation in the update semantics also shows that this invariant is preserved. More details on these constraints are discussed in earlier work [147, 148]. When the heap footprints are not relevant and merely existentially quantified, we will omit them:

$$u|\mu : \tau \ \equiv\ \exists r\ w.\ u|\mu : \tau\ [r * w]\,.$$

Theorem 6.1 (Type Preservation)**.**
Type preservation for the update semantics:

$$A; \Gamma \vdash e : \tau \wedge U|\mu : \Gamma \wedge U \vdash e|\mu \Downarrow_{\boxed{\text{u}}} u|\mu' \longrightarrow u|\mu' : \tau\,.$$

Type preservation for the value semantics:

$$A; \Gamma \vdash e : \tau \wedge V : \Gamma \wedge V \vdash e \Downarrow_{\boxed{\text{v}}} v \longrightarrow v : \tau\,.$$

This states that the value typing relation for either semantics is preserved across the evaluation relation for well-typed expressions. This is because the value typing relations of the update and value semantics are later combined into one refinement relation, which is shown to be preserved across evaluation in Theorem 6.5, type preservation is obtained by simply erasing one of the semantics from Theorem 6.5.

Since the set of types is extensible, the value-typing relation for both semantics must also be extensible. To ensure that the user's extensions to the value-typing relation do not violate the uniqueness invariant, Cogent places a number of proof obligations on abstract types that must be discharged by the user. These requirements are outlined in Section 6.2.7.

6.2.4 Frame Requirements

In addition to type preservation, which ensures that each Cogent value is well-formed and does not contain internal aliasing, we must also show that the mutable store μ is in good order throughout evaluation — memory should not be leaked, and programs should not write to memory to which they have no access. These *memory safety* requirements are summed up by Cogent's *frame* relation, which describes how a program may affect the store. Given an input set of writable pointers w_i, an input store μ_i, an output set of pointers w_o and an output store μ_o, the relation, $w_i \mid \mu_i$ **frame** $w_o \mid \mu_o$, ensures three properties for any pointer p:

$$\begin{aligned}
\text{inertia:} \quad & p \notin w_i \cup w_o \longrightarrow \mu_i(p) = \mu_o(p)\,, \\
\text{leak freedom:} \quad & p \in w_i \longrightarrow p \notin w_o \longrightarrow \mu_o(p) = \bot\,, \text{ and} \\
\text{fresh allocation:} \quad & p \notin w_i \longrightarrow p \in w_o \longrightarrow \mu_i(p) = \bot\,.
\end{aligned}$$

Inertia ensures that pointers not in the frame remain unchanged; *leak freedom* ensures that pointers removed from the frame no longer point to anything; and *fresh allocation* ensures that pointers added to the frame were not already used. The frame relation

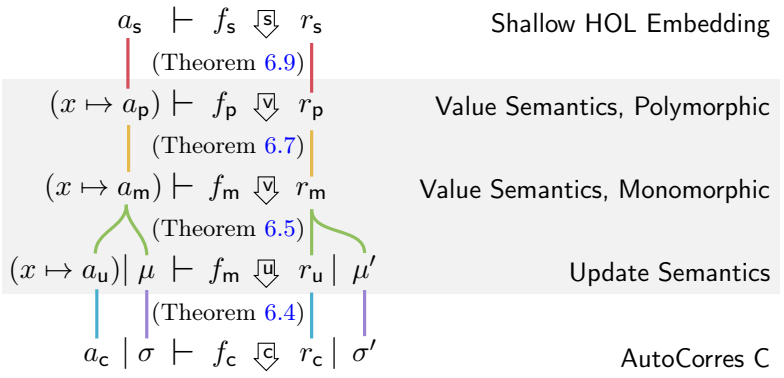


FIGURE 6.3: Cogent's semantic levels and refinement theorems.

implies that any property of a given value is unaffected by updates to unrelated parts of the heap. The frame relation holds for all Cogent computations, ensuring memory safety along with type safety:

Theorem 6.2 (Preservation and Frame Relation)**.**

$$A; \Gamma \vdash e : \tau \wedge U|\mu : \Gamma\ [r * w] \wedge U \vdash e|\mu \Downarrow_u u|\mu' \longrightarrow$$
$$\exists r'\ w'.\ r' \subseteq r \wedge w|\mu\ \textbf{frame}\ w'|\mu' \wedge u|\mu : \tau\ [r' * w']\,.$$

This states that a well-typed program will evaluate in the update semantics to a well-typed value, *and* that the frame relation holds between the writable pointers of the input environment and the output value. Note that this theorem implies update semantics type preservation (Theorem 6.1), because this theorem is also a simplification of Theorem 6.5.

6.2.5 Refinement Theorem

The overall proof that the C code refines the purely functional shallow embedding in Isabelle/HOL is broken into a number of sub-proofs and translation validation phases. Figure 6.3 gives an overview of Cogent's refinement theorems for a function $f(x)$ applied to an argument a. The compiler generates four embeddings: a top-level shallow embedding in terms of pure functions; a polymorphic deep embedding of the Cogent program, which is interpreted using the *value* semantics; a monomorphic deep embedding of the Cogent program, which can be interpreted using either the *value* or *update* semantics; and an Isabelle/HOL representation of the C code generated by the compiler, imported into Isabelle/HOL by the C-parser [187] used in the seL4 project [91]. The C-parser generates a deep embedding of C in Isabelle/HOL, and, using AutoCorres [66, 67], is then abstracted to a corresponding state-monadic embedding of the C code in HOL.

Each of these semantic layers is connected to a refinement proof by *forward simulation*: Given a *refinement relation* that relates corresponding values between two layers, we prove that if the more *concrete* (lower in the hierarchy) layer evaluates, then the more *abstract* (higher in the hierarchy) layer, given corresponding inputs, will also evaluate to corresponding outputs. The composition of all these refinements means that any property preserved by refinement, such as functional correctness, proved about all executions of the most *abstract* embedding — the Shallow HOL embedding — will also apply to the most *concrete* embedding, that is, the C code.

6.2.5.1 Refinement from Cogent's Update Semantics to C

In the first stage, Cogent proves refinement between the C implementation and the deep embedding in the update semantics. AutoCorres imports C as a nondeterministic state-monadic program shallowly embedded in Isabelle/HOL. To make our definitions more symmetrical with those of Cogent, we define a C evaluation relation as follows:

Definition 6.3 (C Evaluation Relation)**.**

$$a_\mathsf{c} \vdash \sigma \mid f_\mathsf{c} \Downarrow_\mathsf{c} r_\mathsf{c} \mid \sigma' \equiv (r_\mathsf{c}, \sigma') \in \mathit{results}(f_\mathsf{c}\ a_\mathsf{c}\ \sigma) \wedge \neg \mathit{failed}(f_\mathsf{c}\ a_\mathsf{c}\ \sigma)\,.$$

This states that given an input a_c and C heap σ, the C function f_c evaluates to r_c and an output heap σ' and that no undefined behaviour occurred (indicated by $\neg \mathit{failed}$).

The Cogent compiler additionally generates a *value relation* $\mathcal{V}^\mathsf{u}_\mathsf{c}$ and a *heap relation* $\mathcal{H}^\mathsf{u}_\mathsf{c}$, which together form the refinement relation for this refinement lemma. Using an automated technique described elsewhere [165], the Cogent proof then automatically discharges this proof obligation on a per-program basis via translation validation, while leaving open proof obligations for the user to discharge for abstract functions implemented in C.

Theorem 6.4 (Update ⇒ C refinement)**.** *For any function $f(x)$ with monomorphic Cogent embedding f_m and C embedding f_c, given an argument represented in the update semantics of Cogent as a_u and in C as a_c, we have:*

$$\begin{aligned}&\mathcal{V}^u_c(a_c, a_u) \wedge \mathcal{H}^u_c(\sigma, \mu) \wedge a_\mathsf{c} \vdash \sigma \mid f_\mathsf{c} \Downarrow_\mathsf{c} r_\mathsf{c} \mid \sigma' \longrightarrow \\ &\quad \exists r_u\ \mu'.\ (x \mapsto a_u) \vdash f_m \mid \mu \Downarrow_\mathsf{u} r_u \mid \mu' \wedge \mathcal{V}^u_c(r_c, r_u) \wedge \mathcal{H}^u_c(\sigma', \mu')\,.\end{aligned}$$

This states that if the C embedding evaluates to a result, then the corresponding Cogent update semantics will, given corresponding input values and heaps, evaluate to corresponding output values and heaps.

6.2.5.2 Refinement from Value to Update Semantics

For the second stage, we must bridge the gap between the *update* semantics and *value* semantics. This is accomplished by Cogent's proof for all well-typed programs that the update semantics refines the value semantics. As previously mentioned, this theorem combines both of the value typing relations from the two semantics $v : \tau$ and $u|\mu : \tau\ [r * w]$ into one combined relation $u|\mu \overset{\mathcal{R}}{\sim} v : \tau\ [r * w]$. In addition to typing both values, u and v, this relation also requires that they represent the same conceptual value. Prior work [147, 148] proves that the update semantics refines the value semantics by proving that this relation is preserved across the evaluation of both, and furthermore that evaluation in the update semantics implies the evaluation in the value semantics. As mentioned in Section 6.2.4, this also simultaneously proves the frame requirements hold for Cogent code.

Theorem 6.5 (Value ⇒ Update refinement)**.** *For any e where $A;\Gamma \vdash e : \tau$, if $U|\mu \overset{\mathcal{R}}{\sim} V : \Gamma\ [r * w]$ and $U \vdash e \mid \mu \Downarrow_{\mathsf{u}} u \mid \mu'$, then there exists a value v and pointer sets $r' \subseteq r$ and w' such that $V \vdash e \Downarrow_{\mathsf{v}} v$, and $u|\mu' \overset{\mathcal{R}}{\sim} v : \tau\ [r' * w']$ and $w \mid \mu$ **frame** $w' \mid \mu'$.*

This proof is parameterized by the assumption that the same holds for abstract functions. We discuss how to discharge this assumption in Section 6.3.

6.2.5.3 Monomorphisation

Recall that the Cogent compiler eliminates polymorphism by *monomorphising*, that is, replacing polymorphic functions with specialised copies, one for each type used in the program. Using template C, the Cogent compiler can do the same for the user-supplied C code. To prove that this elimination of polymorphism preserves correctness we must show that the monomorphic program refines the polymorphic program. This is accomplished by replicating the compiler's monomorphisation operations in Isabelle/HOL as functions on deep embeddings: $\mathcal{M}_e$ to monomorphise expressions and $\mathcal{M}_v$ to monomorphise values. Then, the refinement relation is simply defined:

Definition 6.6 (Monomorphisation Relation)**.**

$$\mathcal{R}^{\mathsf{p}}_{\mathsf{m}}(v_{\mathsf{m}}, v_{\mathsf{p}}) \equiv (v_{\mathsf{m}} = \mathcal{M}_v(N, v_{\mathsf{p}}))\,,$$

where N is a *name mapping*, supplied by the compiler, indicating which set of monomorphic functions correspond to which polymorphic functions.

Proving refinement proceeds on much the same lines as the other layers:

Theorem 6.7 (Polymorphic ⇒ Monomorphic refinement)**.** *For any function $f(x)$ with a polymorphic embedding f_p and argument a_p, let $f_m = \mathcal{M}_e(N, f_p)$ and $a_m = \mathcal{M}_v(a_p)$. Then we have:*

$$\mathcal{R}^p_m(a_m, a_p) \wedge (x \mapsto a_m) \vdash f_m \Downarrow r_m \longrightarrow \exists r_p.\, (x \mapsto a_p) \vdash f_p \Downarrow r_p \wedge \mathcal{R}^p_m(r_m, r_p)\,.$$

6.2.5.4 Refinement from Shallow Evaluation to Cogent's Value Semantics

Having reached the top of the refinement stack, we must cross the bridge back to shallow embeddings, that is, the pure HOL functions that serve as an executable specification. Since this embedding is just pure functions, our "evaluation" relation is just function application:

Definition 6.8 (Shallow Embedding Evaluation)**.**

$$a_\mathsf{s} \vdash f_\mathsf{s} \Downarrow_\mathsf{s} r_\mathsf{s} \equiv (f_\mathsf{s}\ a_\mathsf{s} = r_\mathsf{s})\,.$$

To show refinement, the compiler must once again connect deep and shallow embeddings, just as with the Cogent to C refinement (Theorem 6.4). Therefore, as with C, the compiler automatically produces a proof of this theorem on a per-program basis via translation validation. The compiler generates a refinement relation $\mathcal{R}^\mathsf{s}_\mathsf{p}$ for all types used in the program, and then proves:

Theorem 6.9 (Shallow ⇒ Polymorphic refinement)**.** *For any function $f(x)$ with a shallow embedding f_s and polymorphic deep embedding f_p, and arguments a_s and a_p respectively, we have:*

$$\mathcal{R}^s_p(a_p, a_s) \wedge (x \mapsto a_p) \vdash f_p \Downarrow r_p \longrightarrow \exists r_s.\, a_s \vdash f_s \Downarrow_\mathsf{s} r_s \wedge \mathcal{R}^s_p(r_p, r_s)\,.$$

6.2.6 Overall Refinement

To combine all of these refinement phases, we first define a refinement relation for all the layers of refinement:

Definition 6.10 (Combined Refinement Relation)**.**

$$\begin{aligned}&\mathcal{R}^\mathsf{s}_\mathsf{c}(v_\mathsf{c}, \sigma, v_\mathsf{u}, \mu, v_\mathsf{m}, v_\mathsf{p}, v_\mathsf{s}, \tau, r, w) \equiv \\ &\quad \mathcal{R}^\mathsf{s}_\mathsf{p}(v_\mathsf{p}, v_\mathsf{s}) \wedge \mathcal{R}^\mathsf{p}_\mathsf{m}(v_\mathsf{m}, v_\mathsf{p}) \wedge v_\mathsf{u}|\mu \overset{\mathcal{R}}{\sim} v_\mathsf{m} :\ \tau\ [r * w] \wedge \mathcal{V}^\mathsf{u}_\mathsf{c}(v_\mathsf{c}, v_\mathsf{u}) \wedge \mathcal{H}^\mathsf{u}_\mathsf{c}(\sigma, \mu)\,.\end{aligned}$$

Theorem 6.11 (Combined Refinement)**.** *Given a function $f(x)$ with embeddings f_c, f_m, f_p, f_s; argument value a_c, a_u, a_m, a_p,a_s; heap σ and store μ, we show:*

$$\begin{aligned}
&\mathcal{R}^s_c(a_c, \sigma, a_u, \mu, a_m, a_p, a_s, \tau, r, w) \wedge a_c \vdash f_c \mid \sigma \overset{\boxed{c}}{\Downarrow} v_c \mid \sigma' \longrightarrow \\
&\quad \exists r'\ w'\ \mu'.\, r' \subseteq r \wedge w|\mu\ \mathbf{frame}\ w'|\mu' \\
&\qquad \wedge \exists v_u\ v_m\ v_p\ v_s.\, (x \mapsto a_u) \vdash f_m|\mu \overset{\boxed{u}}{\Downarrow} v_u|\mu' \\
&\qquad\quad \wedge (x \mapsto a_m) \vdash f_m \overset{\boxed{v}}{\Downarrow} v_m \\
&\qquad\quad \wedge (x \mapsto a_p) \vdash f_p \overset{\boxed{v}}{\Downarrow} v_p \\
&\qquad\quad \wedge a_s \vdash f_s \overset{\boxed{s}}{\Downarrow} v_s \\
&\qquad\quad \wedge \mathcal{R}^s_c(v_c, \sigma', v_u, \mu', v_m, v_p, v_s, \tau', r', w')\,.
\end{aligned}$$

As this theorem encompasses *all* levels of refinement as well as type preservation and the frame requirements, it is sufficient to prove this theorem about (each embedding of) each abstract function, where the semantics of the deep embeddings f_p and f_m is given by the user-supplied environments ξ_v and ξ_u. A proof of this theorem for an abstract function is sufficient to integrate its verification with the Cogent verification chain.

6.2.7 Requirements on Abstract Types

Theorem 6.11 encompasses all requirements the Cogent FFI places on abstract functions, but users can also provide abstract *types*, extending the dynamic value typing rules as they see fit. Therefore, Cogent imposes several constraints on the value typing judgments which ensure that key type system invariants such as memory safety are maintained for abstract types.

Consider an abstract type of the form "$\mathtt{A}\ \overline{\tau}$", where $\mathtt{A}$ is the name of the abstract type and $\overline{\tau}$ is a list of type parameters. Since it is not surrounded with the !-operator, it is *linear* and therefore writable. The requirements of the user-defined value typing relation are as follows.

Definition 6.12 (Value Typing Requirements)**.**

$$\begin{aligned}
\text{bang}_\mathsf{v}\text{:}\quad & v : \mathtt{A}\ \overline{\tau} \longrightarrow v : (\mathtt{A}\ \overline{\tau!})!\,, \\
\text{bang}_\mathsf{u}\text{:}\quad & u|\mu : \mathtt{A}\ \overline{\tau}\ [r * w] \longrightarrow u|\mu : (\mathtt{A}\ \overline{\tau!})\ [r \cup w * \emptyset]\,, \\
\text{no-alias}\text{:}\quad & u|\mu : \mathtt{A}\ \overline{\tau}\ [r * w] \longrightarrow r \cap w = \emptyset\,, \\
\text{valid}\text{:}\quad & u|\mu : \mathtt{A}\ \overline{\tau}\ [r * w] \longrightarrow \forall p \in r \cup w.\ \mu(p) \neq \bot\,, \text{ and} \\
\text{frame}\text{:}\quad & u|\mu_i : \mathtt{A}\ \overline{\tau}\ [r * w] \longrightarrow w_i|\mu_i\ \mathbf{frame}\ w_o|\mu_o \\
& \quad \longrightarrow (r \cup w) \cap w_i = \emptyset \longrightarrow u|\mu_o : \mathtt{A}\ \overline{\tau}\ [r * w]\,.
\end{aligned}$$

The *bang* rules ensure that abstract values respect the !-operator, that is, when ! is applied to a value, the value becomes read-only; *no-alias* ensures that there is no aliasing of writable pointers by readable pointers within a value; *valid* enforces that all pointers are valid, that is, point to values; and *frame* ensures that an abstract value is unchanged if it is not part of the store that is currently being modified. For a read-only type $(\mathtt{A}\ \overline{\tau})!$, the requirements are the same, save that the writable pointer set w must also be empty:

$$\text{read-only:} \quad u|\mu : (\mathtt{A}\ \overline{\tau})!\ [r * w] \longrightarrow w = \emptyset\,.$$

6.2.8 Summary of Requirements

As can be seen from the previous subsections, proof engineers must provide a number of implementations, abstractions, and proofs for each function and type they import. We briefly summarize these here. For each abstract type imported from C, the proof engineer simply needs to prove the requirements of Definition 6.12. For each imported foreign function f, proof engineers must define a version of the function for all semantic levels in the Cogent hierarchy. That is, they must define a C implementation, an update semantics ($\xi_{\mathsf{u}}\ f$), a monomorphic value semantics ($\xi_{\mathsf{m}}\ f$), a polymorphic value semantics ($\xi_{\mathsf{p}}\ f$), and a shallow embedding in HOL. Once all embeddings have been provided, overall refinement (Theorem 6.11) must be proven for the foreign function f, which necessitates proofs of all of the other preservation and refinement theorems:

- Type preservation in the update semantics (Theorem 6.2): required by all functions that call f to prove type preservation (Theorem 6.2), and by f and all functions that call it to prove refinement (Theorems 6.4 and 6.5).
- Type preservation in the monomorphic value semantics (Theorem 6.1): required by all functions that call f to prove type preservation (Theorem 6.1), and by f and all functions that call it to prove refinement (Theorems 6.5 and 6.7).
- Refinement from the update semantics to C (Theorem 6.4): required by all functions that call f to prove refinement (Theorem 6.4).
- Refinement from the value semantics to the update semantics (Theorem 6.5): required by all functions that call f to prove refinement (Theorem 6.5).
- Refinement from the polymorphic to monomorphic value semantics (Theorem 6.7): required by all functions that call f to prove refinement (Theorem 6.7).
- Refinement from the shallow embedding to the polymorphic value semantics (Theorem 6.9): required by all functions that call f to prove refinement (Theorem 6.9).

```
length_s : [a] → word32
length_s xs = of_nat (List.length xs)

get_s : [a] → word32 → a
get_s xs i d = if unat i < List.length xs then xs ! (unat i) else d

put_s: [a] → word32 → a → [a]
put_s xs i v = xs[unat i := v]

fold_s : (b → a → c → b) → b → [a] → word32 → word32 → c → b
fold_s f acc xs frm to obs = List.fold (λacc el. f acc el obs) acc (slice frm to xs)

mapAcc_s : (b → a → c → (a, b)) → b → [a] → word32 → word32 → c → ([a], b)
mapAcc_s f acc xs frm to obs =
  let f' = (λel (xs, acc). let (el', acc') = f acc el obs in (xs @ [el'], acc'))
      (xs', acc') = List.fold f' (slice frm to xs) ([], acc)
  in (take frm xs @ xs' @ drop (max frm to) xs, acc')
```

FIGURE 6.4: The functional correctness specification of the array operations defined in Isabelle/HOL.

6.3 Arrays

With the necessary background now established, we proceed to the main contribution of this work. *Arrays* in Cogent are stored on the heap with elements having any *unboxed* type. Arrays of pointers are a separate data structure in Cogent, as their interface is complicated by the uniqueness type system. We verify five array operations: the functions length, get, and put, and the iterators fold and mapAcc, with the 32-bit Cogent implementations shown earlier in Figures 6.1 and 6.2 on pages 134 and 135, respectively.

6.3.1 Specification and Implementation

Our operations on arrays are specified as Isabelle/HOL functions on lists. The specification for the array functions presented in Figures 6.1 and 6.2 is provided in Figure 6.4. Most array operations have obvious analogs in Isabelle/HOL's list library. The library functions unat and of-nat convert words and natural numbers, take n xs and drop n xs return and remove the first n elements of the list xs respectively, slice n m xs returns the sublist that starts at index n and ends at index m of the list xs, ! returns the i^{th} element of a list and @ appends two lists.

The get and mapAcc operations do not have straightforward analogs in Isabelle/HOL's list library. The mapAcc operation is not part of the library at all, and must be implemented in terms of fold, whereas get behaves differently to its corresponding list

operation. Our implementation of `get` is not undefined when the given index is out of bounds, but instead returns the provided default value.

In Figure 6.5, we present a version of the template C implementation of arrays with some syntactic simplifications for presentation. Quoted type parameters that are later instantiated to concrete Cogent types, such as `T` or `O`, are surrounded by `<` and `>`. Generated C structure types for Cogent records and tuples such as *ArrAcc*, which normally have compiler-generated names, are *italicized*. The *dispatch* functions, in **bold** font, are how Cogent deals with higher-order functions: because the C-parser semantics do not accommodate function pointers, Cogent instead assigns a unique identifier to each function at compile time, and defines a dispatch function for each function type. Each dispatch function takes a function identifier as an argument and calls the function that corresponds to that identifier.

6.3.2 Proving Refinement

As previously mentioned, to verify our abstract C library, we must additionally provide Isabelle/HOL abstractions of our C implementation that can connect with the automatically-generated Cogent embeddings. Specifically, we must provide Isabelle/HOL abstractions for each abstract function including `put`, `get`, etc., as well as a combined value-update-type correspondence relation, analogous to the refinement relation for Theorem 6.5, for each abstract type, such as `WArr<T>`.

6.3.2.1 Abstractions

We now extend the definition of Cogent values with a new constructor for arrays. In the update semantics, arrays take the form `UWA T` *len p*, where `T` is the type of the elements of the array, *len* is the length of the array, and *p* is a pointer to the first element of the array. This is quite similar to the representation used in C, where a struct containing the length and a pointer is used (see Figure 6.5). We additionally store the type in the Cogent value so that the same form of Cogent value can be used for all the different structs generated by the Cogent compiler from the C template. In the value semantics, however, arrays take the form `VWA T` *xs*, where *xs* is simply an Isabelle/HOL list of Cogent values, similar to the representation used in the shallow embedding (see Figure 6.4).

The various array function abstractions supplied to Cogent are defined as input-output relations that are later interpreted as functions, as shown in Figures 6.6 and 6.7. These abstractions are derived directly from the axiomatization used in the verification of BilbyFs. In the update semantics (Figure 6.6), the input is a pair of the Cogent store before

```
struct WArr<T> {
   u32 len;
   <T>* vals;
}
u32 length(WArr<T>* arr) {
   return arr->len;
}
<T> get(WArr<T>* x, u32 i, <T> d) {
   if (i < x->len) {
      return x->vals[i];
   }
   return d;
}
WArr<T>* put(WArr<T>* d, u32 i, <T> v) {
   if (i < x->len) {
      x->vals[i] = v;
   }
   return x;
}
<A> fold(fid f, <A> acc, WArr<T>* x, u32 frm, u32 to, <O> obs) {
   u32 i, e;
   e = x->len;
   if (to < e) {
      e = to;
   }
   for (i = frm; i < e; i++) {
      acc = dispatch_fold(f, x->vals[i], acc, obs);
   }
   return acc;
}
ArrAcc mapAcc(fid f, <A> acc, WArr<T>* x, u32 frm, u32 to, <O> obs) {
   u32 i e;
   e = x->len;
   if (to < e) {
      e = to;
   }
   for (i = frm; i < el i++) {
      ElemAcc ea = dispatch_mapAcc(f, x->vals[i], acc, obs);
          x->vals[i] = ea.elem;
          acc = ea.acc;
   }
   ArrAcc ret = {.arr = x, .acc = acc};
   return ret;
}
```

FIGURE 6.5: C implementation of key operations on arrays.

$$\mathsf{length_u}\ (\mu,\ x)\ (\mu\text{'},\ y) = \exists \mathtt{T}\ len\ p.\ (\mu\ x = \mathtt{UWA\ T}\ len\ p) \wedge (y = len) \wedge (\mu = \mu\text{'})$$

$$\begin{aligned}&\mathsf{get_u}\ (\mu,\ (x,\ i,\ d))\ (\mu',\ y) = \\ &\quad \exists \mathtt{T}\ len\ p.\ (\mu\ x = \mathtt{UWA\ T}\ len\ p) \wedge (i < len \longrightarrow \mu\ (p{+}(\mathsf{size}\ \mathtt{T}){\times}i) = y) \\ &\qquad \wedge\ (i \geq len \longrightarrow y = d) \wedge (\mu = \mu')\end{aligned}$$

$$\begin{aligned}&\mathsf{put_u}\ (\mu,\ (x,\ i,\ v))\ (\mu',\ y) = \\ &\quad \exists \mathtt{T}\ len\ p.\ (\mu\ x = \mathtt{UWA\ T}\ len\ p) \wedge (i < len \longrightarrow \mu(p{+}(\mathsf{size}\ \mathtt{T}){\times}i \mapsto v) = \mu') \\ &\qquad \wedge\ (i \geq len \longrightarrow \mu = \mu') \wedge (x = y)\end{aligned}$$

$$\begin{aligned}&\mathsf{fold_u}\ (\mu_1,\ (f,\ acc,\ x,\ s,\ e,\ obs))\ (\mu_3,\ y) = \\ &\quad \exists \mathtt{T}\ len\ p.\ (\mu_1\ x = \mathtt{UWA\ T}\ len\ p) \wedge (s \geq len \vee s \geq e \longrightarrow \mu_1 = \mu_3 \wedge y = acc) \\ &\qquad \wedge\ (s < len \wedge s < e \longrightarrow \\ &\qquad\quad \exists v\ acc'\ \mu_2.\ \mu_1\ (p{+}(\mathsf{size}\ \mathtt{T}){\times}s) = v \\ &\qquad\qquad \wedge\ [a \mapsto (v, acc, obs)] \vdash f\,a \mid \mu_1 \overset{\mathrm{u}}{\Downarrow} acc' \mid \mu_2 \\ &\qquad\qquad \wedge\ \mathsf{fold_u}\ (\mu_2,\ (f,\ acc',\ x,\ s{+}1,\ e,\ obs))\ (\mu_3,\ y))\end{aligned}$$

$$\begin{aligned}&\mathsf{mapAcc_u}\ (\mu_1,\ (f,\ acc,\ x,\ s,\ e,\ obs))\ (\mu_4,\ y) = \\ &\quad \exists \mathtt{T}\ len\ p.\ (\mu_1\ x = \mathtt{UWA\ T}\ len\ p) \\ &\qquad \wedge\ (s < len \wedge s < e \longrightarrow \\ &\qquad\quad \exists v\ v'\ acc'\ \mu_2\ \mu_3.\ \mu_1\ (p{+}(\mathsf{size}\ \mathtt{T}){\times}s) = v \\ &\qquad\qquad \wedge\ [a \mapsto (v, acc, obs)] \vdash f\,a \mid \mu_1 \overset{\mathrm{u}}{\Downarrow} (v', acc') \mid \mu_2 \\ &\qquad\qquad \wedge\ \mu_3 = \mu_2(p{+}(\mathsf{size}\ t){\times}s \mapsto v') \\ &\qquad\qquad \wedge\ \mathsf{mapAcc_u}\ (\mu_3,\ (f,\ acc',\ x,\ s{+}1,\ e,\ obs))\ (\mu_4,\ y)) \\ &\qquad \wedge\ (s \geq len \vee s \geq e \longrightarrow \mu_1 = \mu_4 \wedge y = (x,\ acc))\end{aligned}$$

FIGURE 6.6: Cogent-compatible C abstractions of operations on arrays for the *update* semantics.

the array operation and the argument(s) to the function, and the output is a pair of the store after the operation is complete and the return value.

Note that the definitions for fold and mapAcc are recursive in two ways. The most obvious is the direct structural recursion on the array indices, which make it straightforward for Isabelle/HOL to be able to show termination. The other form of recursion is the mutual recursion with the Cogent semantics (note that fold and mapAcc invoke the Cogent semantics to evaluate the argument function). This is because the Cogent semantics are themselves parameterized by environments (ξ_u and ξ_v) containing the abstractions we are currently defining. We cannot define these embeddings and the Cogent semantics simultaneously as is normally done for mutual definitions. This is because the abstractions are defined by users later in the process after the semantics of Cogent have already been defined. Thus, we cannot close the recursion before users have provided these definitions. Therefore, we must impose an additional constraint[5] that an n-order abstract function can only call a function of an order less than n. This forces us to prove refinement for all

[5]This constraint is not a proof obligation, but merely a requirement for our framework to work automatically.

$$\mathsf{length_v}\ x\ y = \exists \mathtt{T}\ xs.\ (x = \mathtt{VWA\ T}\ xs) \wedge (\mathsf{List.length}\ xs = \mathsf{unat}\ y)$$

$$\begin{aligned}&\mathsf{get_v}\ (x,\ i)\ y = \\ &\quad \exists \mathtt{T}\ xs.\ (x = \mathtt{VWA\ T}\ xs) \wedge (\mathsf{unat}\ i < \mathsf{List.length}\ xs \longrightarrow xs\ !\ i = y) \\ &\qquad \wedge (\mathsf{unat}\ i \geq \mathsf{List.length}\ xs \longrightarrow y = 0)\end{aligned}$$

$$\mathsf{put_v}\ (x,\ i,\ v)\ y = \exists \mathtt{T}\ xs.\ (x = \mathtt{VWA\ T}\ xs) \wedge (y = \mathtt{VWA\ T}\ xs[\mathsf{unat}\ i := v])$$

$$\begin{aligned}&\mathsf{fold_v}\ (f,\ acc,\ x,\ s,\ e,\ obs)\ y = \\ &\quad \exists \mathtt{T}\ xs.\ (x = \mathtt{VWA\ T}\ xs) \wedge (\mathsf{unat}\ s \geq \mathsf{List.length}\ xs \vee s \geq e \longrightarrow y = acc) \\ &\qquad \wedge (\mathsf{unat}\ s < \mathsf{List.length}\ xs \wedge s < e \longrightarrow \\ &\qquad\quad \exists v\ acc'.\ (xs\ !\ s = v) \\ &\qquad\qquad \wedge [a \mapsto (v, acc, obs)] \vdash f\,a \Downarrow acc' \wedge \mathsf{fold_v}\ (f,\ acc',\ x,\ s{+}1,\ e,\ obs)\ y)\end{aligned}$$

$$\begin{aligned}&\mathsf{mapAcc_v}\ (f,\ acc,\ x,\ s,\ e,\ obs)\ y = \\ &\quad \exists \mathtt{T}\ xs.\ (x = \mathtt{VWA\ T}\ xs) \\ &\qquad \wedge (\mathsf{unat}\ s < \mathsf{List.length}\ xs \wedge s < e \longrightarrow \\ &\qquad\quad \exists v\ v'\ acc'\ x'.\ xs\ !\ s = v \wedge [a \mapsto (v, acc, obs)] \vdash f\,a \Downarrow (v', acc') \\ &\qquad\qquad \wedge x' = \mathtt{VWA\ T}\ xs[\mathsf{unat}\ s := v'] \wedge \mathsf{mapAcc_v}\ (f,\ acc',\ x',\ s{+}1,\ e,\ obs)\ y) \\ &\qquad\quad \wedge (\mathsf{unat}\ s \geq \mathsf{List.length}\ xs \vee s \geq e \longrightarrow y = (x,\ acc))\end{aligned}$$

FIGURE 6.7: Cogent-compatible C abstractions of operations on arrays for the *value* semantics.

functions of order $n-1$ or less, before we can prove refinement for an abstract function of order n. By imposing this ordering, we can iteratively build up these environments in a staged way. This constraint is not burdensome, and is already satisfied by all Cogent codebases [165].

Since Cogent supports polymorphism, we need to provide two abstractions for our operations in the value semantics, corresponding to the two value semantics layers in our refinement chain, monomorphic and polymorphic. As our abstractions (and also our implementation) are entirely parametric in the element type, the abstractions that are shown in Figure 6.7 can be used for both the monomorphic and the polymorphic layers, which, as we will see, trivializes the refinement proof for the monomorphisation layer of the hierarchy.

6.3.2.2 Value Typing

Unlike the approach for native Cogent values, where the value typing relations are defined as erasures of the value-update refinement relation, we define individual value-typing relations for arrays in the two semantics, and then combine them into a refinement relation in Definition 6.16

We define these typing relations with two equations, one for the writable arrays `Array T`, and one for read-only arrays (`Array T`)!. In the value semantics, these two types are identical, merely requiring that the list elements are well-typed:

Definition 6.13 (Array: Value Semantics Value Typing)**.**

$$\texttt{VWA T}\ xs : \texttt{Array T} \equiv (\forall i < \mathsf{List.length}\ xs.\ xs\ !\ i : \tau) \text{ and}$$
$$\texttt{VWA T}\ xs : (\texttt{Array T})! \equiv ((\texttt{VWA T}\ xs) : \texttt{Array T})\,.$$

For the update semantics, we define an auxiliary predicate **okay**(`UWA T` *len p*), which states that each of the values in the array (located at successive pointers starting at p) is well typed, as well as a necessary condition on *len* to ensure that our pointer arithmetic will not result in overflow. This predicate is used in the typing relation for both the writable and read-only array types. The heap footprint $[r * w]$ must consist of not just the pointer p but all of the successive pointers to each array element, because all of these memory locations are contained in the array, and ownership of all of them is passed along with the array. Since the array contains only unboxed types, we know that there are no other pointers in the heap footprint. For the read-only array type, r contains the heap footprint and w is empty, and vice versa for the writable array type.

Definition 6.14 (Array: Update Semantics Value Typing)**.**

$$\begin{aligned}
&\textbf{okay}\ (\texttt{UWA T}\ \mathit{len}\ p) \equiv (\mathsf{unat}\ \mathit{len} \times \mathsf{size}\ \texttt{T} \leq \texttt{max_word}) \\
&\quad \wedge (\forall i < \mathit{len}.\ \exists v.\ \mu\ (p + \mathsf{size}\ \texttt{T} \times i) = v\ \wedge v | \mu : \texttt{T}\ [\emptyset * \emptyset])\,, \\
&\texttt{UWA T}\ \mathit{len}\ p\ |\mu : (\texttt{Array T})!\ [r * w] \equiv \textbf{okay}\ (\texttt{UWA T}\ \mathit{len}\ p) \\
&\quad \wedge (r = \{p + i \mid \forall i.\ i < \mathit{len}\} \wedge w = \emptyset)\,, \text{ and} \\
&\texttt{UWA T}\ \mathit{len}\ p\ |\mu : (\texttt{Array T})\ [r * w] \equiv \textbf{okay}\ (\texttt{UWA T}\ \mathit{len}\ p) \\
&\quad \wedge (w = \{p + i \mid \forall i.\ i < \mathit{len}\} \wedge r = \emptyset)\,,
\end{aligned}$$

in which `max_word` is $2^{32} - 1$ (as we are using 32-bit pointers).

Recall that these the value typing relations for abstract types must satisfy the constraints of Definition 6.12. Since arrays only have elements which are of unboxed types, these constraints are trivial to discharge.

6.3.2.3 Refinement Relations

Just as Cogent's typing relations and semantics are extended by our rules for arrays, so too are the various refinement relations in each layer of the semantics. As previously

mentioned, the C implementation of our arrays bears a strong resemblance to our update semantics values. Hence, the value relation $\mathcal{V}^{\mathsf{u}}_{\mathsf{c}}$ is very simple, and the two representations of arrays are related if the length values and the pointer values are equal. For an array of 32-bit words, the value relation is defined as follows:

Definition 6.15 (`U32` Array: Update ⇒ C Value Relation)**.**

$$\mathcal{V}^{\mathsf{u}}_{\mathsf{c}}(x_{\mathsf{c}}, \texttt{UWA U32}\ len_{\mathsf{u}}\ p_{\mathsf{u}}) \equiv (len_{\mathsf{u}} = \texttt{len}_{\mathsf{c}}\ x_{\mathsf{c}}\ \wedge\ p_{\mathsf{u}} = \texttt{arr}_{\mathsf{c}}\ x_{\mathsf{c}})\,,$$

where $\texttt{len}_{\mathsf{c}}$ and $\texttt{arr}_{\mathsf{c}}$ are the struct projections generated by AutoCorres for the array type.

Note that arrays of unboxed types, such as 64-bit words, would have similar value relations, except that the element and pointer type would be set to the different unboxed type.

Next we define the value relation for arrays between the two Cogent semantics, update and value. As previously mentioned, we make use of the two value typing relations here with additional conditions to pairwise relate the corresponding elements of the two values:

Definition 6.16 (Array: Value (Mono) ⇒ Update Relation)**.**

$$\begin{aligned}&\texttt{UWA T}\ len_{\mathsf{u}}\ p_{\mathsf{u}} \mid \mu \overset{\mathcal{R}}{\sim} \texttt{VWA T}\ xs_{\mathsf{m}} : \texttt{T}\ [r * w] \equiv (\mathsf{unat}\ len_{\mathsf{u}} = \mathsf{length}\ xs_{\mathsf{m}})\\ &\quad \wedge (\forall i < len_{\mathsf{u}}.\ \exists v_{\mathsf{u}}.\ \mu(p_{\mathsf{u}} + (\mathsf{size}\ \texttt{T}) \times i) = v_{\mathsf{u}}\ \wedge\ v_{\mathsf{u}} \mid \mu \overset{\mathcal{R}}{\sim} (xs_{\mathsf{m}}\ !\ \mathsf{unat}\ i) : \texttt{T}\ [\emptyset * \emptyset])\\ &\quad \wedge (\texttt{VWA T}\ xs_{\mathsf{m}} : \texttt{T})\ \wedge\ (\texttt{UWA T}\ len_{\mathsf{u}}\ p_{\mathsf{u}} \mid \mu : \texttt{T}\ [r * w])\,.\end{aligned}$$

Note the heap footprints of elements are always empty, as the array can only contain unboxed values.

Given that we use the same abstractions for both monomorphic and polymorphic layers, the refinement relation $\mathcal{R}^{\mathsf{p}}_{\mathsf{m}}$ is just equality and its refinement theorem is trivial. Accordingly, we omit the monomorphisation relation for brevity.

Finally, we define the value relation between our value semantics and the shallow Isabelle/HOL representation. Since our value semantics representation of arrays bear a strong resemblance to our Isabelle/HOL list representation, the value relation $\mathcal{R}^{\mathsf{s}}_{\mathsf{p}}$ is very simple. The two representation of arrays are related if the lengths of the lists are the same and the elements are pairwise related.

Definition 6.17 (Array: Shallow ⇒ Polymorphic Relation)**.**

$$\mathcal{R}^{\mathsf{s}}_{\mathsf{p}}(x_{\mathsf{s}}, \texttt{VWA T}\ xs_{\mathsf{p}}) \equiv (\mathsf{length}\ xs_{\mathsf{p}} = \mathsf{length}\ x_{\mathsf{s}}) \wedge\ (\forall i < \mathsf{length}\ xs_{\mathsf{p}}.\ \mathcal{R}^{\mathsf{s}}_{\mathsf{p}}(x_{\mathsf{s}}\ !\ i, xs_{\mathsf{p}}\ !\ i))\,.$$

6.3.2.4 Refinement Theorem

Now that we have the abstractions, value typing and refinement relations, we have all the ingredients we need to prove refinement for our array operations.

The theorems structurally resemble the refinement theorems presented in Section 6.2.5. For first order functions `length`, `get` and `put`, the proofs tend to follow directly from the definition of their abstractions, implementation, refinement relation and value typing relation — the creativity is largely in the definitions, not the proofs. This is because we want the proofs to be easily automatable in future. Nonetheless, we sketch the proofs for our `put` operation, specifically for arrays of `U32`, as an illustrative example.

We first show that the update semantics abstraction is refined by the embedding of the C implementation that is automatically generated by AutoCorres. This has some similarities to Theorem 6.4, but instead of invoking the Cogent update semantics we instead appeal to our abstract function from the environment $\xi_{\mathsf{u}}(\mathsf{put}_{\mathsf{U32}})$:

Theorem 6.18 (Verifying `put`: Update ⇒ C refinement)**.**
Where the C embedding of $\mathsf{put}_{\mathsf{U32}}$ *is* put_c*:*

$$\mathcal{V}^u_c(a_c, a_u) \wedge \mathcal{H}^u_c(\sigma, \mu) \wedge a_c \vdash \sigma \mid put_c \Downarrow_{\boxed{c}} r_c \mid \sigma' \longrightarrow$$
$$\exists \mu'\ r_u.\ \xi_u(\mathsf{put}_{U32})(\mu, a_u) = (\mu', r_u) \wedge \mathcal{V}^u_c(r_c, r_u) \wedge \mathcal{H}^u_c(\sigma', \mu')\,.$$

Proof. Recall that the argument to `put` for arrays of 32-bit words is a tuple that contains an index, the array, and a 32-bit word to write to it. We take cases on the index. In the case that the index is out of bounds, both the abstraction (Figure 6.6) and the implementation (Figure 6.5) return the array unmodified with the store unmodified as well, and so the theorem is trivial. In the case where the index is within bounds, our value relation $\mathcal{V}^{\mathsf{u}}_{\mathsf{c}}$ on the arguments implies that the two argument words and indices are the same. Writing the same value to the same index within bounds only writes corresponding values to the same store locations, so it follows that our heap relation $\mathcal{H}^{\mathsf{u}}_{\mathsf{c}}$ is preserved. As it is destructively updated, the actual location of the array in memory is not changed, so the relation $\mathcal{V}^{\mathsf{u}}_{\mathsf{c}}$ is trivially preserved to the output array. □

For the next level up in the refinement hierarchy, we must show refinement from our value semantics abstraction (Figure 6.7) to our update semantics abstraction (Figure 6.6). Even though this refinement step is below monomorphisation in our hierarchy, our abstractions for `put` are agnostic to the element type of the array, so we can generalize the proof to arrays of any element type. The theorem resembles Theorem 6.5, but with the Cogent semantics replaced with our supplied abstractions in ξ_{v} and ξ_{u}.

Theorem 6.19 (Verifying put: Value ⇒ Update refinement)**.** *For an element of type* T*, if* $a_u|\mu \overset{\mathcal{R}}{\sim} a_m :$ (Array T, U32, T) $[r * w]$ *and* $\xi_u(\mathsf{put}_\mathsf{T})(\mu, a_u) = (\mu', v_u)$*, then there exists a value* v_m *and pointer sets* $r' \subseteq r$ *and* w' *such that* $\xi_\mathsf{v}(\mathsf{put}_\mathsf{T})(a_m) = v_m$*, and* $v_u|\mu' \overset{\mathcal{R}}{\sim} v_m :$ Array T $[r' * w']$ *and* $w \mid \mu$ **frame** $w' \mid \mu'$*.*

Proof. We also prove this by cases on the index. In the case where the index is out of bounds, we trivially have correspondence. In the case where the index is within bounds, we prove that modifying the element at the given index from the pointer on the store is equivalent to modifying the element at the given index in the corresponding list. To demonstrate this, we need to show that there is a one-to-one mapping between store addresses of elements to list indices. This is why we include in our typing relation that the array element addresses do not overflow the heap (Definition 6.14). Since this tells us the array cannot wrap around itself, each element in the array has a unique address, giving us our mapping. We also need to show that the frame conditions are satisfied, but because our implementation is memory safe, these follow easily from our definitions. □

Next, we must show refinement from the polymorphic layer to the monomorphic layer, but because put is a first-order function (neither taking functions as arguments nor returning them), the value relations for its arguments and return values simplify to equality. Furthermore, as our Cogent abstractions for monomorphic and polymorphic layers are identical, the proof of refinement reduces to showing that identical functions will give equal results given equal inputs.

With monomorphisation thus handled, we must now make the final shift to the specification level (Figure 6.4), and prove a theorem analogous to Theorem 6.9. Note that this theorem is also generic for any element type:

Theorem 6.20 (Verifying put: Shallow ⇒ Polymorphic Value)**.** *The shallow embedding of* put *is called* put_s*. Given arguments* a_s *and* a_p*, we have:*

$$\mathcal{R}_p^s(a_p, a_s) \wedge \xi_\mathsf{v}(\mathsf{put})(a_p) = r_p \longrightarrow \exists r_s.\ a_s \vdash \mathsf{put}_s \ \overset{\boxed{\mathsf{s}}}{\Diamond}\ r_s \wedge \mathcal{R}_p^s(r_p, r_s)$$

Proof. This follows from the definitions of put_v (in ξ_v) and put_s, as well as the value relation from Definition 6.17. □

Theorems 6.18 to 6.20 are all we need to compose the correctness of put with Cogent's refinement hierarchy. For higher-order functions fold and mapAcc the proofs are broadly similar, but slightly complicated by the presence of functions as arguments. Our theorems assume that type preservation and refinement hold for all of their argument functions — an assumption that is discharged by the Cogent compiler (for Cogent functions) or by

```
repeat_s : nat → (a → b → bool) → (a → b → a) → a → b → a
repeat_s 0 _ _ acc _ = acc
repeat_s (Suc n) f g acc obs =
  if (f acc obs) then acc else repeat_s n f g (g acc obs) obs
```

FIGURE 6.8: Shallow embedding of the generic loop.

manual proofs (for C functions). As always with looping functions, the majority of the proof effort is concentrated on proving that loop invariants are maintained, and not on any aspect of the Cogent framework.

6.4 Generic Loops

Aside from arrays and their associated iterators and operations, the most commonly used library functions in BilbyFs are generic loop functions. These are used to create loops that do not simply iterate over a particular data structure, and instead accommodate non-standard search patterns over data structures. For example, in Section 6.5.2 we use such a function in a binary search.

We first verify the function repeat, which is given the following type signature in Cogent:

$$\mathsf{repeat} : (\mathtt{U32},\, (a!,\, b!) \to \mathtt{Bool},\, (a,\, b!) \to a,\, a,\, b!) \to a\,.$$

The expression repeat n *stop step acc obs* operates on some mutable state *acc* (of linear type) and some observer data *obs* (of read-only type), and runs the loop body *step* on it at most n times, or until *stop* returns true. Figure 6.8 gives the Isabelle/HOL shallow embedding and Figure 6.9 gives the template C implementation. Figures 6.10 and 6.11 gives the Cogent-compatible embeddings for the environments ξ_u and ξ_v. Note that these must invoke the Cogent semantics twice, once to evaluate each of the argument functions. Discharging the required proof obligations connecting all of these embeddings and maintaining Cogent's invariants is even more straightforward than for other higher-order functions, such as mapAcc, as here we do not need to consider custom data structures such as arrays. This means that we can define a polymorphic abstraction of the C code compatible with AutoCorres, enabling us to prove the entire refinement chain polymorphically and thereby largely bypass the boilerplate of multiple type instantiations.

6.5 Composing Verification of Cogent and C

Now that we have demonstrated how to prove the obligations placed on C code, we illustrate how to integrate them with proofs about Cogent code. Firstly, as a simple

example, we return to the `sum` function presented in Figure 6.2.

After compiling this program and linking it to our C library, the compiler produces a refinement theorem similar to Theorem 6.11. However, the phases that are generated per-program via translation validation, such as the final refinement to C, leave open proof obligations for the user to discharge about abstract functions[6]. In our sum example, Cogent will generate obligations about length and about fold. The obligation about length is exactly our C refinement theorem for length (the length analog of Theorem 6.18), and the obligation for fold is an instance of our theorem for fold: the obligation requires showing refinement under the assumption that the argument function is add, whereas our theorem is generically proven for any function that maintains Cogent's invariants and refinement. Since add is defined in Cogent, Cogent generates the required theorem for the argument function, allowing us to easily discharge this obligation.

We additionally must instantiate our sets of abstract values and types with arrays, and the environments ξ_v and ξ_u with our abstractions from Figures 6.6 and 6.7. This instantiation requires us to additionally provide proofs similar to that of Theorem 6.19 for each function, as well as proofs that all the type conditions from Section 6.2.7. These proofs ultimately connect our C code to the generated shallow embedding, which strongly resembles the original Cogent code:

$$\mathsf{add_s}\ x\ y = x + y$$
$$\mathsf{sum_s}\ xs = \mathsf{fold_s}\ \mathsf{add_s}\ 0\ xs\ 0\ (\mathsf{length}_s\ xs)\ ()$$

With all of these proofs in place, we get a refinement theorem like Theorem 6.11 for the function sum, which leaves no function unverified. This refinement theorem allows us to prove properties about $\mathsf{sum_s}$ using equational reasoning, and have these properties also apply to the C implementation.

6.5.1 Discharging Word Array Assumptions in BilbyFs

As noted earlier, the previous verification of the functional correctness of key operations in BilbyFs [9] assumed the correctness of abstract functions, including an axiomatization of array operations [10]. We removed the axiomatization for the five core array operations that we verified, and were able to show that the functional correctness proofs compose for the combined system. The array operations that remain unverified are functions like `create`, `set`, `copy`, `cmp`, that depend on platform-specific functions such as `malloc`.

[6] At the time of writing, Cogent's shallow phase (Theorem 6.9) implicitly assumes abstract function correctness rather than doing so explicitly as done in other phases. So for now, we just copy and discharge these.

```
1  <A> repeat(u32 n, fid f, fid g, <A> acc, <O> obs) {
2     u32 i;
3     for (i = 0; i < n; i++) {
4        if (dispatch_f(f, acc, obs)) {
5           break;
6        } else {
7           acc = dispatch_g(g, acc, obs);
8        }
9     }
10    return acc;
11 }
```

FIGURE 6.9: C implementation of the generic loop.

$$
\begin{aligned}
&\mathsf{repeat}_{\mathrm{u}}\ (\mu_1,\ (n,\ f,\ g,\ acc,\ obs))\ (\mu_3,\ y) = \\
&\quad (n > 0 \longrightarrow \exists b.\ [a \mapsto (acc, obs)] \vdash f\ a \mid \mu_1 \Downarrow_{\mathrm{u}} b \mid \mu_1 \wedge (b \longrightarrow \mu_1 = \mu_3 \wedge y = acc) \\
&\qquad \wedge (\neg b \longrightarrow \exists \mu_2\ acc'.\ [a \mapsto (acc, obs)] \vdash g\ a \mid \mu_1 \Downarrow_{\mathrm{u}} acc' \mid \mu_2 \\
&\qquad\quad \wedge \mathsf{repeat}_{\mathrm{u}}\ (\mu_2,\ (n{-}1,\ f,\ g,\ acc',\ obs))\ (\mu_3,\ y))) \\
&\quad \wedge (n = 0 \longrightarrow \mu_1 = \mu_3 \wedge y = acc)
\end{aligned}
$$

FIGURE 6.10: Cogent-compatible C abstraction for the generic loop for the *update* semantics.

$$
\begin{aligned}
&\mathsf{repeat}_{\mathrm{v}}\ (n,\ f,\ g,\ acc,\ obs)\ y = \\
&\quad (n > 0 \longrightarrow \exists b.\ [a \mapsto (acc, obs)] \vdash f\ a \Downarrow_{\mathrm{v}} b \wedge (b \longrightarrow y = acc) \wedge \\
&\qquad (\neg b \longrightarrow \exists acc'.\ [a \mapsto (acc, obs)] \vdash g\ a \Downarrow_{\mathrm{v}} acc' \\
&\qquad\quad \wedge \mathsf{repeat}_{\mathrm{v}}\ (n{-}1,\ f,\ g,\ acc',\ obs)\ y)) \\
&\quad \wedge (n = 0 \longrightarrow y = acc)
\end{aligned}
$$

FIGURE 6.11: Cogent-compatible C abstraction for the generic loop for the *value* semantics.

6.5.2 Binary Search

As a final example, we implement binary search, which has a non-standard iteration pattern, in Cogent. The implementation shown in Figure 6.12 uses our previously defined and verified array functions and repeat. Note that the repeat function expects a numeric argument that indicates the maximum number of iterations, and can be thought of as the *fuel* for the function. For binary search, the maximum number of iterations is bounded by $\mathcal{O}(\log n)$, for an array of size n. However, since Cogent does not have a native function for computing the log of a number, and we also do not want to compute the log with the repeat function, we do not set the fuel argument to the exact maximum number of iterations. Instead, we set the fuel argument, the maximum number of iterations, to be the length of the array. The implementation still requires $\mathcal{O}(\log n)$ time, as it will always exit early from the stop condition. Note that this fuel argument is an easy way to ensure that Isabelle/HOL is convinced that our functions terminate.

```
type Range = (U32, U32, Bool)
stop : (Range, ((Array U32)!, U32)) → Bool
stop ((l, r, b), (arr, v)) = b ∨ l ≥ r
search : (Range, ((Array U32)!, U32)) → Range
search ((l, r, b), (arr, v)) =
  let m = l + (r − l) ÷ 2 and
      x = get (arr, m, 0)
  in if | x < v → (m+1, r, b)
        | x > v → (l, m, b)
        | else → (m, r, True)
binary-search : ((Array U32)!, U32) → U32
binary-search (arr, v) =
  let len = length arr and
      (l, r, b) = repeat (len, stop, search, (0, len, False), (arr, v))
  in if b then l else → len
```

FIGURE 6.12: The binary search algorithm in Cogent.

We firstly prove that the generated shallow embedding, which strongly resembles the Cogent code, is correct:

Theorem 6.21 (Correctness of Cogent binary search)**.**

$$\begin{aligned}&\mathsf{sorted}\, xs \;\wedge\; \mathsf{length}\, xs < 2^{32} \;\longrightarrow\\ &\quad (i < \mathsf{length}\, xs \;\longrightarrow\; xs\,!\,i = v) \;\wedge\; (\neg\, i < \mathsf{length}\, xs \;\longrightarrow\; v \notin \mathsf{set}\, xs)\,,\end{aligned}$$

where $i = \mathsf{unat}\ (\mathsf{binary\text{-}search}_s\ (xs,\ v))$.

Note that the functions length and set, and predicate **sorted** are Isabelle/HOL's list library functions that return the length of a list, turn a list to a set, and check whether a list is sorted, respectively. We denote search failure by returning an index that is out of bounds.

From our overall refinement theorem (Theorem 6.11), we can easily conclude that the C implementation is correct and remove any reference to the FFI:

Corollary 6.22 (Correctness of the C binary search)**.** *Let xs be the list abstraction of the array arr for C heap* σ*,* **valid** $(\sigma,\ arr)$ *be the predicate that states that the array is valid, that is, the array's size (in bytes) is less than the size of memory* $(2^{32}$ *bytes) and is well-formed, and* **same** $(\sigma,\ \sigma',\ arr)$ *be the predicate that states that an array is the same for the given C heaps:*

$$\begin{aligned}&\mathbf{sorted}\, xs \;\wedge\; \mathbf{valid}\,(\sigma, arr) \;\wedge\; (arr, v) \vdash \sigma \mid \mathsf{binary\text{-}search}_c \Diamond\!\!\boxed{\mathsf{c}}\!\!\Diamond\ i \mid \sigma' \;\longrightarrow\\ &\quad \mathsf{valid}\,(\sigma', arr) \;\wedge\; \mathbf{same}\,(\sigma, \sigma', arr) \;\wedge\; (\mathsf{unat}\, i < \mathsf{length}\, xs \;\longrightarrow\; xs\,!\,(\mathsf{unat}\, i) = v)\\ &\quad \wedge\; (\neg\, \mathsf{unat}\, i < \mathsf{length}\, xs \;\longrightarrow\; v \notin \mathsf{set}\, xs)\,.\end{aligned}$$

This theorem does not depend on any additional assumptions about functions nor any other part of the Cogent framework. With this, we have shown that the Cogent framework enables the verification of combined Cogent-C systems.

6.6 Limitations and Future Work

In our development of the word array and generic loop libraries, we came across several aspects that could be improved. These relate to how Cogent deals with higher-order functions in its verification framework. The verification framework defines a hierarchy for higher-order functions, where an n-order function can only call functions that have an order of $n - 1$ or less. This means that a version of a higher-order function needs to be defined for each order.

Cogent automatically handles definition of native higher-order functions for each level. However, foreign functions do not benefit from this automation, meaning that a developer technically has to manually define the embeddings for each level and prove the FFI requirements for each level. In practice, only one level needs to be defined as the embeddings and proofs can be made to be agnostic of the level, but the abstract environments (ξ_u and ξ_v) still need to be explicitly defined. Providing some form of automation for this would help improve usability.

Another issue related to the verification framework for the hierarchy of higher-order functions is that the maximum level that the Cogent compiler can automatically generate is up to two levels. This choice was made because most systems code do not use higher-order functions, and if they do, second order functions are sufficient. This, however, is not sufficient for more abstract code. Removing this limitation is one potential avenue of future work.

Besides higher-order function hierarchy, the Cogent compiler currently assumes that all foreign functions are correct. While this is not wrong to assume, as the correctness of a native function that calls a foreign function depends on the correctness of the foreign function, how goes about doing this is less than ideal. Currently, it axiomatizes the correctness of foreign functions. Moreover, it assumes that these should be correct for all possible inputs. The first issue is troublesome as it lacks modularity. The second issue is troublesome as higher-order foreign functions weaker correctness theorems as additional type preservation as they themselves depend on the correctness of the functions that could be arguments to them. Solving both of these would greatly improve Cogent.

Other potential avenues for future work pertain to Cogent lacks native support for useful language constructs, including recursion and memory management. Hence, there exist

several useful data structures that cannot be implemented directly in Cogent. For example, arrays of containing boxed values cannot be implemented nor can doubly linked lists. Such data structures and their operations are ideal candidates to implement using the approach described in this chapter. Not only would it improve the usability of Cogent, but it would also provide additional case studies to investigate the limitations of this approach. Moreover, it would allow Cogent language developers to test new features without having to modify the existing language and verification infrastructure.

6.7 Summary

We have demonstrated our cross-language approach to proving software correct. Our systems mix Cogent, a safe functional language with a compiler that proves most of the required theorems automatically, and C, an unsafe imperative language with few guarantees. Specifically, we verified the array implementation and general loop iterators provided in Cogent's ADT library, which were used the implementation of real-world file systems, and we showed that they maintain the invariants required by Cogent. This enabled us to eliminate some key assumptions in the pre-existing verification of the BilbyFs file system in Cogent. These case-studies demonstrate that manual C verification can be straightforwardly composed with Cogent's refinement chain, leading to a top-level shallow embedding that can be seamlessly connected with functional correctness specifications to ensure the correctness of an overall Cogent-C system.

Chapter 7

Conclusions and Future Directions

This thesis presented two complementary investigations. The first and primary investigation addressed the lack of formally verified indexed pattern search algorithms. In particular, this thesis presented the verification of efficient algorithms for constructing text indexes and algorithms that use these indexes for pattern search. The secondary investigation addressed the related but orthogonal problem of reducing the formal verification costs for programs. Specifically for programs composed of parts written in different languages with different static guarantees and levels of expressiveness and abstraction. The following sections summarize the investigations and describe future work based on these investigations.

7.1 Formally Verified Indexed Pattern Search

This thesis has presented the formalization of efficient indexed pattern search algorithms in Isabelle/HOL [137]. Specifically, two key algorithms for constructing text indexes were formally verified: the SA-IS algorithm for linear-time construction of suffix arrays [140]; and the Burrows-Wheeler Transform (BWT) [35], an invertible transformation that is the foundation of BWT based text indexes [54, 55, 116]. Rounding out this project, two indexed search algorithms, binary search and backward search [54], that use these text indexes for indexed pattern search were also formally verified. By formalizing the SA-IS algorithm, the BWT and the indexed search algorithms, this thesis develops a deeper understanding of these algorithms and corroborates their reliability and accuracy. Ultimately, these accomplishments both demonstrate the feasibility and establishes the foundations for using formal methods to guarantee the correctness of algorithms for text retrieval, data compression and bioinformatics applications.

The first part of the project demonstrated the first formal verification of a linear time and space suffix array construction algorithm. This was achieved by first developing an axiomatic characterization of suffix arrays, which was validated by verifying the correctness of a simple suffix array construction algorithm, that was then used as the functional correctness specification. Next, core suffix properties that are foundational to the SA-IS algorithm and several other suffix array construction algorithms were formalized and in many cases generalized. Finally, the SA-IS algorithm was implemented in Isabelle/HOL, such that it had enough detail to be extracted to executable Haskell code, and then it was shown to satisfy the functional correctness specification, using the formalized suffix properties.

The second part of the project demonstrated the first formalization of the BWT, proving that it is invertible. This was achieved by implementing the canonical construction of the BWT and an alternate construction using suffix arrays in Isabelle/HOL and then proving their equivalence. Using the construction based on suffix arrays, core properties about the BWT, which are the basis for it being invertible and its usage as a text index, were formalized. As a result of this, additional reusable properties about counting and locating items in lists were also formalized. Finally, using the core properties about the BWT, the BWT was shown to be invertible by implementing it in Isabelle/HOL and verifying the correctness of a linear-time inverse.

To complete this work on indexed pattern search, two algorithms, binary search and backward search, that use suffix arrays and the BWT, respectively, for indexed pattern search were formalized. Here, the pattern search problem was formally specified along with a reformulation in terms of an interval over the suffix array, which was then shown to be equivalent. Using this alternate formulation, binary search over the suffix array was shown to correctly produce the interval over the suffix array containing all instances matching the pattern and similarly for backward search over the BWT.

The formalizations of the Isabelle/HOL implementations of the SA-IS algorithm, BWT and search algorithms provide correctness guarantees for the semantics of these algorithms. This establishes the foundation for developing demonstrably correct and efficient implementations by refinement using either top-down [77, 100] or bottom-up [66, 143, 148] approaches. It also forms a critical step for formal complexity analysis of these algorithms using similar refinement approaches to Zhan and Haslbeck [190]. The axiomatic characterization of suffix arrays, which resulted from the SA-IS formalization, forms the basis for the verification of other suffix array construction algorithms [84, 141], with algorithms that share features with the SA-IS algorithm being particularly attractive [58, 64, 78, 94, 113, 139]. Likewise, the formalization of the BWT forms the basis for the verification of algorithms that utilize structures based on the BWT [55, 56, 116] and

also for the formalization of theoretical investigations in text retrieval and data compression [60, 119, 168]. Ultimately, this work furthers the reliability of text retrieval, data compression and bioinformatics, demonstrating how formal verification techniques can be used to achieve these aims.

7.2 Composing Verifications of Multi-Language Programs

Additionally, this thesis has demonstrated a cross-language approach for software verification. This was achieved using a mixed Cogent-C system, where programs written in the safe functional language Cogent [148] can interface with foreign functions written in the unsafe imperative language C, and still maintain the strong static guarantees that Cogent provides. Moreover, the automatic refinement proofs from Isabelle/HOL shallow embeddings to C, which the Cogent compiler produces for Cogent programs, can be composed with manually developed correctness proofs of the Isabelle/HOL shallow embeddings and the correctness proofs of the foreign C functions to obtain correctness of the whole Cogent-C system. This addresses the important issue of guaranteeing correctness of multi-language software developments.

In this second part of the thesis, safe interoperability of Cogent programs with foreign C functions is demonstrated for C functions satisfying Cogent's FFI. Specifically, this is shown by proving that an array and general loop iterator ADT library, implemented in C, is well-typed and adheres to Cogent's frame constraints, that is, a description on how to interact with memory used by Cogent programs, can be used by Cogent programs without invalidating static guarantees that Cogent's type system provides. Moreover, it has demonstrated that the manual proof of correctness of the ADT library and the correctness proofs of the Isabelle/HOL shallow embeddings of the Cogent programs can be composed with the automatically generated refinements proofs, which are produced by the Cogent compiler, to obtain correctness for the whole system even for pre-existing verifications such as BilbyFs, a file system written in Cogent [9].

The formally verified and Cogent compatible library of arrays and general loop iterators, implemented in C, demonstrates that Cogent provides a cross-language approach for software verification for mixed Cogent-C systems. Thus, software developers can develop systems that benefit from Cogent's strong static guarantees and certifying compiler for simpler verification, while still being able to use more expressive language features present in C. Moreover, these language features can be gradually incorporated into formally verified libraries that are compatible with Cogent, amortizing their verification costs over subsequent verification projects, and ultimately resulting in the development of fully verified systems.

Cogent's FFI also provides a way for language developers to test new language features without having to carry out costly integrations with the existing language and verification framework. This greatly simplifies language development while also allowing software developers to use experimental features in a plug-and-play fashion. At present, Cogent lacks native support for useful language constructs, such as recursion or iteration and memory management. These are excellent candidates for future additions to Cogent compatible C libraries, as demonstrated by our C library for a restricted form of iteration. These libraries and their verifications can then be used as a basis for native support in Cogent. Overall Cogent is both a basis and an example on how to support multi-language software verification and how to enrich a language seamlessly.

7.3 Final Remarks

This thesis has demonstrated that formal verification can be effectively applied to indexed pattern search algorithms. By formally verifying the SA-IS algorithm, a linear-time suffix array construction, and the BWT for text index construction and the backwards search algorithm for pattern search within Isabelle/HOL, we have provided the first machine-checked guarantees for algorithms that are used in high integrity applications but were previously unverified algorithms. These results lay the groundwork for formally verified, high-assurance implementations of efficient indexed pattern search algorithms.

This thesis has also contributed to the broader challenge of verifying multi-language systems. Through the Cogent project, we have shown how to compositionally verify software components written in both the high-level functional language Cogent, which has strong static guarantee, and the low-level imperative language C. By using a principled FFI, the two languages demonstrates that it is possible to obtain a fully verified system, composing both automated and manual proofs across the language boundaries. Thus, we have demonstrated an approach that reduces the cost of verification, making formally verified systems more attainable.

Appendix A

Indexed Pattern Search Formalization

theory *Pattern-Search-Set*
  imports *SuffixArray.Suffix-Util*
    BurrowsWheeler.Count-Util
begin

A.1 Pattern Search

The pattern search problem attempts to find instances in a sequence S that match the pattern P. That is, locations in S where P occurs.

definition *patsearch-set* :: *'a list* ⇒ *'a list* ⇒ *nat set*
  where
patsearch-set xs ys = {*i*. *i* < *length xs* ∧ *list-slice xs i* (*i* + *length ys*) = *ys*}

lemma *patsearch-set-nil*:
  patsearch-set xs [] = {*0*..<*length xs*}
  by (*simp add*: *patsearch-set-def antisym subset-iff*)

lemma *suc-patsearch-set-cons-cons*:
  shows *Suc* ‘ (*patsearch-set xs* (*a* # *b* # *as*)) ⊆ *patsearch-set xs* (*b* # *as*)
proof *safe*
  fix *x*
  assume *x* ∈ *patsearch-set xs* (*a* # *b* # *as*)
  hence *A*: *x* < *length xs list-slice xs x* (*x* + *length* (*a* # *b* # *as*)) = *a* # *b* # *as*
    by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq patsearch-set-def*)+

  have *list-slice xs* (*Suc x*) (*Suc x* + *length* (*b* # *as*)) = *b* # *as*

```
    using A(1) A(2) list-slice-Suc by fastforce
  moreover
  have length (list-slice xs x (x + length (a # b # as))) = length (a # b # as)
    using A(2) by auto
  hence min (length xs) (x + length (a # b # as)) − x = length (a # b # as)
    by fastforce
  hence x + length (a # b # as) ≤ length xs
    by (metis A(1) add-diff-inverse-nat linorder-le-cases min-def order.asym)
  hence Suc x < length xs
    by force
  ultimately show Suc x ∈ patsearch-set xs (b # as)
    by (simp add: patsearch-set-def)
qed

lemma suc-funpow-patsearch-set-suffix:
  assumes Suc k < length ys
  shows (Suc^^k) ‘ (patsearch-set xs ys) ⊆ patsearch-set xs (suffix ys k)
  using assms
proof (induct ys arbitrary: k)
  case Nil
  then show ?case
    by simp
next
  case (Cons a ys k)
  note IH = this

  show ?case
  proof (cases k)
    case 0
    then show ?thesis
      by simp
  next
    case (Suc n)
    with IH
    have (Suc ^^ n) ‘ patsearch-set xs ys ⊆ patsearch-set xs (suffix ys n)
      by (metis Suc-less-eq length-Cons)
    moreover
    have patsearch-set xs (suffix ys n) = patsearch-set xs (suffix (a # ys) k)
      by (simp add: Suc)
    moreover
    from suc-patsearch-set-cons-cons
    have Suc ‘ patsearch-set xs (a # ys) ⊆ patsearch-set xs ys
    by (metis Cons.prems Cons-nth-drop-Suc One-nat-def Suc-less-eq add-lessD1 drop0 length-Cons
          plus-1-eq-Suc)
```

```
    hence (Suc ^^ k) ‘ patsearch-set xs (a # ys) ⊆ (Suc ^^ n) ‘ patsearch-set xs ys
      by (simp add: Suc image-subset-iff)
    ultimately show ?thesis
      by order
  qed
qed

lemma patsearch-set-app-subset:
  patsearch-set xs (ys @ zs) ⊆ patsearch-set xs ys
proof
  fix x
  assume x ∈ patsearch-set xs (ys @ zs)
  hence x ∈ {i. i < length xs ∧ list-slice xs i (i + length (ys @ zs)) = ys @ zs}
    using patsearch-set-def by blast
  hence A: x < length xs list-slice xs x (x + length (ys @ zs)) = ys @ zs
    by blast+
  have x ∈ {i. i < length xs ∧ list-slice xs i (i + length ys) = ys}
  proof
    from A(2)
    have list-slice xs x (x + length ys) = ys
      by (metis add-diff-cancel-left' append.assoc append-eq-conv-conj list-slice-to-suffix)
    with A(1)
    show x < length xs ∧ list-slice xs x (x + length ys) = ys
      by blast
  qed
  then show x ∈ patsearch-set xs ys
    using patsearch-set-def by blast
qed

lemma patsearch-set-count-list:
  patsearch-set xs [a] = {} ⟷ count-list xs a = 0
proof
  assume count-list xs a = 0
  hence ∀ i < length xs. xs ! i ≠ a
    by (meson notin-count nth-mem)
  hence ∀ i < length xs. list-slice xs i (i + length [a]) ≠ [a]
    by (metis add.right-neutral length-Cons lessI list.size(3) nth-Cons-0 nth-list-slice)
  then show patsearch-set xs [a] = {}
    by (simp add: patsearch-set-def)
next
  assume patsearch-set xs [a] = {}
  hence ∀ i < length xs. list-slice xs i (i + length [a]) ≠ [a]
    by (simp add: patsearch-set-def)
  hence ∀ i < length xs. xs ! i ≠ a
```

```
    using list-slice-Suc by fastforce
  then show count-list xs a = 0
    by (simp add: in-set-conv-nth)
qed

lemma patsearch-set-cons-not-empty:
  patsearch-set xs (a # ys) ≠ {} ⟹ patsearch-set xs ys ≠ {}
proof (induct ys arbitrary: a)
  case Nil
  then show ?case
    using patsearch-set-count-list patsearch-set-nil by fastforce
next
  case (Cons b ys)
  with suc-patsearch-set-cons-cons[of xs a b ys]
  show ?case
    by blast
qed

lemma patsearch-set-app-not-empty1:
  patsearch-set xs (ys @ zs) ≠ {} ⟹ patsearch-set xs zs ≠ {}
proof (induct ys arbitrary: zs)
  case Nil
  then show ?case
    by simp
next
  case (Cons a ys)
  with patsearch-set-cons-not-empty
  show ?case
    by (metis append-Cons)
qed

lemma patsearch-set-not-empty-subset:
  patsearch-set xs ys ≠ {} ⟹ set ys ⊆ set xs
  by (metis (mono-tags, lifting) Collect-empty-eq list-slice-subset patsearch-set-def)

lemma patsearch-set-app-not-empty2:
  patsearch-set xs (ys @ zs) ≠ {} ⟹ patsearch-set xs ys ≠ {}
  using patsearch-set-app-subset by blast

end
theory Pattern-Search-Idx
  imports Pattern-Search-Set
          SuffixArray.Prefix
begin
```

A.2 Indexed Pattern Search

If the suffixes of the sequence S are listed in sorted order, then all instances that match P will be found in one contiguous block in the sorted suffix list. Below we define to functions that find the start (inclusive) and end (exclusive) of the block. We choose to define it this way so that if start greater than or equal to end, then we know that the pattern P does not exist in S. Moreover, it simplifies properties about the end location (see below for more details).

A.2.1 Comparison Properties

lemma *le-take-n*:
  fixes *xs* :: *'a* :: *linorder list*
  shows *xs* ≤ *ys* ⟹ *take n xs* ≤ *take n ys*
proof (*induct xs arbitrary*: *ys n*)
  case *Nil*
  then show *?case*
    by *simp*
next
  case (*Cons a xs ys n*)
  note *IH* = *this*

  have ∃ *b ys'*. *ys* = *b* # *ys'*
    using *Cons.prems le-Nil neq-Nil-conv* **by** *blast*
  then obtain *b ys'* **where**
    ys = *b* # *ys'*
    by *blast*
  hence *a* ≤ *b*
    using *IH*(*2*)
    by *fastforce*
  hence *a* = *b* ∨ *a* < *b*
    by *auto*
  then show *?case*
  proof
    assume *a* < *b*
    then show *?case*
      by (*simp add*: ‹*ys* = *b* # *ys'*› *take-Cons'*)
  next
    assume *a* = *b*
    show *?case*
    proof (*cases n*)
      case *0*
      then show *?thesis*

```
      by auto
    next
      case (Suc m)
      with IH(1)[of ys' m]
      have prefix xs m ≤ prefix ys' m
        using Cons.prems ‹a = b› ‹ys = b # ys'› by fastforce
      then show ?thesis
        using Suc ‹a ≤ b› ‹ys = b # ys'› by fastforce
    qed
  qed
qed
```

A.2.2 Max Properties

```
lemma Max-Suc:
  ⟦A ≠ {}; finite A⟧ ⟹ Max (Suc ‘ A) = Suc (Max A)
  by (simp add: hom-Max-commute)

lemma Max-upt:
  i ≤ j ⟹ Max {i..<Suc j} = j
proof −
  assume i ≤ j
  hence {i..<Suc j} ≠ {}
    by auto
  moreover
  have finite {i..<Suc j}
    by simp
  moreover
  have j ∈ {i..<Suc j}
    by (simp add: ‹i ≤ j›)
  moreover
  have ∀ x ∈ {i..<Suc j}. x ≤ j
    by auto
  ultimately show ?thesis
    using Max-eqI by blast
qed
```

A.2.3 Starting Index for Indexed Pattern Search

We define a function that returns the first location of that block. Specifically, we find the first location in the sorted suffix list of S that has a prefix which is greater than or equal to the pattern P, up to the length P. That is, the comparison is only between P and the prefix which is at most length P as well.

abbreviation *patsearch-idx-start* :: *'a* :: *linorder list list* ⇒ *'a list* ⇒ *nat*
where
patsearch-idx-start xs ys ≡
Min ({*i*. *i* < *length xs* ∧ *ys* ≤ *prefix* (*xs* ! *i*) (*length ys*)} ∪ {*length xs*})

lemma *patsearch-idx-start-upper*:
shows *patsearch-idx-start xs ys* ≤ *length xs*
by *force*

lemma *patsearch-idx-start-less*:
assumes *i* < *patsearch-idx-start xs ys*
shows *prefix* (*xs* ! *i*) (*length ys*) < *ys*
proof −
let *?X* = {*i*. *i* < *length xs* ∧ *ys* ≤ *prefix* (*xs* ! *i*) (*length ys*)}

have *i* < *length xs*
using *assms order-less-le-trans patsearch-idx-start-upper* **by** *blast*

have *i* ∉ *?X*
by (*metis* (*no-types*, *lifting*) *Min-gr-iff assms finite.insertI finite-nat-set-iff-bounded*
inf-sup-aci(*5*) *insert-is-Un insert-not-empty less-not-refl*
mem-Collect-eq subset-eq subset-insertI)
then show *?thesis*
using ‹*i* < *length xs*› *linorder-le-less-linear* **by** *blast*
qed

lemma *patsearch-idx-start-fail*:
assumes *patsearch-idx-start xs ys* = *length xs*
and *i* < *length xs*
shows *prefix* (*xs* ! *i*) (*length ys*) < *ys*
proof −
have *i* < *patsearch-idx-start xs ys*
using *assms*(*1*) *assms*(*2*) **by** *argo*
then show *?thesis*
using *patsearch-idx-start-less* **by** *blast*
qed

lemma *patsearch-idx-start-success*:
assumes *sorted xs*
and *patsearch-idx-start xs ys* ≤ *i*
and *i* < *length xs*
shows *ys* ≤ *prefix* (*xs* ! *i*) (*length ys*)
proof −
let *?X* = {*i*. *i* < *length xs* ∧ *ys* ≤ *prefix* (*xs* ! *i*) (*length ys*)}

```
  have P1: ∀ x ∈ ?X. x < length xs
    by blast
  have P2: finite ?X
    by auto

  let ?i = patsearch-idx-start xs ys

  have ?X ≠ {}
    using assms(2) assms(3) by force
  hence ys ≤ take (length ys) (xs ! ?i)
    by (metis (mono-tags, lifting) Min-in P2 Un-insert-right assms(2,3) finite.insertI insert-iff
                                   linorder-not-le mem-Collect-eq sup-bot-right)
  moreover
  from sorted-nth-mono[OF assms]
  have xs ! ?i ≤ xs ! i .
  hence take (length ys) (xs ! ?i) ≤ take (length ys) (xs ! i)
    using le-take-n by blast
  ultimately show ?thesis
    using order-trans by blast
qed

lemma patsearch-idx-start-nil:
  patsearch-idx-start xs [] = 0
  using not-less-Nil patsearch-idx-start-less by blast

lemma patsearch-idx-start-upper2:
  assumes ys ≤ prefix (xs ! i) (length ys)
shows patsearch-idx-start xs ys ≤ i
  using assms linorder-not-less by fastforce

lemma patsearch-idx-start-lower:
  assumes sorted xs
  and     i < length xs
  and     prefix (xs ! i) (length ys) < ys
shows i < patsearch-idx-start xs ys
proof (rule ccontr)
  assume ¬ i < patsearch-idx-start xs ys
  hence patsearch-idx-start xs ys ≤ i
    using not-le-imp-less by blast
  with patsearch-idx-start-success[OF assms(1) - assms(2)] assms(3)
  show False
    using assms(2) dual-order.strict-trans2 linorder-not-less by blast
qed
```

lemma *patsearch-idx-start-card*:
assumes *sorted xs*
shows *patsearch-idx-start xs ys = card* $\{i.\ i < length\ xs \wedge prefix\ (xs\ !\ i)\ (length\ ys) < ys\}$
proof −
let *?A* $= \{i.\ i < length\ xs \wedge prefix\ (xs\ !\ i)\ (length\ ys) < ys\}$
let *?i = patsearch-idx-start xs ys*

have *A*: *?i* $\leq$ *length xs*
using *patsearch-idx-start-upper* **by** *blast*

have *B*: $\forall\, i < ?i.$ *prefix* $(xs\ !\ i)$ $(length\ ys) < ys$
using *patsearch-idx-start-less* **by** *blast*

have *C*: $\forall\, i < length\ xs.\ ?i \leq i \longrightarrow ys \leq prefix\ (xs\ !\ i)\ (length\ ys)$
using *assms patsearch-idx-start-success* **by** *blast*

have $\{0..<?i\} = ?A$
proof *safe*
fix *x*
assume $x \in \{0..<?i\}$
with *A*
show $x <$ *length xs*
by (*meson atLeastLessThan-iff order-less-le-trans*)
next
fix *x*
assume $x \in \{0..<?i\}$
with *B*
show *prefix* $(xs\ !\ x)$ $(length\ ys) < ys$
using *atLeastLessThan-iff* **by** *blast*
next
fix *x*
assume $x <$ *length xs prefix* $(xs\ !\ x)$ $(length\ ys) < ys$
with *C*
show $x \in \{0..<?i\}$
by (*meson atLeastLessThan-iff bot-nat-0.extremum linorder-not-less*)
qed
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *card-upt*)
qed

lemma *patsearch-idx-startpoint*:
assumes *sorted xs*
and $k \leq$ *length xs*
and $\forall\, i < k.$ *prefix* $(xs\ !\ i)$ $(length\ ys) < ys$

```
  and     ∀ i < length xs. k ≤ i ⟶ ys ≤ prefix (xs ! i) (length ys)
shows patsearch-idx-start xs ys = k
proof −
  let ?A = {i. i < length xs ∧ prefix (xs ! i) (length ys) < ys}
  have {0..<k} = ?A
  proof safe
    fix x
    assume x ∈ {0..<k}
    with assms(2)
    show x < length xs
      by auto
  next
    fix x
    assume x ∈ {0..<k}
    with assms(3)
    show prefix (xs ! x) (length ys) < ys
      by auto
  next
    fix x
    assume x < length xs prefix (xs ! x) (length ys) < ys
    with assms(4)
    show x ∈ {0..<k}
      by auto
  qed
  with patsearch-idx-start-card[OF assms(1)]
  show ?thesis
    by (metis card-upt)
qed
```

A.2.4 Ending Index for Indexed Pattern Search

We define a function that returns the one more than the last location of that block. Specifically, we find the last location in the sorted suffix list of S that has a prefix which is less than or equal to the pattern P, up to the length P, and add 1 to it. That is, the comparison is only between P and the prefix which is at most length P as well.

Note that this value is exclusive as there can be the case that every prefix of the suffixes are greater than the pattern.

```
abbreviation patsearch-idx-end :: 'a :: linorder list list ⇒ 'a list ⇒ nat
  where
patsearch-idx-end xs ys ≡
  Min ({i. i < length xs ∧ ys < prefix (xs ! i) (length ys)} ∪ {length xs})
```

lemma *patsearch-idx-end-nil*:
  patsearch-idx-end xs [] = *length xs*
  by *simp*

lemma *patsearch-idx-end-upper*:
  shows *patsearch-idx-end xs ys* ≤ *length xs*
  by *force*

lemma *patsearch-idx-end-less*:
  assumes *i* < *patsearch-idx-end xs ys*
  shows *prefix* (*xs* ! *i*) (*length ys*) ≤ *ys*
proof −
  let *?X* = {*i*. *i* < *length xs* ∧ *ys* < *prefix* (*xs* ! *i*) (*length ys*)}

  have *i* < *length xs*
    using *assms order-less-le-trans patsearch-idx-end-upper* **by** *blast*

  have *i* ∉ *?X*
    by (*metis* (*no-types*, *lifting*) *Min-gr-iff assms finite.insertI finite-nat-set-iff-bounded*
                inf-sup-aci(*5*) *insert-is-Un insert-not-empty less-not-refl*
                mem-Collect-eq subset-eq subset-insertI)
  then show *?thesis*
    using ‹*i* < *length xs*› *linorder-le-less-linear* **by** *blast*
qed

lemma *patsearch-idx-end-fail*:
  assumes *patsearch-idx-end xs ys* = *length xs*
  and *i* < *length xs*
shows *prefix* (*xs* ! *i*) (*length ys*) ≤ *ys*
proof −
  have *i* < *patsearch-idx-end xs ys*
    using *assms*(*1*) *assms*(*2*) **by** *argo*
  then show *?thesis*
    using *patsearch-idx-end-less* **by** *blast*
qed

lemma *patsearch-idx-end-success*:
  assumes *sorted xs*
  and *patsearch-idx-end xs ys* ≤ *i*
  and *i* < *length xs*
shows *ys* < *prefix* (*xs* ! *i*) (*length ys*)
proof −
  let *?X* = {*i*. *i* < *length xs* ∧ *ys* < *prefix* (*xs* ! *i*) (*length ys*)}
  have *P1*: ∀ *x* ∈ *?X*. *x* < *length xs*

```
    by blast
  have P2: finite ?X
    by auto

  let ?i = patsearch-idx-end xs ys

  have ?X ≠ {}
    using assms(2) assms(3) by force
  hence ys < take (length ys) (xs ! ?i)
    by (metis (mono-tags, lifting) Min-in P2 Un-insert-right assms(2,3) finite.insertI insert-iff
                                   linorder-not-le mem-Collect-eq sup-bot-right)
  moreover
  from sorted-nth-mono[OF assms]
  have xs ! ?i ≤ xs ! i .
  hence take (length ys) (xs ! ?i) ≤ take (length ys) (xs ! i)
    using le-take-n by blast
  ultimately show ?thesis
    by order
qed

lemma patsearch-idx-end-upper2:
  assumes ys < prefix (xs ! i) (length ys)
shows patsearch-idx-end xs ys ≤ i
  using assms linorder-not-less by fastforce

lemma patsearch-idx-end-lower:
  assumes sorted xs
  and     i < length xs
  and     prefix (xs ! i) (length ys) ≤ ys
shows i < patsearch-idx-end xs ys
proof (rule ccontr)
  assume ¬ i < patsearch-idx-end xs ys
  hence patsearch-idx-end xs ys ≤ i
    using not-le-imp-less by blast
  with patsearch-idx-end-success[OF assms(1) - assms(2)] assms(3)
  show False
    using assms(2) dual-order.strict-trans2 linorder-not-less by blast
qed

lemma patsearch-idx-end-card:
  assumes sorted xs
  shows patsearch-idx-end xs ys = card {i. i < length xs ∧ prefix (xs ! i) (length ys) ≤ ys}
proof −
  let ?A = {i. i < length xs ∧ prefix (xs ! i) (length ys) ≤ ys}
```

```
  let ?i = patsearch-idx-end xs ys

  have A: ?i ≤ length xs
    using patsearch-idx-end-upper by blast

  have B: ∀ i < ?i. prefix (xs ! i) (length ys) ≤ ys
    using patsearch-idx-end-less by blast

  have C: ∀ i < length xs. ?i ≤ i ⟶ ys < prefix (xs ! i) (length ys)
    using assms patsearch-idx-end-success by blast

  have {0..<?i} = ?A
  proof safe
    fix x
    assume x ∈ {0..<?i}
    with A
    show x < length xs
      by (meson atLeastLessThan-iff order-less-le-trans)
  next
    fix x
    assume x ∈ {0..<?i}
    with B
    show prefix (xs ! x) (length ys) ≤ ys
      using atLeastLessThan-iff by blast
  next
    fix x
    assume x < length xs prefix (xs ! x) (length ys) ≤ ys
    with C
    show x ∈ {0..<?i}
      by (meson atLeastLessThan-iff bot-nat-0.extremum linorder-not-less)
  qed
  then show ?thesis
    by (metis (no-types, lifting) card-upt)
qed

lemma patsearch-idx-endpoint:
  assumes sorted xs
  and     k ≤ length xs
  and     ∀ i < k. prefix (xs ! i) (length ys) ≤ ys
  and     ∀ i < length xs. k ≤ i ⟶ ys < prefix (xs ! i) (length ys)
shows patsearch-idx-end xs ys = k
proof −
  let ?A = {i. i < length xs ∧ prefix (xs ! i) (length ys) ≤ ys}
  have {0..<k} = ?A
```

```
  proof safe
    fix x
    assume x ∈ {0..<k}
    with assms(2)
    show x < length xs
      by auto
  next
    fix x
    assume x ∈ {0..<k}
    with assms(3)
    show prefix (xs ! x) (length ys) ≤ ys
      by auto
  next
    fix x
    assume x < length xs prefix (xs ! x) (length ys) ≤ ys
    with assms(4)
    show x ∈ {0..<k}
      by auto
  qed
  with patsearch-idx-end-card[OF assms(1)]
  show ?thesis
    by (metis card-upt)
qed
```

A.2.5 Pattern Search with Sorted List Index

```
lemma in-sorted-patsearch-range1:
  assumes sorted xs
  and     patsearch-idx-start xs ys ≤ i
  and     i < patsearch-idx-end xs ys
shows prefix (xs ! i) (length ys) = ys
proof −
  have xs ≠ []
    using assms(3) by force
  hence A: i < length xs
    using assms(3) order-less-le-trans patsearch-idx-end-upper by blast

  from patsearch-idx-start-success[OF assms(1) assms(2) A] assms(2) A
  have ys ≤ prefix (xs ! i) (length ys)
    using order.strict-trans1 by blast
  with patsearch-idx-end-less[OF assms(3)]
  show prefix (xs ! i) (length ys) = ys
    by order
qed
```

lemma *in-sorted-patsearch-range2*:
  assumes *sorted xs*
  and *i < length xs*
  and *prefix (xs ! i) (length ys) = ys*
shows *patsearch-idx-start xs ys ≤ i ∧ i < patsearch-idx-end xs ys*
proof
  let *?X = {i. i < length xs ∧ ys < take (length ys) (xs ! i)}*

  show *i < patsearch-idx-end xs ys*
  proof (*cases ?X = {}*)
    assume *?X ≠ {}*
    moreover
    have *i ∉ ?X*
      by (*simp add*: *assms*(*3*))
    have *∀j ≤ i. j ∉ ?X*
      by (*metis* (*no-types*, *lifting*) *assms leD le-take-n mem-Collect-eq sorted-iff-nth-mono*)
    ultimately have *Min ?X > i*
      using *not-le-imp-less* **by** *force*
    then show *?thesis*
      using ‹*?X ≠ {}*› **by** *force*
  next
    assume *?X = {}*
    hence *patsearch-idx-end xs ys = length xs*
      by (*metis Min-singleton sup-bot-left*)
    then show *?thesis*
      using *assms*(*2*) **by** *presburger*
  qed
next
  let *?X = {i. i < length xs ∧ ys ≤ take (length ys) (xs ! i)}*
  have *finite ?X*
    by *force*

  have *ys ≤ prefix (xs ! i) (length ys)*
    by (*simp add*: *assms*(*3*))
  hence *i ∈ ?X*
    using *assms*(*2*) **by** *blast*
  hence *?X ≠ {}*
    by *blast*
  hence *patsearch-idx-start xs ys = min (Min ?X) (length xs)*
    by *simp*
  hence *patsearch-idx-start xs ys = Min ?X*
    using ‹*finite ?X*› ‹*?X ≠ {}*›
    by (*metis* (*no-types*, *lifting*) *Min-eq-iff mem-Collect-eq min.strict-order-iff*)

then show *patsearch-idx-start xs ys* $\leq i$
using ‹$i \in ?X$› **by** *auto*
qed

theorem *in-sorted-patsearch-range*:
assumes *sorted xs*
shows $i <$ *length xs* $\wedge$ *prefix* (*xs* ! i) (*length ys*) = *ys*
$\longleftrightarrow$ *patsearch-idx-start xs ys* $\leq i \wedge i <$ *patsearch-idx-end xs ys*
proof
assume $i <$ *length xs* $\wedge$ *prefix* (*xs* ! i) (*length ys*) = *ys*
then show *patsearch-idx-start xs ys* $\leq i \wedge i <$ *patsearch-idx-end xs ys*
using *assms in-sorted-patsearch-range2* **by** *blast*
next
assume *A*: *patsearch-idx-start xs ys* $\leq i \wedge i <$ *patsearch-idx-end xs ys*
hence *prefix* (*xs* ! i) (*length ys*) = *ys*
using *assms in-sorted-patsearch-range1* **by** *blast*
moreover
have $i <$ *length xs*
using *A order-less-le-trans patsearch-idx-end-upper* **by** *blast*
ultimately show $i <$ *length xs* $\wedge$ *prefix* (*xs* ! i) (*length ys*) = *ys*
by *blast*
qed

end
theory *Binary-Search*
imports *Main*
begin

A.3 Binary Search

A.3.1 Termination Cases

lemma *binary-search-term1*:
$l < r \Longrightarrow r -$ *Suc* ((*l* + *r*) *div 2*) $< r - l$
by (*induct l arbitrary*: *r*; *simp*)

lemma *binary-search-term2*:
(*l* :: *nat*) $< r \Longrightarrow$ (*l* + *r*) *div 2* $- l < r - l$
by (*induct l arbitrary*: *r*; *simp*)

A.3.2 Leftmost Definition

Find the leftmost matching value.

function *binary-search-l* :: *'a::ord list* $\Rightarrow$ *'a::ord* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat*

where
$l < r \Longrightarrow$ *binary-search-l xs v l r* =
(*let m* = (*l* + *r*) *div 2*
in
(*if xs* ! *m* < *v then binary-search-l xs v* (*Suc m*) *r*
else binary-search-l xs v l m)) |
¬ *l* < *r* ⟹ *binary-search-l xs v l r* = *l*
by *fastforce*+

termination
by (*relation measure* (λ(*s*, *ps*, *l*, *r*). *r* − *l*);
clarsimp simp: *binary-search-term1 binary-search-term2*)

A.3.3 Rightmost Definition

Find the rightmost matching value.

function *binary-search-r* :: *'a*::*ord list* ⇒ *'a*::*ord* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
$l < r \Longrightarrow$ *binary-search-r xs v l r* =
(*let m* = (*l* + *r*) *div 2*
in
(*if xs* ! *m* > *v then binary-search-r xs v l m*
else binary-search-r xs v (*Suc m*) *r*)) |
¬ *l* < *r* ⟹ *binary-search-r xs v l r* = *r*
by *fastforce*+

termination
by (*relation measure* (λ(*s*, *ps*, *l*, *r*). *r* − *l*);
clarsimp simp: *binary-search-term1 binary-search-term2*)

end
theory *Binary-Search-Verification*
imports *Binary-Search*
BurrowsWheeler.Count-Util
begin

A.4 Verification of Binary Search

A.4.1 Leftmost Binary Search

A.4.1.1 Bound Properties

lemma *binary-search-l-upper*:
⟦*sorted xs*; *l* ≤ *r*; *r* ≤ *length xs*⟧ ⟹ *binary-search-l xs v l r* ≤ *r*

```
proof (induct rule: binary-search-l.induct[of - xs v l r])
  case (1 l r xs v)
  note IH = this

  let ?m = (l + r) div 2

  have ?m = ?m
    by simp

  have ?m < r
    using IH(1) by linarith
  hence Suc ?m ≤ r
    by simp

  have ?m < length xs
    using IH(6) ‹?m < r› dual-order.strict-trans1 by blast
  hence ?m ≤ length xs
    by simp

  have l ≤ ?m
    by (simp add: IH(5) less-eq-div-iff-mult-less-eq)

  note IH1 = IH(2)[OF ‹?m = ?m› - IH(4) ‹Suc ?m ≤ r› IH(6)]
  moreover
  note IH2 = IH(3)[OF ‹?m = ?m› - IH(4) ‹l ≤ ?m› ‹?m ≤ length xs›]
  moreover
  have xs ! ?m < v ∨ ¬ xs ! ?m < v
    by simp
  ultimately show ?case
    using ‹Suc ?m ≤ r› by fastforce
next
  case (2 l r xs v)
  then show ?case
    by simp
qed

lemma binary-search-l-lower:
  ⟦sorted xs; l ≤ r; r ≤ length xs⟧ ⟹ l ≤ binary-search-l xs v l r
proof (induct rule: binary-search-l.induct[of - xs v l r])
  case (1 l r xs v)
  note IH = this

  let ?m = (l + r) div 2
```

```
  have ?m = ?m
    by simp

  have ?m < r
    using IH(1) by linarith
  hence Suc ?m ≤ r
    by simp

  have ?m < length xs
    using IH(6) ‹?m < r› dual-order.strict-trans1 by blast
  hence ?m ≤ length xs
    by simp

  have l ≤ ?m
    by (simp add: IH(5) less-eq-div-iff-mult-less-eq)

  note IH1 = IH(2)[OF ‹?m = ?m› - IH(4) ‹Suc ?m ≤ r› IH(6)]
  moreover
  note IH2 = IH(3)[OF ‹?m = ?m› - IH(4) ‹l ≤ ?m› ‹?m ≤ length xs›]
  moreover
  have xs ! ?m < v ∨ ¬ xs ! ?m < v
    by simp
  ultimately show ?case
    using ‹Suc ?m ≤ r› by fastforce
next
  case (2 l r xs v)
  then show ?case
    by simp
qed
```

A.4.1.2 Ordering Properties

```
lemma binary-search-l-less:
  shows ⟦l ≤ r; i < binary-search-l xs v l r; l ≤ i; sorted xs; r ≤ length xs⟧ ⟹
        xs ! i < v
proof (induct rule: binary-search-l.induct[of - xs v l r])
  case (1 l r xs v)
  note IH = this

  let ?m = (l + r) div 2
  have ?m = ?m
    by simp

  have ?m < r
```

```
  using IH(1) by linarith
hence Suc ?m ≤ r
  by simp

have ?m < length xs
  using IH(8) ‹?m < r› dual-order.strict-trans1 by blast
hence ?m ≤ length xs
  by simp

have l ≤ ?m
  by (simp add: IH(4) less-eq-div-iff-mult-less-eq)

note IH1 = IH(2)[OF ‹?m = ?m› - ‹Suc ?m ≤ r› - - IH(7,8)]

note IH2 = IH(3)[OF ‹?m = ?m› - ‹l ≤ ?m› - IH(6,7) ‹?m ≤ length xs›]

have xs ! ?m < v ∨ ¬ xs ! ?m < v
  by blast
then show ?case
proof
  assume ¬ xs ! ((l + r) div 2) < v
  moreover
  have binary-search-l xs v l r = binary-search-l xs v l ?m
    using IH(1) calculation by auto
  hence i < binary-search-l xs v l ?m
    using IH(5) by presburger
  ultimately show ?case
    using IH2 by blast
next
  assume xs ! ?m < v
  moreover
  have binary-search-l xs v l r = binary-search-l xs v (Suc ?m) r
    using IH(1) calculation by auto
  hence i < binary-search-l xs v (Suc ?m) r
    using IH(5) by presburger
  ultimately have Suc ?m ≤ i ⟹ xs ! i < v
    using IH1 by blast
  moreover
  have ¬ Suc ?m ≤ i ⟹ xs ! i < v
  proof −
    assume ¬ Suc ?m ≤ i
    hence i < Suc ?m
      using leI by blast
    hence i ≤ ?m
```

```
      by simp
    with ‹xs ! ?m < v› sorted-nth-mono[OF IH(7) - ‹?m < length xs›]
    show ?thesis
      using order.strict-trans1 by blast
  qed
  ultimately show ?case
    by blast
 qed
next
 case (2 l r xs v)
 then show ?case
  by simp
qed

lemma binary-search-l-gre:
 shows ⟦l ≤ r; binary-search-l xs v l r ≤ i; i < r; sorted xs; r ≤ length xs⟧ ⟹
       v ≤ xs ! i
proof (induct rule: binary-search-l.induct[of - xs v l r])
 case (1 l r xs v)
 note IH = this
 let ?m = (l + r) div 2
 have ?m = ?m
  by simp

 have ?m < r
  using IH(1) by linarith
 hence Suc ?m ≤ r
  by simp

 have ?m < length xs
  using IH(8) ‹?m < r› dual-order.strict-trans1 by blast
 hence ?m ≤ length xs
  by simp

 have l ≤ ?m
  by (simp add: IH(4) less-eq-div-iff-mult-less-eq)

 note IH1 = IH(2)[OF ‹?m = ?m› - ‹Suc ?m ≤ r› - IH(6−8)]

 note IH2 = IH(3)[OF ‹?m = ?m› - ‹l ≤ ?m› - - IH(7) ‹?m ≤ length xs›]

 have xs ! ?m < v ∨ ¬ xs ! ?m < v
  by simp
 then show ?case
```

```
  proof
    assume xs ! ?m < v
    moreover
    have binary-search-l xs v l r = binary-search-l xs v (Suc ?m) r
      by (simp add: IH(1) calculation)
    hence binary-search-l xs v (Suc ?m) r ≤ i
      using IH(5) by presburger
    ultimately show ?case
      using IH1 by blast
  next
    assume ¬ xs ! ?m < v
    moreover
    have binary-search-l xs v l r = binary-search-l xs v l ?m
      by (simp add: IH(1) calculation)
    hence binary-search-l xs v l ?m ≤ i
      using IH(5) by presburger
    ultimately have i < ?m ⟹ ?case
      using IH2 by blast
    moreover
    have ¬ i < ?m ⟹ ?case
    proof −
      assume ¬ i < ?m
      hence ?m ≤ i
        using leI by blast
      moreover
      have i < length xs
        using IH(6) IH(8) dual-order.strict-trans1 by blast
      ultimately have xs ! ?m ≤ xs ! i
        using sorted-nth-mono[OF IH(7)] by blast
      with ‹¬ xs ! ?m < v› 
      show ?case
        by order
    qed
    ultimately show ?case
      by blast
  qed
next
  case (2 l r xs v)
  then show ?case
    by simp
qed

theorem binary-search-l-gen-correct:
  assumes sorted xs
```

and $l \leq r$
and $r \leq$ *length xs*
shows $(\forall\, i.\ l \leq i \wedge i < \textit{binary-search-l xs v l r} \longrightarrow \textit{xs} \,!\, i < v) \wedge$
$(\forall\, i.\ \textit{binary-search-l xs v l r} \leq i \wedge i < r \longrightarrow v \leq \textit{xs} \,!\, i)$
by (*meson assms binary-search-l-gre binary-search-l-less dual-order.strict-trans1 le0*
verit-comp-simplify1(*2*))

A.4.1.3 Correctness

lemma *binary-search-l-less-correct*:
assumes *sorted xs*
and $i <$ *binary-search-l xs v 0* (*length xs*)
shows *xs* ! $i < v$
using *binary-search-l-less*[*of 0 length xs i xs v*, *simplified*, *OF assms*(*2*) *assms*(*1*)] **by** *blast*

lemma *binary-search-l-gre-correct*:
assumes *sorted xs*
and *binary-search-l xs v 0* (*length xs*) $\leq i$
and $i <$ *length xs*
shows $v \leq$ *xs* ! i
using *binary-search-l-gre*[*of 0 length xs xs v i*, *simplified*, *OF assms*(*2*,*3*,*1*)] **by** *blast*

corollary *binary-search-l-correct*:
assumes *sorted xs*
shows $(\forall\, i.\ i <$ *binary-search-l xs v 0* (*length xs*) $\longrightarrow$ *xs* ! $i < v) \wedge$
$(\forall\, i.\ \textit{binary-search-l xs v 0}$ (*length xs*) $\leq i \wedge i <$ *length xs* $\longrightarrow v \leq$ *xs* ! $i)$
using *assms binary-search-l-gen-correct* **by** *blast*

A.4.1.4 Counting Properties

lemma *binary-search-l-card*:
assumes *sorted xs*
shows *binary-search-l xs v 0* (*length xs*) = *card* $\{i.\ i <$ *length xs* $\wedge$ *xs* ! $i < v\}$
proof −
from *binary-search-l-less-correct*[*OF assms*]
have *P*: $\forall\, i <$ *binary-search-l xs v 0* (*length xs*). *xs* ! $i < v$
by *blast*

from *binary-search-l-gre-correct*[*OF assms*]
have *Q*: $\forall\, i <$ *length xs*. *binary-search-l xs v 0* (*length xs*) $\leq i \longrightarrow v \leq$ *xs* ! i
by *blast*

have *R*: *binary-search-l xs v 0* (*length xs*) $\leq$ *length xs*
using *assms binary-search-l-upper* **by** *blast*

```
  have {0..<binary-search-l xs v 0 (length xs)} = {i. i < length xs ∧ xs ! i < v}
  proof (intro equalityI subsetI)
    fix x
    assume x ∈ {0..<binary-search-l xs v 0 (length xs)}
    hence x < binary-search-l xs v 0 (length xs)
      using atLeastLessThan-iff by blast
    with R
    have x < length xs
      by simp
    moreover
    have xs ! x < v
      using P ‹x < binary-search-l xs v 0 (length xs)› by blast
    ultimately show x ∈ {i. i < length xs ∧ xs ! i < v}
      by blast
  next
    fix x
    assume x ∈ {i. i < length xs ∧ xs ! i < v}
    hence x < length xs xs ! x < v
      by blast+
    hence x < binary-search-l xs v 0 (length xs)
      using Q by (meson linorder-not-le)
    then show x ∈ {0..<binary-search-l xs v 0 (length xs)}
      using atLeastLessThan-iff by blast
  qed
  then show ?thesis
    by (metis card-upt)
qed
```

A.4.2 Binary Search Rightmost

A.4.2.1 Bound Properties

```
lemma binary-search-r-upper:
  ⟦sorted xs; l ≤ r; r ≤ length xs⟧ ⟹ binary-search-r xs v l r ≤ r
proof (induct rule: binary-search-r.induct[of - xs v l r])
  case (1 l r xs v)
  note IH = this

  let ?m = (l + r) div 2

  have ?m = ?m
    by simp

  have ?m < r
    using IH(1) by linarith
```

```
  hence Suc ?m ≤ r
    by simp

  have ?m < length xs
    using IH(6) ‹?m < r› dual-order.strict-trans1 by blast
  hence ?m ≤ length xs
    by simp

  have l ≤ ?m
    by (simp add: IH(5) less-eq-div-iff-mult-less-eq)

  note IH1 = IH(2)[OF ‹?m = ?m› - IH(4) ‹l ≤ ?m› ‹?m ≤ length xs›]
  moreover
  note IH2 = IH(3)[OF ‹?m = ?m› - IH(4) ‹Suc ?m ≤ r› IH(6)]
  moreover
  have v < xs ! ?m  ∨ ¬ v < xs ! ?m
    by simp
  ultimately show ?case
    using ‹Suc ?m ≤ r› by fastforce
next
  case (2 l r xs v)
  then show ?case
    by simp
qed

lemma binary-search-r-lower:
  ⟦sorted xs; l ≤ r; r ≤ length xs⟧ ⟹ l ≤ binary-search-r xs v l r
proof (induct rule: binary-search-r.induct[of - xs v l r])
  case (1 l r xs v)
  note IH = this

  let ?m = (l + r) div 2

  have ?m = ?m
    by simp

  have ?m < r
    using IH(1) by linarith
  hence Suc ?m ≤ r
    by simp

  have ?m < length xs
    using IH(6) ‹?m < r› dual-order.strict-trans1 by blast
  hence ?m ≤ length xs
```

```
    by simp

  have l ≤ ?m
    by (simp add: IH(5) less-eq-div-iff-mult-less-eq)

  note IH1 = IH(2)[OF ‹?m = ?m› - IH(4) ‹l ≤ ?m› ‹?m ≤ length xs›]
  moreover
  note IH2 = IH(3)[OF ‹?m = ?m› - IH(4) ‹Suc ?m ≤ r› IH(6)]
  moreover
  have v < xs ! ?m  ∨ ¬ v < xs ! ?m
    by simp
  ultimately show ?case
    using ‹Suc ?m ≤ r› by fastforce
next
  case (2 l r xs v)
  then show ?case
    by simp
qed
```

A.4.2.2 Ordering Properties

```
lemma binary-search-r-less:
  shows ⟦l ≤ r; i < binary-search-r xs v l r; l ≤ i; sorted xs; r ≤ length xs⟧ ⟹
        xs ! i ≤ v
proof (induct rule: binary-search-r.induct[of - xs v l r])
  case (1 l r xs v)
  note IH = this

  let ?m = (l + r) div 2
  have ?m = ?m
    by simp

  have ?m < r
    using IH(1) by linarith
  hence Suc ?m ≤ r
    by simp

  have ?m < length xs
    using IH(8) ‹?m < r› dual-order.strict-trans1 by blast
  hence ?m ≤ length xs
    by simp

  have l ≤ ?m
    by (simp add: IH(4) less-eq-div-iff-mult-less-eq)
```

```
note IH1 = IH(2)[OF ‹?m = ?m› - ‹l ≤ ?m› - IH(6,7) ‹?m ≤ length xs›]

note IH2 = IH(3)[OF ‹?m = ?m› - ‹Suc ?m ≤ r› - - IH(7,8)]

have v < xs ! ?m ∨ ¬ v < xs ! ?m
  by blast
then show ?case
proof
  assume v < xs ! ?m
  moreover
  have binary-search-r xs v l r = binary-search-r xs v l ?m
    using IH(1) calculation by auto
  hence i < binary-search-r xs v l ?m
    using IH(5) by presburger
  ultimately show ?case
    using IH1 by blast
next
  assume ¬ v < xs ! ?m
  moreover
  have binary-search-r xs v l r = binary-search-r xs v (Suc ?m) r
    using IH(1) calculation by auto
  hence i < binary-search-r xs v (Suc ?m) r
    using IH(5) by presburger
  ultimately have Suc ?m ≤ i ⟹ ?case
    using IH2 by blast
  moreover
  have ¬ Suc ?m ≤ i ⟹ ?case
  proof −
    assume ¬ Suc ?m ≤ i
    hence i < Suc ?m
      using leI by blast
    hence i ≤ ?m
      by simp
    with sorted-nth-mono[OF IH(7) - ‹?m < length xs›]
    have xs ! i ≤ xs ! ?m
      by blast
    moreover
    have xs ! ?m ≤ v
      using ‹¬ v < xs ! ?m› leI by blast
    ultimately show ?case
      by order
  qed
  ultimately show ?case
```

```
    by blast
  qed
next
  case (2 l r xs v)
  then show ?case
    by simp
qed

lemma binary-search-r-gre:
  shows ⟦l ≤ r; binary-search-r xs v l r ≤ i; i < r; sorted xs; r ≤ length xs⟧ ⟹
        v < xs ! i
proof (induct rule: binary-search-r.induct[of - xs v l r])
  case (1 l r xs v)
  note IH = this
  let ?m = (l + r) div 2
  have ?m = ?m
    by simp

  have ?m < r
    using IH(1) by linarith
  hence Suc ?m ≤ r
    by simp

  have ?m < length xs
    using IH(8) ‹?m < r› dual-order.strict-trans1 by blast
  hence ?m ≤ length xs
    by simp

  have l ≤ ?m
    by (simp add: IH(4) less-eq-div-iff-mult-less-eq)

  note IH1 = IH(2)[OF ‹?m = ?m› - ‹l ≤ ?m› - - IH(7) ‹?m ≤ length xs›]

  note IH2 = IH(3)[OF ‹?m = ?m› - ‹Suc ?m ≤ r› - IH(6,7,8)]

  have v < xs ! ?m ∨ ¬ v < xs ! ?m
    by blast
  then show ?case
  proof
    assume ¬ v < xs ! ?m
    moreover
    have binary-search-r xs v l r = binary-search-r xs v (Suc ?m) r
      by (simp add: IH(1) calculation)
    hence binary-search-r xs v (Suc ?m) r ≤ i
```

```
      using IH(5) by presburger
    ultimately show ?case
      using IH2 by blast
  next
    assume v < xs ! ?m
    moreover
    have binary-search-r xs v l r = binary-search-r xs v l ?m
      by (simp add: IH(1) calculation)
    hence binary-search-r xs v l ?m ≤ i
      using IH(5) by presburger
    ultimately have i < ?m ⟹ ?case
      using IH1 by blast
    moreover
    have ¬ i < ?m ⟹ ?case
    proof −
      assume ¬ i < ?m
      hence ?m ≤ i
        using leI by blast
      moreover
      have i < length xs
        using IH(6) IH(8) dual-order.strict-trans1 by blast
      ultimately have xs ! ?m ≤ xs ! i
        using sorted-nth-mono[OF IH(7)] by blast
      with ‹v < xs ! ?m›
      show ?case
        by order
    qed
    ultimately show ?case
      by blast
  qed
next
  case (2 l r xs v)
  then show ?case
    by simp
qed

theorem binary-search-r-gen-correct:
  assumes sorted xs
  and     l ≤ r
  and     r ≤ length xs
shows (∀ i. l ≤ i ∧ i < binary-search-r xs v l r ⟶ xs ! i ≤ v) ∧
      (∀ i. binary-search-r xs v l r ≤ i ∧ i < r ⟶ v < xs ! i)
  by (meson assms binary-search-r-gre binary-search-r-less dual-order.strict-trans1 le0
        verit-comp-simplify1(2))
```

A.4.2.3 Correctness

corollary *binary-search-r-less-correct*:
  assumes *sorted xs*
  and $i < binary\text{-}search\text{-}r\ xs\ v\ 0\ (length\ xs)$
shows $xs\ !\ i \leq v$
  using *binary-search-r-less*[*of 0 length xs i xs v*, *simplified*, *OF assms*(*2*,*1*)] **by** *blast*

corollary *binary-search-r-gre-correct*:
  assumes *sorted xs*
  and $binary\text{-}search\text{-}r\ xs\ v\ 0\ (length\ xs) \leq i$
  and $i < length\ xs$
shows $v < xs\ !\ i$
  using *binary-search-r-gre*[*of 0 length xs xs v i*, *simplified*, *OF assms*(*2*,*3*,*1*)] **by** *blast*

corollary *binary-search-r-correct*:
  assumes *sorted xs*
shows $(\forall i.\ i < binary\text{-}search\text{-}r\ xs\ v\ 0\ (length\ xs) \longrightarrow xs\ !\ i \leq v) \wedge$
    $(\forall i.\ binary\text{-}search\text{-}r\ xs\ v\ 0\ (length\ xs) \leq i \wedge i < length\ xs \longrightarrow v < xs\ !\ i)$
  using *assms binary-search-r-gen-correct* **by** *blast*

A.4.2.4 Counting Properties

lemma *binary-search-r-card*:
  assumes *sorted xs*
  shows $binary\text{-}search\text{-}r\ xs\ v\ 0\ (length\ xs) = card\ \{i.\ i < length\ xs \wedge xs\ !\ i \leq v\}$
proof −
  let $?n = binary\text{-}search\text{-}r\ xs\ v\ 0\ (length\ xs)$

  from *binary-search-r-less-correct*[*OF assms*]
  have P: $\forall i < ?n.\ xs\ !\ i \leq v$
    by *blast*

  from *binary-search-r-gre-correct*[*OF assms*]
  have Q: $\forall i < length\ xs.\ ?n \leq i \longrightarrow v < xs\ !\ i$
    by *blast*

  have R: $?n \leq length\ xs$
    using *assms binary-search-r-upper* **by** *blast*

  have $\{0..<?n\} = \{i.\ i < length\ xs \wedge xs\ !\ i \leq v\}$
  proof (*intro equalityI subsetI*)
    fix x
    assume $x \in \{0..<?n\}$
    hence $x < ?n$

```
    using atLeastLessThan-iff by blast
  with P R
  show x ∈ {i. i < length xs ∧ xs ! i ≤ v}
    using dual-order.strict-trans1 by blast
 next
  fix x
  assume x ∈ {i. i < length xs ∧ xs ! i ≤ v}
  hence x < length xs xs ! x ≤ v
    by blast+
  hence x < ?n
    using Q by (metis leD leI)
  then show x ∈ {0..<?n}
    using atLeastLessThan-iff by blast
 qed
 then show ?thesis
  by (metis card-upt)
qed
```

A.4.3 Combined

```
lemma binary-search-range-empty:
 assumes sorted xs
 shows binary-search-l xs v 0 (length xs) = binary-search-r xs v 0 (length xs) ⟷
       v ∉ set xs
proof −
 have A: finite {i. i < length xs ∧ xs ! i ≤ v}
  by force

 have B: {i. i < length xs ∧ xs ! i < v} ⊆ {i. i < length xs ∧ xs ! i ≤ v}
  by force

 from binary-search-l-card[OF assms]
 have C: binary-search-l xs v 0 (length xs) = card {i. i < length xs ∧ xs ! i < v} .

 from binary-search-r-card[OF assms]
 have D: binary-search-r xs v 0 (length xs) = card {i. i < length xs ∧ xs ! i ≤ v} .

 show ?thesis
 proof
  assume binary-search-l xs v 0 (length xs) = binary-search-r xs v 0 (length xs)
  with C D
  have card {i. i < length xs ∧ xs ! i < v} = card {i. i < length xs ∧ xs ! i ≤ v}
   by simp
  hence {i. i < length xs ∧ xs ! i < v} = {i. i < length xs ∧ xs ! i ≤ v}
```

```
    using A B by (meson card-subset-eq)
  then show v ∉ set xs
    by (metis (mono-tags, lifting) add-eq-self-zero card-le-eq-card-less-pl-count-list
                                   notin-count)
 next
  assume v ∉ set xs
  hence {i. i < length xs ∧ xs ! i < v} = {i. i < length xs ∧ xs ! i ≤ v}
    using A B in-set-conv-nth by fastforce
  with C D
  show binary-search-l xs v 0 (length xs) = binary-search-r xs v 0 (length xs)
    by simp
 qed
qed

lemma binary-search-in-range:
 assumes sorted xs
 and     i < length xs
 shows binary-search-l xs v 0 (length xs) ≤ i ∧ i < binary-search-r xs v 0 (length xs) ⟷
       xs ! i = v
proof (intro iffI)
 assume A: xs ! i = v
 hence ¬ xs ! i < v
   by blast
 hence binary-search-l xs v 0 (length xs) ≤ i
   using assms binary-search-l-less-correct leI by blast
 moreover
 from A
 have ¬ v < xs ! i
   by blast
 hence i < binary-search-r xs v 0 (length xs)
   using binary-search-r-gre-correct[OF assms(1) - assms(2)] leI by blast
 ultimately show binary-search-l xs v 0 (length xs) ≤ i ∧ i < binary-search-r xs v 0 (length
xs)
   by blast
next
 assume binary-search-l xs v 0 (length xs) ≤ i ∧ i < binary-search-r xs v 0 (length xs)
 hence A: binary-search-l xs v 0 (length xs) ≤ i i < binary-search-r xs v 0 (length xs)
   by blast+
 hence v ≤ xs ! i
   using binary-search-l-gre-correct[OF assms(1) - assms(2)] by blast
 moreover
 have xs ! i ≤ v
   using binary-search-r-less-correct[OF assms(1) A(2)] by blast
 ultimately show xs ! i = v
```

```
    by simp
qed

end
theory SA-Search
  imports ../pattern-search/Pattern-Search-Idx
          BurrowsWheeler.SA-Count
begin
```

A.5 Indexed Pattern Search on Suffix Arrays

A.5.1 Sorted Sequences Properties

```
lemma sorted-wrt-list-slice:
  assumes sorted-wrt P xs
  shows sorted-wrt P (list-slice xs i j)
proof (intro sorted-wrtI)
  fix i' j'
  assume i' < j' j' < length (list-slice xs i j)

  have list-slice xs i j ! i' = xs ! (i + i')
    using ‹i' < j'› ‹j' < length (list-slice xs i j)› nth-list-slice order.strict-trans by blast
  moreover
  have list-slice xs i j ! j' = xs ! (i + j')
    using ‹j' < length (list-slice xs i j)› nth-list-slice by blast
  moreover
  {
    have i + i' < i + j'
      by (simp add: ‹i' < j'›)
    moreover
    have j ≤ length xs ∨ j > length xs
      by linarith
    hence i + j' < length xs
    proof
      assume j ≤ length xs
      with length-list-slice[of xs i j]
      have length (list-slice xs i j) = j − i
        by presburger
      hence j' < j − i
        using ‹j' < length (list-slice xs i j)› by presburger
      then show ?thesis
        using ‹j ≤ length xs› by auto
    next
      assume length xs < j
```

```
    with length-list-slice[of xs i j]
    have length (list-slice xs i j) = length xs − i
      by presburger
    hence j′ < length xs − i
      using ‹j′ < length (list-slice xs i j)› by presburger
    then show ?thesis
      by simp
  qed
  ultimately have P (xs ! (i + i′)) (xs ! (i + j′))
    using assms(1) sorted-wrt-iff-nth-less by blast
 }
 ultimately show P (list-slice xs i j ! i′) (list-slice xs i j ! j′)
  by simp
qed

lemma sorted-lists-starting-symbol-range:
 assumes sorted xs
 and     i ≤ j
 and     j < length xs
 and     xs ! i = v # as
 and     xs ! j = v # bs
 and     i ≤ k
 and     k ≤ j
shows ∃ cs. xs ! k = v # cs
proof −
 have i = k ∨ (i < k ∧ k < j) ∨ k = j
  using assms(6,7) le-neq-implies-less by blast
 moreover
 have i = k ⟹ ?thesis
  using assms(4) by auto
 moreover
 have k = j ⟹ ?thesis
  by (simp add: assms(5))
 moreover
 have i < k ∧ k < j ⟹ ?thesis
 proof −
  assume i < k ∧ k < j
  hence i < k k < j
   by blast+

  have k < length xs
   using assms(3) assms(7) dual-order.strict-trans2 by blast
  with sorted-wrt-nth-less[OF assms(1) ‹i < k›]
  have xs ! i ≤ xs ! k .
```

```
    hence v # as ≤ xs ! k
      by (simp add: assms(4))

    from sorted-wrt-nth-less[OF assms(1) ‹k < j› ‹j < length xs›]
    have xs ! k ≤ xs ! j .
    hence xs ! k ≤ v # bs
      using assms(5) by fastforce

    have ∃ c cs. xs ! k = c # cs
      by (metis ‹v # as ≤ xs ! k› le-Nil neq-Nil-conv)
    then obtain c cs where
      xs ! k = c # cs
      by blast
    moreover
    {
      have v ≤ c
        using ‹xs ! i ≤ xs ! k› assms(4) calculation by fastforce
      moreover
      have c ≤ v
        using ‹xs ! k = c # cs› ‹xs ! k ≤ xs ! j› assms(5) by force
      ultimately have c = v
        by simp
    }
    ultimately show ?thesis
      by simp
  qed
  ultimately show ?thesis
    by blast
qed

lemma sorted-suffixes-starting-symbol-range:
  assumes set xs ⊆ {0..<length s}
  and     strict-sorted (map (suffix s) xs)
  and     i ≤ j
  and     j < length xs
  and     s ! (xs ! i) = v
  and     s ! (xs ! j) = v
  and     i ≤ k
  and     k ≤ j
shows s ! (xs ! k) = v
proof −
  have A1: sorted (map (suffix s) xs)
    using assms(2) strict-sorted-iff by blast
  have A2: j < length (map (suffix s) xs)
```

```
    by (simp add: assms(4))

  have ∃as. map (suffix s) xs ! i = v # as
    by (metis assms(1,3,4,5) atLeastLessThan-iff in-mono nth-map nth-mem order-le-less-trans
        suffix-cons-Suc)
  then obtain as where
    A3: map (suffix s) xs ! i = v # as
    by blast

  have ∃bs. map (suffix s) xs ! j = v # bs
    by (metis A1 A2 A3 assms(3,4,6) drop-all linorder-less-linear linorder-not-le not-less-Nil
        nth-map sorted-iff-nth-mono suffix-cons-Suc)
  then obtain bs where
    A4: map (suffix s) xs ! j = v # bs
    by blast
  from sorted-lists-starting-symbol-range[OF A1 assms(3) A2 A3 A4 assms(7,8)]
  show ?thesis
    using assms(4) assms(8) nth-via-drop by fastforce
qed

lemma sorted-lists-starting-symbol-tl-range:
  assumes sorted xs
  and     i ≤ j
  and     j < length xs
  and     xs ! i = v # as
  and     xs ! j = v # bs
shows sorted (map tl (list-slice xs i (Suc j)))
proof (intro sorted-wrt-mapI)
  fix i' j'
  assume A: i' < j' j' < length (list-slice xs i (Suc j))

  have xs ! i = list-slice xs i (Suc j) ! 0
    by (metis A(2) add.right-neutral bot.extremum-strict bot-nat-def length-greater-0-conv
        list.size(3) nth-list-slice)

  have sorted (list-slice xs i (Suc j))
    using assms(1) sorted-wrt-list-slice by blast
  with sorted-wrt-nth-less[OF - A]
  have list-slice xs i (Suc j) ! i' ≤ list-slice xs i (Suc j) ! j'
    by blast
  moreover
  {
    have i ≤ i + i'
      by simp
```

```
    moreover
    have i + i' ≤ j
      using A(1,2) by force
    ultimately have ∃cs. xs ! (i + i') = v # cs
      using sorted-lists-starting-symbol-range[OF assms] by simp
    moreover
    have list-slice xs i (Suc j) ! i' = xs ! (i + i')
      using A(1) A(2) nth-list-slice order.strict-trans by blast
    ultimately have ∃cs. list-slice xs i (Suc j) ! i' = v # cs
      by simp
  }
  moreover
  {
    have i ≤ i + j'
      by simp
    moreover
    have i + j' ≤ j
      using A(2) by force
    ultimately have ∃cs. xs ! (i + j') = v # cs
      using sorted-lists-starting-symbol-range[OF assms] by simp
    moreover
    have list-slice xs i (Suc j) ! j' = xs ! (i + j')
      using A(2) nth-list-slice by blast
    ultimately have ∃cs. list-slice xs i (Suc j) ! j' = v # cs
      by simp
  }
  ultimately show tl (list-slice xs i (Suc j) ! i') ≤ tl (list-slice xs i (Suc j) ! j')
    by force
qed

lemma strict-sorted-lists-starting-symbol-tl-range:
  assumes strict-sorted xs
  and     i ≤ j
  and     j < length xs
  and     xs ! i = v # as
  and     xs ! j = v # bs
shows strict-sorted (map tl (list-slice xs i (Suc j)))
proof −
  have sorted xs
    by (simp add: assms(1) strict-sorted-imp-sorted)
  hence sorted (map tl (list-slice xs i (Suc j)))
    using assms(2) assms(3) assms(4) assms(5) sorted-lists-starting-symbol-tl-range by blast
  moreover
  have distinct (map tl (list-slice xs i (Suc j)))
```

proof (*intro distinct-conv-nth*[*THEN iffD2*] *allI impI*)
fix i' j'
assume *A*: $i' < length\ (map\ tl\ (list\text{-}slice\ xs\ i\ (Suc\ j)))$
$j' < length\ (map\ tl\ (list\text{-}slice\ xs\ i\ (Suc\ j)))$
$i' \neq j'$

have *distinct* (*list-slice xs i* (*Suc j*))
using *assms*(*1*) *distinct-list-slice strict-sorted-iff* **by** *auto*
hence *list-slice xs i* (*Suc j*) ! $i' \neq$ *list-slice xs i* (*Suc j*) ! j'
by (*metis A*(*1*) *A*(*2*) *A*(*3*) *length-map nth-eq-iff-index-eq*)
moreover
from *sorted-lists-starting-symbol-range*[*OF* ‹*sorted xs*› *assms*(*2*−), *of i* + i']
have $\exists\, cs.\ xs\ !\ (i + i') = v\ \#\ cs$
using *A*(*1*) **by** *force*
moreover
from *sorted-lists-starting-symbol-range*[*OF* ‹*sorted xs*› *assms*(*2*−), *of i* + j']
have $\exists\, cs.\ xs\ !\ (i + j') = v\ \#\ cs$
using *A*(*2*) **by** *force*
ultimately show *map tl* (*list-slice xs i* (*Suc j*)) ! $i' \neq$ *map tl* (*list-slice xs i* (*Suc j*)) ! j'
by (*metis A*(*1*) *A*(*2*) *length-map list.sel*(*3*) *nth-list-slice nth-map*)
qed
ultimately show *?thesis*
by (*simp add*: *strict-sorted-iff*)
qed

A.5.2 Searching Suffix Arrays

context *Suffix-Array-General* **begin**

A.5.2.1 Symbol Range Properties

lemma *suffix-array-starting-symbol-range*:
assumes $i \leq j$
and $j < length\ s$
and $s\ !\ (sa\ s\ !\ i) = v$
and $s\ !\ (sa\ s\ !\ j) = v$
and $i \leq k$
and $k \leq j$
shows $s\ !\ (sa\ s\ !\ k) = v$
using *sorted-suffixes-starting-symbol-range*[*OF sa-subset-upt sa-g-sorted assms*(*1*) - *assms*(*3*−),
simplified sa-length,
OF assms(*2*)]
by *blast*

lemma *suffix-array-starting-symbol-tl-range*:

```
  assumes i ≤ j
  and     j < length s
  and     s ! (sa s ! i) = v
  and     s ! (sa s ! j) = v
shows strict-sorted (map (suffix s) (map Suc (list-slice (sa s) i (Suc j))))
proof (intro sorted-wrt-mapI)
  fix i' j'
  assume A: i' < j' j' < length (map Suc (list-slice (sa s) i (Suc j)))
  hence B: j' < length (list-slice (sa s) i (Suc j))
    by (metis length-map)

  {
    have suffix s (map Suc (list-slice (sa s) i (Suc j)) ! i')
          = suffix s (Suc (sa s ! (i + i')))
      by (metis A(1) A(2) length-map nth-list-slice nth-map order.strict-trans)
    moreover
    have suffix s (Suc (sa s ! (i + i'))) = tl (suffix s (sa s ! (i + i')))
      by (simp add: drop-Suc tl-drop)
    ultimately have suffix s (map Suc (list-slice (sa s) i (Suc j)) ! i')
                    = tl (suffix s (sa s ! (i + i')))
      by simp
  }
  moreover
  {
    have suffix s (map Suc (list-slice (sa s) i (Suc j)) ! j')
          = suffix s (Suc (sa s ! (i + j')))
      by (metis A(2) length-map nth-list-slice nth-map)
    moreover
    have suffix s (Suc (sa s ! (i + j'))) = tl (suffix s (sa s ! (i + j')))
      by (simp add: drop-Suc tl-drop)
    ultimately have suffix s (map Suc (list-slice (sa s) i (Suc j)) ! j')
                    = tl (suffix s (sa s ! (i + j')))
      by simp
  }
  moreover
  have P2: j < length (map (suffix s) (sa s))
    by (simp add: assms(2) sa-length)
  have ∃ as. suffix s (sa s ! i) = v # as
    by (metis Cons-nth-drop-Suc assms(1) assms(2) assms(3) order-le-less-trans sa-nth-ex)
  then obtain as where
    P3: map (suffix s) (sa s) ! i = v # as
    using P2 assms(1) by force
  have ∃ bs. suffix s (sa s ! j) = v # bs
    using assms(2) assms(4) sa-nth-ex suffix-cons-Suc by blast
```

then obtain *bs* **where**
P4: *map* (*suffix s*) (*sa s*) ! *j* = *v* # *bs*
using *assms*(*2*) *sa-length* **by** *force*
from *strict-sorted-lists-starting-symbol-tl-range*[*OF sa-g-sorted assms*(*1*) *P2 P3 P4*]
have *strict-sorted* (*map tl* (*list-slice* (*map* (*suffix s*) (*sa s*)) *i* (*Suc j*))) **.**
with *sorted-wrt-mapD*[*OF - A*(*1*), *of* (<) *tl list-slice* (*map* (*suffix s*) (*sa s*)) *i* (*Suc j*)] *B*
have *tl* (*list-slice* (*map* (*suffix s*) (*sa s*)) *i* (*Suc j*) ! *i'*)
< *tl* (*list-slice* (*map* (*suffix s*) (*sa s*)) *i* (*Suc j*) ! *j'*)
by *simp*
moreover
have *tl* (*map* (*suffix s*) (*list-slice* (*sa s*) *i* (*Suc j*)) ! *i'*)
= *tl* (*suffix s* (*sa s* ! (*i* + *i'*)))
by (*metis A*(*1*) *B nth-list-slice nth-map order.strict-trans*)
moreover
have *tl* (*map* (*suffix s*) (*list-slice* (*sa s*) *i* (*Suc j*)) ! *j'*)
= *tl* (*suffix s* (*sa s* ! (*i* + *j'*)))
by (*metis B nth-list-slice nth-map*)
ultimately show *suffix s* (*map Suc* (*list-slice* (*sa s*) *i* (*Suc j*)) ! *i'*)
< *suffix s* (*map Suc* (*list-slice* (*sa s*) *i* (*Suc j*)) ! *j'*)
by (*metis map-list-slice*)
qed

A.5.2.2 Pattern Search on Suffix Arrays Properties

lemma *in-sa-range*:
shows *i* < *length xs* ∧ *prefix* (*suffix xs* (*sa xs* ! *i*)) (*length ys*) = *ys*
⟷ *patsearch-idx-start* (*map* (*suffix xs*) (*sa xs*)) *ys* ≤ *i* ∧
i < *patsearch-idx-end* (*map* (*suffix xs*) (*sa xs*)) *ys*
proof −
have *sorted* (*map* (*suffix xs*) (*sa xs*))
using *sa-g-sorted strict-sorted-imp-sorted* **by** *blast*
with *in-sorted-patsearch-range*[*of map* (*suffix xs*) (*sa xs*) *i ys*]
show *?thesis*
using *sa-length* **by** *auto*
qed

theorem *patsearch-set-eq-sa-range*:
patsearch-set xs ys =
(*nth* (*sa xs*)) ‘ {*patsearch-idx-start* (*map* (*suffix xs*) (*sa xs*)) *ys* ..<
patsearch-idx-end (*map* (*suffix xs*) (*sa xs*)) *ys*}
(**is** *patsearch-set xs ys* = (*nth* (*sa xs*)) ‘ {*?s*..<*?e*})
proof *safe*
fix *x*
assume *x* ∈ *patsearch-set xs ys*

```
  hence x < length xs list-slice xs x (x + length ys) = ys
    by (simp add: patsearch-set-def)+
  with list-slice-to-suffix[of xs x x + length ys]
  have prefix (suffix xs x) (length ys) = ys
    by simp
  moreover
  from ex-sa-nth[OF ‹x < -›]
  obtain i where
    i < length xs
    sa xs ! i = x
    by blast
  ultimately show x ∈ (nth (sa xs)) ` {?s..<?e}
    using in-sa-range[of i xs ys, THEN iffD1] atLeastLessThan-iff by blast
next
  fix x
  assume x ∈ {?s..<?e}
  with in-sa-range[of x xs ys, THEN iffD2]
  have x < length xs prefix (suffix xs (sa xs ! x)) (length ys) = ys
    using atLeastLessThan-iff by blast+

  have sa xs ! x < length xs
    using ‹x < length xs› sa-nth-ex by auto
  moreover
  from list-slice-to-suffix[of xs sa xs ! x (sa xs ! x) + length ys]
  have list-slice xs (sa xs ! x) (sa xs ! x + length ys) = ys
    by (simp add: ‹prefix (suffix xs (sa xs ! x)) (length ys) = ys›)
  ultimately show sa xs ! x ∈ patsearch-set xs ys
    by (simp add: patsearch-set-def)
qed

lemma patsearch-idx-start-singleton-strict-upper:
  assumes a ∈ set s
  shows patsearch-idx-start (map (suffix s) (sa s)) [a] < length s
proof −
  let ?xs = map (suffix s) (sa s)
  let ?X = {i. i < length ?xs ∧ [a] ≤ prefix (?xs ! i) (length [a])}

  have finite ?X
    using finite-nat-set-iff-bounded by blast

  have P: ∀ x ∈ ?X. x < length ?xs
    by blast

  have ∃ i < length s. s ! i = a
```

```
    by (meson assms in-set-conv-nth)
  then obtain i where
    i < length s s ! i = a
    by blast

  from ex-sa-nth[OF ‹i < length s›]
  obtain k where
    k < length s sa s ! k = i
    by blast
  hence ∃ as. suffix s (sa s ! k) = a # as
    using ‹i < length s› ‹s ! i = a› suffix-cons-Suc by blast
  hence [a] ≤ prefix (?xs ! k) (length [a])
    using ‹k < length s› sa-length by force
  hence k ∈ ?X
    by (simp add: ‹k < length s› sa-length)
  hence ?X ≠ {}
    by blast
  hence patsearch-idx-start (map (suffix s) (sa s)) [a] = Min ?X
    by (metis (no-types, lifting) Min.union Min-in Min-singleton ‹finite ?X› P finite.emptyI
                                  finite.insertI insert-not-empty min.absorb3)
  hence Min ?X ≤ k
    using Min-le ‹finite ?X› ‹k ∈ ?X› by blast
  then show ?thesis
    using ‹k < length s› ‹patsearch-idx-start ?xs [a] = Min ?X› by linarith
qed



lemma patsearch-idx-start-singleton:
  assumes a ∈ set s
  shows patsearch-idx-start (map (suffix s) (sa s)) [a] = card {i. i < length s ∧ s ! i < a}
      (is patsearch-idx-start ?xs [a] = card ?X)
proof −
  note sorted = sa-g-sorted[THEN strict-sorted-imp-sorted]

  have card ?X < length s
    using assms card-less-idx-upper-strict by fastforce
  hence card ?X < length ?xs
    using sa-length by fastforce

  have 0 < count-list s a
    by (simp add: assms count-in)
  with sa-suffix-nth[of s a 0] ‹card ?X < length s› ‹card ?X < length ?xs›
  have ∃ as. ?xs ! (card ?X) = a # as
    by fastforce
```

```
  hence [a] ≤ prefix (?xs ! (card ?X)) (length [a])
    by force
  with patsearch-idx-start-upper2
  have patsearch-idx-start ?xs [a] ≤ card ?X
    by blast

  show ?thesis
  proof (cases card ?X)
    case 0
    then show ?thesis
      using ‹patsearch-idx-start ?xs [a] ≤ card ?X› by linarith
  next
    case (Suc n)
    hence n < card ?X
      by simp
    with sa-suffix-nth-less[of n s a]
    have ∀ as. suffix s (sa s ! n) < a # as
      by fastforce
    moreover
    have n < length ?xs
      using ‹card ?X < length ?xs› ‹n < card ?X› order-less-trans by blast
    ultimately have ?xs ! n < [a]
      by simp
    hence ∃ b as. ?xs ! n = b # as ∧ b < a
      by (metis Cons-less-Cons ‹n < length ?xs› length-map not-less-Nil nth-map sa-length
sa-nth-ex
          suffix-cons-ex)
    hence prefix (?xs ! n) (length [a]) < [a]
      by force
    with patsearch-idx-start-lower[OF sorted ‹n < length ?xs›]
    have n < patsearch-idx-start ?xs [a]
      by blast
    with ‹patsearch-idx-start ?xs [a] ≤ card ?X› Suc
    show ?thesis
      by linarith
  qed
qed

lemma patsearch-idx-end-singleton:
  assumes a ∈ set s
  shows patsearch-idx-end (map (suffix s) (sa s)) [a] =
        card {i. i < length s ∧ s ! i < a} + count-list s a
       (is patsearch-idx-end ?xs [a] = card ?X + count-list s a)
proof −
```

note *sorted* = *sa-g-sorted*[*THEN strict-sorted-imp-sorted*]

have *A*: *card ?X* + *count-list s a* ≤ *length s*
using *card-pl-count-list-strict-upper* **by** *blast*
moreover
have ∃ *m*. *count-list s a* = *Suc m*
by (*simp add*: *assms count-in not0-implies-Suc*)
then obtain *m* **where**
count-list s a = *Suc m*
by *blast*
ultimately have *card ?X* + *m* < *length s*
by *linarith*
moreover
have *card ?X* + *m* < *length ?xs*
by (*simp add*: *calculation sa-length*)
ultimately have ∃ *as*. *?xs* ! (*card ?X* + *m*) = *a* # *as*
by (*simp add*: ‹*count-list s a* = *Suc m*› *sa-length sa-suffix-nth*[*of s a m*])
hence *prefix* (*?xs* ! (*card ?X* + *m*)) (*length* [*a*]) ≤ [*a*]
by *fastforce*
with *patsearch-idx-end-lower*[*OF sorted* ‹*card ?X* + *m* < *length ?xs*›]
have *P*: *card ?X* + *m* < *patsearch-idx-end ?xs* [*a*]
by *blast*

have *card ?X* + *count-list s a* = *length s* ∨ *card ?X* + *count-list s a* < *length s*
using *A antisym-conv1* **by** *blast*
then show *?thesis*
proof
assume *card ?X* + *count-list s a* < *length s*
with *sa-suffix-nth-gr*[*of s a count-list s a*]
have ∀ *as*. *a* # *as* < *?xs* ! (*card ?X* + *count-list s a*)
using *sa-length* **by** *fastforce*
moreover
have *card ?X* + *count-list s a* < *length ?xs*
by (*simp add*: ‹*card ?X* + *count-list s a* < *length s*› *sa-length*)
hence ∃ *b as*. *?xs* ! (*card ?X* + *count-list s a*) = *b* # *as*
using *sa-length sa-nth-ex suffix-cons-ex* **by** *fastforce*
then obtain *b as* **where**
?xs ! (*card ?X* + *count-list s a*) = *b* # *as*
by *blast*
ultimately have *a* < *b*
by *auto*
hence [*a*] < *prefix* (*?xs* ! (*card ?X* + *count-list s a*)) (*length* [*a*])
by (*simp add*: ‹*?xs* ! (*card ?X* + *count-list s a*) = *b* # *as*›)
with *patsearch-idx-end-upper2*

```
    have patsearch-idx-end ?xs [a] ≤ card ?X + count-list s a
      by blast
    then show ?thesis
      using P ‹count-list s a = Suc m› by linarith
  next
    assume card ?X + count-list s a = length s
    hence card ?X + count-list s a = length ?xs
      by (simp add: sa-length)
    then show ?thesis
      using P ‹count-list s a = Suc m› patsearch-idx-end-upper[of ?xs [a]] by linarith
  qed
qed

end


end
theory SA-Binary-Search-Verification
  imports SA-Search
          ../binary-search/Binary-Search-Verification
begin
```

A.6 Verification of Binary Search on Suffix Arrays

```
context Suffix-Array-General begin
```

A.6.1 Leftmost

```
lemma binary-search-sa-leftmost:
  binary-search-l (map (λi. prefix (suffix s i) (length ps)) (sa s)) ps 0 (length s) =
   patsearch-idx-start (map (suffix s) (sa s)) ps
proof −
  let ?xs = map (λi. prefix (suffix s i) (length ps)) (sa s)
  let ?ys = map (suffix s) (sa s)

  have sorted ?xs
  proof (intro sorted-wrt-mapI)
    fix i j
    assume i < j j < length (sa s)
    with sorted-wrt-mapD[OF sa-suffix-sorted]
    have suffix s (sa s ! i) ≤ suffix s (sa s ! j)
      by blast
    with le-take-n
    show prefix (suffix s (sa s ! i)) (length ps) ≤ prefix (suffix s (sa s ! j)) (length ps)
```

```
    by blast
  qed

  have length ?xs = length ?ys
    by simp

  from binary-search-l-card[OF ‹sorted ?xs›]
  have binary-search-l ?xs ps 0 (length s) = card {i. i < length ?xs ∧ ?xs ! i < ps}
    by (simp add: sa-length)
  moreover
  have {i. i < length ?xs ∧ ?xs ! i < ps} =
        {i. i < length ?ys ∧ prefix (?ys ! i) (length ps) < ps}
    by force
  moreover
  from patsearch-idx-start-card[OF sa-suffix-sorted, of s ps]
  have patsearch-idx-start ?ys ps = card {i. i < length ?ys ∧ prefix (?ys ! i) (length ps) < ps}
    by blast
  ultimately show ?thesis
    by simp
qed

function binary-search-sa-l
  where
l < r ⟹ binary-search-sa-l s ps l r =
  (let m = (l + r) div 2
     in
      (if prefix (suffix s (sa s ! m)) (length ps) < ps then
        binary-search-sa-l s ps (Suc m) r
       else binary-search-sa-l s ps l m)) |
¬ l < r ⟹ binary-search-sa-l s ps l r = l
  by fastforce+

termination
  by (relation measure (λ(s, ps, l, r). r − l);
      clarsimp simp: binary-search-term1 binary-search-term2)

lemma binary-search-sa-l-eq-abs-gen:
  ⟦l ≤ r; r ≤ length s⟧ ⟹
   binary-search-l (map (λi. prefix (suffix s i) (length ps)) (sa s)) ps l r =
   binary-search-sa-l s ps l r
proof (induct rule: binary-search-sa-l.induct[of - s ps l r])
  case (1 l r s ps)
  note IH = this
```

let *?m* = (*l* + *r*) *div 2*
have *?m* = *?m*
by *simp*

have *?m* < *r*
by (*simp add*: *IH*(*1*) *less-mult-imp-div-less*)
hence *Suc ?m* ≤ *r*
by *simp*

have *?m* ≤ *length s*
using *IH*(*5*) ‹*?m* < *r*› **by** *linarith*

have *l* ≤ *?m*
by (*simp add*: *IH*(*4*) *less-eq-div-iff-mult-less-eq*)

note *IH1* = *IH*(*2*)[*OF* ‹*?m* = *?m*› - ‹*Suc ?m* ≤ *r*› *IH*(*5*)]
note *IH2* = *IH*(*3*)[*OF* ‹*?m* = *?m*›- ‹*l* ≤ *?m*› ‹*?m* ≤ *length s*›]

let *?xs* = *map* (λ*i*. *prefix* (*suffix s i*) (*length ps*)) (*sa s*)

have *length ?xs* = *length s*
by (*simp add*: *sa-length*)
hence *P*: *?xs* ! *?m* = *prefix* (*suffix s* (*sa s* ! *?m*)) (*length ps*)
using *IH*(*5*) ‹*?m* < *r*› **by** *auto*

show *?case*
proof (*cases prefix* (*suffix s* (*sa s* ! *?m*)) (*length ps*) < *ps*)
assume *A*: *prefix* (*suffix s* (*sa s* ! *?m*)) (*length ps*) < *ps*
with *IH1*
have *binary-search-l ?xs ps* (*Suc ?m*) *r* = *binary-search-sa-l s ps* (*Suc ?m*) *r*
by *blast*
moreover
have *binary-search-l ?xs ps l r* = *binary-search-l ?xs ps* (*Suc ?m*) *r*
using *A IH*(*1*) *P* **by** *simp*
moreover
have *binary-search-sa-l s ps l r* = *binary-search-sa-l s ps* (*Suc ?m*) *r*
using *A IH*(*1*) **by** *simp*
ultimately show *?thesis*
by *simp*
next
assume *A*: ¬ *prefix* (*suffix s* (*sa s* ! *?m*)) (*length ps*) < *ps*
with *IH2*
have *binary-search-l ?xs ps l ?m* = *binary-search-sa-l s ps l ?m*
by *blast*

```
    moreover
    have binary-search-l ?xs ps l r = binary-search-l ?xs ps l ?m
      using A IH(1) P by simp
    moreover
    have binary-search-sa-l s ps l r = binary-search-sa-l s ps l ?m
      using A IH(1) by simp
    ultimately show ?thesis
      by simp
  qed
next
  case (2 l r s ps)
  then show ?case
    by simp
qed

corollary binary-search-sa-l-eq-abs:
  binary-search-l (map (λi. prefix (suffix s i) (length ps)) (sa s)) ps 0 (length s) =
   binary-search-sa-l s ps 0 (length s)
  using binary-search-sa-l-eq-abs-gen by blast

corollary binary-search-sa-l-correct:
  binary-search-sa-l s ps 0 (length s) = patsearch-idx-start (map (suffix s) (sa s)) ps
  using binary-search-sa-l-eq-abs binary-search-sa-leftmost by presburger
```

A.6.2 Rightmost

```
lemma binary-search-sa-rightmost:
  binary-search-r (map (λi. prefix (suffix s i) (length ps)) (sa s)) ps 0 (length s) =
   patsearch-idx-end (map (suffix s) (sa s)) ps
proof −
  let ?xs = map (λi. prefix (suffix s i) (length ps)) (sa s)
  let ?ys = map (suffix s) (sa s)

  have sorted ?xs
  proof (intro sorted-wrt-mapI)
    fix i j
    assume i < j j < length (sa s)
    with sorted-wrt-mapD[OF sa-suffix-sorted]
    have suffix s (sa s ! i) ≤ suffix s (sa s ! j)
      by blast
    with le-take-n
    show prefix (suffix s (sa s ! i)) (length ps) ≤ prefix (suffix s (sa s ! j)) (length ps)
      by blast
  qed
```

```
  have length ?xs = length ?ys
    by simp

  from binary-search-r-card[OF ‹sorted ?xs›]
  have binary-search-r ?xs ps 0 (length s) = card {i. i < length ?xs ∧ ?xs ! i ≤ ps}
    by (simp add: sa-length)
  moreover
  have {i. i < length ?xs ∧ ?xs ! i ≤ ps} =
        {i. i < length ?ys ∧ prefix (?ys ! i) (length ps) ≤ ps}
    by force
  moreover
  from patsearch-idx-end-card[OF sa-suffix-sorted, of s ps]
  have patsearch-idx-end ?ys ps = card {i. i < length ?ys ∧ prefix (?ys ! i) (length ps) ≤ ps}
    by blast
  ultimately show ?thesis
    by simp
qed

function binary-search-sa-r
  where
l < r ⟹ binary-search-sa-r s ps l r =
  (let m = (l + r) div 2
   in
    (if prefix (suffix s (sa s ! m)) (length ps) > ps then
      binary-search-sa-r s ps l m
     else binary-search-sa-r s ps (Suc m) r)) |
¬ l < r ⟹ binary-search-sa-r s ps l r = r
  by fastforce+
termination
  by (relation measure (λ(s, ps, l, r). r − l);
      clarsimp simp: binary-search-term1 binary-search-term2)

lemma binary-search-sa-r-eq-abs-gen:
  ⟦l ≤ r; r ≤ length s⟧ ⟹
   binary-search-r (map (λi. prefix (suffix s i) (length ps)) (sa s)) ps l r =
   binary-search-sa-r s ps l r
proof (induct rule: binary-search-sa-r.induct[of - s ps l r])
  case (1 l r s ps)
  note IH = this

  let ?m = (l + r) div 2
  have ?m = ?m
    by simp
```

```
have ?m < r
  by (simp add: IH(1) less-mult-imp-div-less)
hence Suc ?m ≤ r
  by simp

have ?m ≤ length s
  using IH(5) ‹?m < r› by linarith

have l ≤ ?m
  by (simp add: IH(4) less-eq-div-iff-mult-less-eq)

note IH1 = IH(2)[OF ‹?m = ?m› - ‹l ≤ ?m› ‹?m ≤ length s›]
note IH2 = IH(3)[OF ‹?m = ?m›- ‹Suc ?m ≤ r› IH(5)]

let ?xs = map (λi. prefix (suffix s i) (length ps)) (sa s)

have length ?xs = length s
  by (simp add: sa-length)
hence P: ?xs ! ?m = prefix (suffix s (sa s ! ?m)) (length ps)
  using IH(5) ‹?m < r› by auto

show ?case
proof (cases ps < prefix (suffix s (sa s ! ?m)) (length ps))
  assume A: ps < prefix (suffix s (sa s ! ?m)) (length ps)
  with IH1
  have binary-search-r ?xs ps l ?m = binary-search-sa-r s ps l ?m
    by blast
  moreover
 have binary-search-r ?xs ps l r = binary-search-r ?xs ps l ?m
    using A IH(1) P by simp
  moreover
  have binary-search-sa-r s ps l r = binary-search-sa-r s ps l ?m
    using A IH(1) by simp
  ultimately show ?thesis
    by simp
next
  assume A: ¬ ps < prefix (suffix s (sa s ! ?m)) (length ps)
  with IH2
  have binary-search-r ?xs ps (Suc ?m) r = binary-search-sa-r s ps (Suc ?m) r
    by blast
  moreover
  have binary-search-r ?xs ps l r = binary-search-r ?xs ps (Suc ?m) r
    using A IH(1) P by simp
```

```
    moreover
    have binary-search-sa-r s ps l r = binary-search-sa-r s ps (Suc ?m) r
      using A IH(1) by simp
    ultimately show ?thesis
      by simp
  qed
next
  case (2 l r s ps)
  then show ?case
    by simp
qed

corollary binary-search-sa-r-eq-abs:
  binary-search-r (map (λi. prefix (suffix s i) (length ps)) (sa s)) ps 0 (length s) =
   binary-search-sa-r s ps 0 (length s)
  using binary-search-sa-r-eq-abs-gen by blast

corollary binary-search-sa-r-correct:
  binary-search-sa-r s ps 0 (length s) = patsearch-idx-end (map (suffix s) (sa s)) ps
  using binary-search-sa-r-eq-abs binary-search-sa-rightmost by presburger
```

A.6.3 Combined

```
theorem binary-search-sa-correct:
  patsearch-set s ps =
    nth (sa s) ‘ {(binary-search-sa-l s ps 0 (length s))..<(binary-search-sa-r s ps 0 (length s))}
  using patsearch-set-eq-sa-range binary-search-sa-l-correct binary-search-sa-r-correct
  by presburger

end

end
theory Backward-Search
  imports BurrowsWheeler.Rank-Util
          BurrowsWheeler.Select-Util
          SuffixArray.Suffix-Array
begin
```

A.7 Backward Search Algorithm

Backward search algorithm obtained from [54].

A.7.1 Definitions

context *Suffix-Array-General* **begin**

A single backward look-up.

fun *backward-lookup* :: *'a list* ⇒ *'a* ⇒ *nat* ⇒ *nat*
  where
backward-lookup bs c i = (*select* (*sort bs*) *c 0*) + (*rank bs c i*)

Chain the backward look-ups for the whole reversed pattern.

fun *backward-lookups* :: *'a list* ⇒ *'a list* ⇒ *nat* ⇒ *nat*
  where
backward-lookups - [] *i* = *i* |
backward-lookups bs (*c*#*cs*) *i* = *backward-lookups bs cs* (*backward-lookup bs c i*)

Find the range of the pattern in the suffix array.

fun *search-backward* :: *'a list* ⇒ *'a list* ⇒ *nat* × *nat* ⇒ *nat* × *nat*
  where
search-backward bs [] (*i*, *j*) = (*i*, *j*) |
search-backward bs (*c*#*cs*) (*i*, *j*) =
  search-backward bs cs (*backward-lookup bs c i*, *backward-lookup bs c j*)

primrec *upt-tuple-set*
  where
upt-tuple-set (*x*, *y*) = {*x*..<*y*}

primrec *upt-tuple-list*
  where
upt-tuple-list (*x*, *y*) = [*x*..<*y*]

Find the range of the pattern in a suffix array but with an early exit.

fun *search-backward-early-exit* :: *'a list* ⇒ *'a list* ⇒ *nat* × *nat* ⇒ *nat* × *nat*
  where
search-backward-early-exit bs [] (*i*, *j*) = (*i*, *j*) |
search-backward-early-exit bs (*c*#*cs*) (*i*, *j*) =
  (*if i* < *j then*
    search-backward-early-exit bs cs (*backward-lookup bs c i*, *backward-lookup bs c j*)
   else (*i*, *j*))

The backward search algorithm.

fun *backward-search* :: *'a list* ⇒ *'a list* ⇒ *nat* × *nat*
  where

backward-search bs xs = search-backward-early-exit bs (rev xs) (0, length bs)

end

end
theory *Backward-Search-Verification*
imports *BurrowsWheeler.BWT-SA-Corres*
../sa-search/SA-Search
Backward-Search
begin

A.8 Verification of the Backward Search Algorithm

context *Suffix-Array-General* **begin**

A.8.1 Single Backward Look-up

lemma *backward-lookup-idx-mono*:
j ≤ i ⟹ backward-lookup bs c j ≤ backward-lookup bs c i
by (*simp add*: *rank-idx-mono*)

lemma *backward-lookup-upper*:
assumes *bs = bwt-sa s*
shows *backward-lookup bs c i ≤ length s*
proof (*cases c ∈ set s*)
assume *c ∈ set s*
hence *rank bs c i ≤ count-list bs c*
by (*simp add*: *rank-upper-bound-gen*)
moreover
have *count-list bs c = count-list s c*
by (*metis assms bwt-sa-perm count-list-perm*)
hence *count-list bs c = card {i. i < length s ∧ s ! i = c}*
by (*simp add*: *count-list-card*)
ultimately have *rank bs c i ≤ card {i. i < length s ∧ s ! i = c}*
by *argo*
moreover
have *0 < count-list (sort bs) c*
by (*metis ‹c ∈ set s› ‹count-list bs c = count-list s c› bot-nat-0.not-eq-extremum*
count-list-0-iff set-sort)
with *select-sorted-0[of sort bs c] bij-betw-sort-idx-ex[of sort bs s c]*
have *select (sort bs) c 0 = card {i. i < length s ∧ s ! i < c}*
by (*metis* (*mono-tags*, *lifting*) *bwt-sa-perm assms bij-betw-same-card*
sorted-list-of-multiset-mset sorted-sort)
moreover

```
  have card {i. i < length s ∧ s ! i < c} + card {i. i < length s ∧ s ! i = c} ≤ length s
    by (metis (full-types) ‹count-list bs c = card {i. i < length s ∧ s ! i = c}›
                           ‹count-list bs c = count-list s c› card-pl-count-list-strict-upper)
  ultimately show ?thesis
    by auto
next
  assume c ∉ set s
  hence count-list s c = 0
    by simp
  hence count-list bs c = 0
    by (metis assms bwt-sa-perm count-list-perm)
  hence rank bs c i = 0
    by (metis notin-count rank-not-in)
  moreover
  have select (sort bs) c 0 = length s
   by (metis (no-types, lifting) ‹count-list bs c = 0› assms bwt-sa-length length-sort notin-count
                                 notin-imp-select-length set-sort)
  ultimately show ?thesis
    by simp
qed

lemma backward-lookup-0:
  assumes a ∈ set s
  and     bs = bwt-sa s
shows backward-lookup bs a 0 = patsearch-idx-start (map (suffix s) (sa s)) [a]
proof −
  have 0 < count-list (sort bs) a
    by (metis bwt-sa-perm assms(1) assms(2) bot-nat-0.not-eq-extremum count-list-0-iff
          count-list-perm mset-sort)
  with select-sorted-0[of sort bs a] bij-betw-sort-idx-ex[of sort bs s a]
  have select (sort bs) a 0 = card {i. i < length s ∧ s ! i < a}
    by (metis (mono-tags, lifting) bwt-sa-perm assms(2) bij-betw-same-card
                                   sorted-list-of-multiset-mset sorted-sort)
  moreover
  have rank bs a 0 = 0
    by (simp add: rank-0)
  ultimately have backward-lookup bs a 0 = card {i. i < length s ∧ s ! i < a}
    by auto
  then show ?thesis
    using patsearch-idx-start-singleton assms by fastforce
qed

lemma backward-lookup-length:
```

assumes *bs* = *bwt-sa s*
and *a* ∈ *set s*
shows *backward-lookup bs a* (*length s*) = *patsearch-idx-end* (*map* (*suffix s*) (*sa s*)) [*a*]
proof −
have *0* < *count-list* (*sort bs*) *a*
by (*metis bwt-sa-perm assms*(*1*) *assms*(*2*) *bot-nat-0.not-eq-extremum count-list-0-iff count-list-perm mset-sort*)
with *select-sorted-0*[*of sort bs a*] *bij-betw-sort-idx-ex*[*of sort bs s a*]
have *select* (*sort bs*) *a 0* = *card* {*i*. *i* < *length s* ∧ *s* ! *i* < *a*}
by (*metis* (*mono-tags*, *lifting*) *bwt-sa-perm assms*(*1*) *bij-betw-same-card sorted-list-of-multiset-mset sorted-sort*)
moreover
have *rank bs a* (*length s*) = *count-list s a*
by (*metis assms*(*1*) *bwt-sa-length bwt-sa-perm count-list-perm rank-length*)
ultimately have
backward-lookup bs a (*length s*) = *card* {*i*. *i* < *length s* ∧ *s* ! *i* < *a*} + *count-list s a*
by *auto*
then show *?thesis*
using *assms*(*2*) *patsearch-idx-end-singleton* **by** *presburger*
qed

theorem *backward-lookup-cons-fail*:
assumes *valid-list s*
and *bs* = *bwt-sa s*
and *patsearch-set s ys* ≠ {}
and *patsearch-set s* (*a* # *ys*) = {}
and *valid-list* (*a* # *ys*) ∨ *bot* ∉ *set* (*a* # *ys*)
shows *backward-lookup bs a* (*patsearch-idx-end* (*map* (*suffix s*) (*sa s*)) *ys*) ≤
backward-lookup bs a (*patsearch-idx-start* (*map* (*suffix s*) (*sa s*)) *ys*)
proof (*cases ys*)
case *Nil*

from *patsearch-idx-start-nil*[*of map* (*suffix s*) (*sa s*)]
have *patsearch-idx-start* (*map* (*suffix s*) (*sa s*)) [] = *0*
by *blast*
moreover
from *patsearch-idx-end-nil*[*of map* (*suffix s*) (*sa s*)]
have *patsearch-idx-end* (*map* (*suffix s*) (*sa s*)) [] = *length s*
by (*simp add*: *sa-length*)
moreover
have *patsearch-set s* [*a*] = {}
using *assms*(*4*) *local.Nil* **by** *auto*

```
  hence ∀ x. rank bs a x = 0
   by (metis assms(2) bwt-sa-perm count-list-perm notin-count rank-not-in patsearch-set-count-list)
  ultimately show ?thesis
    by simp
next
  case (Cons b zs)
  note C0 = this



  let ?xs = map (suffix s) (sa s)
  let ?i = patsearch-idx-start ?xs ys
  let ?j = patsearch-idx-end ?xs ys

  have a ≠ bot
  proof (rule ccontr)
    assume ¬a ≠ bot
    hence a = bot
      by blast
    with assms(5)
    show False
      by (simp add: local.Cons valid-list-iff-butlast-app-last)
  qed


  from patsearch-set-eq-sa-range[of s ys]
  have ?i < ?j
    using assms(3) atLeastLessThan-empty-iff2 by blast
  hence rank bs a ?i ≤ rank bs a ?j
    by (meson le-eq-less-or-eq rank-idx-mono)


  have rank bs a ?i = rank bs a ?j
  proof (rule ccontr)
    assume rank bs a ?i ≠ rank bs a ?j
    with ‹rank bs a ?i ≤ rank bs a ?j›
    have rank bs a ?i < rank bs a ?j
      using le-eq-less-or-eq by blast
    hence count-list (take ?i bs) a < count-list (take ?j bs) a
      by (metis rank-def)
    moreover
    have take ?j bs = take ?i bs @ list-slice bs ?i ?j
      by (metis (no-types, lifting) ‹?i < ?j› append-take-drop-id less-imp-le-nat list-slice.simps
                                    min.absorb1 take-take)
```

ultimately have *count-list* (*list-slice bs ?i ?j*) *a* > *0*
by (*metis* (*no-types*, *lifting*) ‹*rank bs a ?i* ≠ *rank bs a ?j*› *add.right-neutral*
count-list-append gr0I rank-def)
hence ∃ *k* < *length bs* . *?i* ≤ *k* ∧ *k* < *?j* ∧ *bs* ! *k* = *a*
by (*meson in-count-list-slice-nth-ex*)
then obtain *k* **where**
k < *length bs ?i* ≤ *k k* < *?j bs* ! *k* = *a*
by *blast*
hence *prefix* (*suffix s* (*sa s* ! *k*)) (*length ys*) = *ys*
using *in-sa-range* **by** *blast*

have ∃ *n*. *sa s* ! *k* = *Suc n*
proof (*rule ccontr*)
assume ¬(∃ *n*. *sa s* ! *k* = *Suc n*)
hence *sa s* ! *k* = *0*
using *gr0-conv-Suc* **by** *blast*
hence *bs* ! *k* = *s* ! ((*length s* − *Suc 0*) *mod* (*length s*))
using ‹*k* < *length bs*› *assms*(*2*) *bwt-sa-length bwt-sa-nth* **by** *force*
hence *bs* ! *k* = *s* ! (*length s* − *Suc 0*)
by (*metis One-nat-def assms*(*1*) *diff-less less-numeral-extra*(*1*) *mod-less valid-list-length*)
hence *bs* ! *k* = *bot*
by (*metis One-nat-def assms*(*1*) *last-conv-nth length-greater-0-conv valid-list-def*)
with ‹*bs* ! *k* = *a*› ‹*a* ≠ *bot*›
show *False*
by *blast*
qed
then obtain *n* **where**
sa s ! *k* = *Suc n*
by *blast*
hence *bs* ! *k* = *s* ! ((*Suc n* + *length s* − *Suc 0*) *mod* (*length s*))
using ‹*k* < *length bs*› *assms*(*2*) *bwt-sa-length bwt-sa-nth* **by** *auto*
hence *bs* ! *k* = *s* ! *n*
by (*metis One-nat-def Suc-lessD* ‹*k* < *length bs*› ‹*sa s* ! *k* = *Suc n*› *assms*(*2*) *bwt-sa-length*

diff-Suc-1 mod-add-self2 mod-less plus-nat.simps(*2*) *sa-nth-ex*)
hence *s* ! *n* = *a*
by (*simp add*: ‹*bs* ! *k* = *a*›)

have *suffix s n* = *s* ! *n* # *suffix s* (*sa s* ! *k*)
by (*metis Suc-lessD* ‹*k* < *length bs*› ‹*sa s* ! *k* = *Suc n*› *assms*(*2*) *bwt-sa-length sa-nth-ex*
suffix-cons-Suc)
hence *prefix* (*suffix s n*) (*length* (*a* # *ys*)) = *a* # *ys*
by (*simp add*: ‹*prefix* (*suffix s* (*sa s* ! *k*)) (*length ys*) = *ys*› ‹*s* ! *n* = *a*›)
hence *list-slice s n* (*n* + *length* (*a* # *ys*)) = *a* # *ys*

```
    by (simp add: add.commute take-drop)
  moreover
  have n < length s
    by (metis ‹k < length bs› ‹sa s ! k = Suc n› assms(2) bwt-sa-length not-less-eq
        order-less-imp-not-less sa-nth-ex)
  ultimately have n ∈ patsearch-set s (a # ys)
    by (simp add: patsearch-set-def)
  then show False
    by (simp add: assms(4))
 qed
 then show ?thesis
  by simp
qed

lemma sort-nth-a:
 assumes bs = bwt-sa s 0 < count-list s a k < length s select (sort bs) a 0 ≤ k
      k < select (sort bs) a 0 + count-list s a
 shows sort s ! k = a
proof −

 have count-list s a = count-list (sort s) a
  by (metis count-list-perm perm-map-nths-sa sort-eq-map-nths-sa)

 have 0 < count-list (sort bs) a
  by (metis ‹count-list s a = count-list (sort s) a› assms(1,2) bwt-sa-perm
      sorted-list-of-multiset-mset)
 with select-sorted-0[of sort bs a] bij-betw-sort-idx-ex[of sort bs s a]
 have select (sort bs) a 0 = card {i. i < length s ∧ s ! i < a}
  by (metis (mono-tags, lifting) assms(1) bij-betw-same-card bwt-sa-perm
                    sorted-list-of-multiset-mset sorted-sort)

 have ∃ k'. k = select (sort bs) a 0 + k' ∧ k' < count-list (sort s) a
  by (metis ‹count-list s a = count-list (sort s) a› add-less-imp-less-left assms(4,5) less-eqE)
 with sorted-nth-gen[of sort bs a] ‹select (sort bs) a 0 = card -› ‹k < length s›
 show sort s ! k = a
  by (metis (mono-tags, lifting) ‹0 < count-list (sort bs) a› assms(1,4) bwt-sa-perm
                    dual-order.strict-trans2 length-sort select-sorted-0
                    sorted-list-of-multiset-mset sorted-nth-gen sorted-sort)
qed

theorem backward-lookup-cons-success-start:
 assumes valid-list s
 and     bs = bwt-sa s
 and     patsearch-set s ys ≠ {}
```

```
  and     patsearch-set s (a # ys) ≠ {}
  and     valid-list (a # ys) ∨ bot ∉ set (a # ys)
shows patsearch-idx-start (map (suffix s) (sa s)) (a # ys) =
      backward-lookup bs a (patsearch-idx-start (map (suffix s) (sa s)) ys)
proof (cases ys)
  case Nil
  from patsearch-idx-start-nil[of map (suffix s) (sa s)]
  have patsearch-idx-start (map (suffix s) (sa s)) [] = 0
    by blast
  hence backward-lookup bs a (patsearch-idx-start (map (suffix s) (sa s)) ys) =
        backward-lookup bs a 0
    using local.Nil by presburger
  moreover
  have a ∈ set s
    by (meson assms(4) list.set-intros(1) patsearch-set-not-empty-subset subsetD)
  ultimately show ?thesis
    using backward-lookup-0[OF - assms(2), of a] local.Nil by fastforce
next
  case (Cons b zs)

  let ?xs = map (suffix s) (sa s)
  let ?i = patsearch-idx-start ?xs ys
  let ?j = patsearch-idx-end ?xs ys

  let ?i' = patsearch-idx-start ?xs (a # ys)
  let ?j' = patsearch-idx-end ?xs (a # ys)

  have a ≠ bot
    by (metis assms(5) butlast.simps(2) list.discI list.set-intros(1) local.Cons
          valid-list-iff-butlast-app-last)

  from backward-lookup-upper[OF assms(2)]
  have select (sort bs) a 0 + rank bs a ?i ≤ length s
    by simp

  from assms(3) patsearch-set-eq-sa-range[of s ys]
  have ?i < ?j
    using atLeastLessThan-empty-iff2 by blast
  moreover
  have ?j ≤ length s
    using sa-length by fastforce
  ultimately have ?i < length s
    using dual-order.strict-trans1 by blast
```

have *rank bs a ?i* ≤ *count-list s a*
by (*metis* (*mono-tags*, *lifting*) *assms*(*2*) *bwt-sa-perm count-list-perm rank-upper-bound-gen*)

have *P1*: ∀ *k* < *length s*. *k* < *?i* ⟶ *prefix* (*suffix s* (*sa s* ! *k*)) (*length ys*) < *ys*
using *patsearch-idx-start-less*[*of* - *?xs ys*] **by** (*simp add*: *sa-length*)

have *P2*: ∀ *k* < *length s*. *?i* ≤ *k* ⟶ *ys* ≤ *prefix* (*suffix s* (*sa s* ! *k*)) (*length ys*)
using *patsearch-idx-start-success*[*OF sa-suffix-sorted*[*of s*], *of ys*]
by (*metis* (*no-types*, *lifting*) *length-map nth-map sa-length*)

have *rank bs a ?i* < *count-list s a*
proof (*rule ccontr*)
assume ¬ *rank bs a ?i* < *count-list s a*

have *A*: ∀ *k* < *length bs*. *?i* ≤ *k* ⟶ *bs* ! *k* ≠ *a*
by (*metis* (*no-types*, *lifting*) ‹¬ *rank bs a ?i* < *count-list s a*› *rank-upper-bound*
assms(*2*) *bwt-sa-perm count-list-perm le-neq-implies-less*
linorder-not-le rank-idx-mono ‹*rank bs a ?i* ≤ *count-list s a*›)

have ∀ *k* < *length s*.
list-slice s (*bwt-perm s* ! *k*) ((*bwt-perm s* ! *k*) + *length* (*a* # *ys*)) ≠ *a* # *ys*
proof (*intro allI impI*)
fix *k*
assume *k* < *length s*

have *P*: *bs* ! *k* ≠ *bot* ⟹
list-slice s (*bwt-perm s* ! *k*) ((*bwt-perm s* ! *k*) + *length* (*a* # *ys*)) =
bs ! *k* # *prefix* (*suffix s* (*sa s* ! *k*)) (*length ys*)
by (*metis* ‹*k* < *length s*› *assms*(*1*) *assms*(*2*) *diff-add-inverse length-Cons*
list-slice-to-suffix suffix-bwt-perm-sa take-Suc-Cons)

have *k* < *?i* ∨ *?i* ≤ *k*
using *leI* **by** *blast*
moreover
have *k* < *?i* ⟹ *list-slice s* (*bwt-perm s* ! *k*) ((*bwt-perm s* ! *k*) + *length* (*a* # *ys*)) ≠ *a* # *ys*
by (*metis* (*no-types*, *lifting*) *P1 P Nat.add-0-right nth-list-slice list.inject nless-le*
‹*a* ≠ *bot*› ‹*k* < *length s*› *assms*(*2*) *bwt-perm-s-nth nth-Cons-0*
length-greater-0-conv list.discI)
moreover

have *?i* ≤ *k* ⟹ *list-slice s* (*bwt-perm s* ! *k*) ((*bwt-perm s* ! *k*) + *length* (*a* # *ys*)) ≠ *a* # *ys*
by (*metis* (*no-types*, *lifting*) *A* ‹*k* < *length s*› *add.right-neutral assms*(*2*) *bwt-perm-s-nth* *bwt-sa-length length-greater-0-conv list.discI nth-Cons-0* *nth-list-slice*)
ultimately show *list-slice s* (*bwt-perm s* ! *k*) ((*bwt-perm s* ! *k*) + *length* (*a* # *ys*)) ≠ *a* # *ys*
by *blast*
qed
hence *patsearch-set s* (*a* # *ys*) = {}
by (*metis* (*mono-tags*, *lifting*) *empty-Collect-eq ex-bwt-perm-nth patsearch-set-def*)
then show *False*
by (*simp add*: *assms*(*4*))
qed

have *0* < *count-list* (*sort bs*) *a*
by (*metis* (*mono-tags*, *lifting*) ‹*rank bs a ?i* < *count-list s a*› *assms*(*2*) *bwt-sa-perm* *count-list-perm gr0I less-nat-zero-code sort-perm*)
with *select-sorted-0*[*of sort bs a*] *bij-betw-sort-idx-ex*[*of sort bs s a*]
have *select* (*sort bs*) *a 0* = *card* {*i*. *i* < *length s* ∧ *s* ! *i* < *a*}
by (*metis* (*mono-tags*, *lifting*) *bwt-sa-perm assms*(*2*) *bij-betw-same-card* *sorted-list-of-multiset-mset sorted-sort*)

have *select* (*sort bs*) *a 0* + *count-list s a* ≤ *length s*
by (*metis backward-lookup-upper*[*OF assms*(*2*)] *assms*(*2*) *backward-lookup.simps bwt-sa-perm* *count-list-perm rank-length*)
hence *select* (*sort bs*) *a 0* + *rank bs a ?i* < *length s*
using ‹*rank bs a ?i* < *count-list s a*› **by** *linarith*
moreover
have *rank bs a ?i* < *count-list* (*sort bs*) *a*
by (*metis* (*no-types*, *lifting*) ‹*rank bs a ?i* < *count-list s a*› *assms*(*2*) *bwt-sa-perm* *count-list-perm perm-map-nths-sa sort-eq-map-nths-sa*)

ultimately have *rank* (*sort s*) *a* (*select* (*sort bs*) *a 0* + *rank bs a ?i*) = *rank bs a ?i*
using *rank-select sorted-select-0-plus*[*of sort bs*]
by (*metis* (*mono-tags*, *lifting*) *assms*(*2*) *bwt-sa-perm length-sort sorted-list-of-multiset-mset* *sorted-sort*)

have *P1'*: ⋀*k*. ⟦*k* < *length s*; *k* < *?i*; *bs* ! *k* = *a*⟧ ⟹ *prefix* (*suffix s* (*bwt-perm s* ! *k*)) (*length* (*a* # *ys*)) < *a* # *ys*
proof −
fix *k*

```
  assume k < length s k < ?i bs ! k = a
  with P1
  have prefix (suffix s (sa s ! k)) (length ys) < ys
    by blast
  moreover
  {
    have s ! (bwt-perm s ! k) = a
      using ‹bs ! k = a› ‹k < length s› assms(2) bwt-perm-s-nth by auto
    moreover
    have bwt-perm s ! k = (sa s ! k + length s − Suc 0) mod (length s)
      using ‹k < length s› bwt-perm-nth by presburger
    moreover
    have sa s ! k > 0
      by (metis One-nat-def ‹a ≠ bot› add-diff-cancel-left' assms(1) calculation diff-less
            diff-zero gr0I last-conv-nth length-greater-0-conv less-Suc-eq-le
            less-or-eq-imp-le mod-less valid-list-def)
    ultimately have suffix s (bwt-perm s ! k) = a # suffix s (sa s ! k)
      by (metis Nat.add-diff-assoc2 Suc-leI Suc-lessD Suc-pred ‹k < length s› mod-add-self2
            mod-less sa-nth-ex suffix-cons-Suc)
  }
  ultimately show prefix (suffix s (bwt-perm s ! k)) (length (a # ys)) < a # ys
    by simp
qed


have P2': ⋀k. ⟦k < length s; ?i ≤ k; bs ! k = a⟧ ⟹
        a # ys ≤ prefix (suffix s (bwt-perm s ! k)) (length (a # ys))
proof −
  fix k
  assume k < length s ?i ≤ k bs ! k = a
  with P2
  have ys ≤ prefix (suffix s (sa s ! k)) (length ys)
    by blast
  moreover
  {
    have s ! (bwt-perm s ! k) = a
      using ‹bs ! k = a› ‹k < length s› assms(2) bwt-perm-s-nth by auto
    moreover
    have bwt-perm s ! k = (sa s ! k + length s − Suc 0) mod (length s)
      using ‹k < length s› bwt-perm-nth by presburger
    moreover
    have sa s ! k > 0
      by (metis One-nat-def ‹a ≠ bot› add-diff-cancel-left' assms(1) calculation diff-less
            diff-zero gr0I last-conv-nth length-greater-0-conv less-Suc-eq-le
```

```
        less-or-eq-imp-le mod-less valid-list-def)
    ultimately have suffix s (bwt-perm s ! k) = a # suffix s (sa s ! k)
     by (metis Nat.add-diff-assoc2 Suc-leI Suc-lessD Suc-pred ‹k < length s› mod-add-self2
        mod-less sa-nth-ex suffix-cons-Suc)
  }
  ultimately show a # ys ≤ prefix (suffix s (bwt-perm s ! k)) (length (a # ys))
   by simp
qed

have count-list s a = count-list (sort s) a
  by (metis count-list-perm perm-map-nths-sa sort-eq-map-nths-sa)

have 0 < count-list s a
  using ‹rank bs a ?i < count-list s a› by linarith

note sort-nth-a = sort-nth-a[OF assms(2) ‹0 < count-list s a›]

have sort-bwt-perm-nth:
  ⋀k. ⟦k < length s; sort s ! k = a⟧ ⟹
      ∃k' < length s. sa s ! k = bwt-perm s ! k' ∧ rank (sort s) a k = rank (bwt-sa s) a k'
proof −
  fix k
  assume k < length s sort s ! k = a
  moreover
  have ∃k' < length s. sa s ! k = bwt-perm s ! k'
   by (metis ‹k < length s› ex-bwt-perm-nth sa-nth-ex)
  then obtain k' where
    k' < length s sa s ! k = bwt-perm s ! k'
    by blast
  moreover
  from sa-bwt-perm-same-rank[OF assms(1) ‹k < length s› ‹k' < length s›]
  have rank (sort s) a k = rank (bwt-sa s) a k'
    using ‹k < length s› ‹sort s ! k = a› bwt-perm-s-nth sort-sa-nth calculation by force
  ultimately show
    ∃k' < length s. sa s ! k = bwt-perm s ! k' ∧ rank (sort s) a k = rank (bwt-sa s) a k'
    by blast
qed

let ?i' = select (sort bs) a 0 + rank bs a ?i

have ?i' ≤ length ?xs
  using ‹?i' ≤ length s› sa-length by auto
moreover
```

```
have ⋀k. ⟦k < length s; k < ?i'⟧ ⟹
        prefix (suffix s (sa s ! k)) (length (a # ys)) < a # ys
proof −
 fix k
 assume k < length s k < select (sort bs) a 0 + rank bs a ?i


 have k < select (sort bs) a 0 ∨ select (sort bs) a 0 ≤ k
  using linorder-le-less-linear by blast
 moreover

 have k < select (sort bs) a 0 ⟹ prefix (suffix s (sa s ! k)) (length (a # ys)) < a # ys
 proof −
  assume k < select (sort bs) a 0
  with sa-suffix-nth-less[of k s a]
  have suffix s (sa s ! k) < [a]
   by (simp add: ‹select (sort bs) a 0 = card {i. i < length s ∧ s ! i < a}›)
  hence s ! (sa s ! k) < a
   by (metis Cons-less-Cons ‹k < length s› not-less-Nil sa-nth-ex suffix-cons-Suc)
  then show ?thesis
   using ‹k < length s› sa-nth-ex suffix-cons-ex by fastforce
 qed
 moreover

 have select (sort bs) a 0 ≤ k ⟹ prefix (suffix s (sa s ! k)) (length (a # ys)) < a # ys
 proof −
  assume select (sort bs) a 0 ≤ k

  from sorted-card-rank-idx[of sort s k]
  have k = card {j. j < length (sort s) ∧ sort s ! j < sort s ! k} +
           rank (sort s) (sort s ! k) k
   using ‹k < length s› by fastforce
  moreover
  from sort-nth-a[OF ‹k < length s› ‹select (sort bs) a 0 ≤ k›]
  have sort s ! k = a
   using ‹k < select (sort bs) a 0 + rank bs a ?i› ‹rank bs a ?i ≤ count-list s a›
       add-left-mono order.strict-trans2 by blast
  ultimately have k = select (sort bs) a 0 + rank (sort s) a k
   using ‹select (sort bs) a 0 = -› ‹k < length s› sa-card-rank-s-idx sort-sa-nth by force
  hence rank (sort s) a k < rank bs a ?i
   using ‹k < ?i'› by linarith
  moreover
  from sort-bwt-perm-nth[OF ‹k < length s› ‹sort s ! k = a›]
```

```
    obtain k' where
      k' < length s sa s ! k = bwt-perm s ! k' rank (sort s) a k = rank (bwt-sa s) a k'
      by blast
    ultimately have rank (bwt-sa s) a k' < rank bs a ?i
      by simp
    hence k' < ?i
      by (metis (no-types, lifting) assms(2) linorder-not-le rank-idx-mono)
    with P1'[OF ‹k' < length s›]
    have prefix (suffix s (bwt-perm s ! k')) (length (a # ys)) < a # ys
      by (metis ‹k < length s› ‹k' < length s› ‹sa s ! k = bwt-perm s ! k'› ‹sort s ! k = a›
            assms(2) bwt-perm-s-nth sort-sa-nth)
    then show ?thesis
      using ‹sa s ! k = bwt-perm s ! k'› by argo
  qed
  ultimately show prefix (suffix s (sa s ! k)) (length (a # ys)) < a # ys
    by blast
qed
hence ∀k < ?i'. prefix (?xs ! k) (length (a # ys)) < a # ys
  using ‹?i' ≤ length s› sa-length by force
moreover


have ⋀k. ⟦k < length s; select (sort bs) a 0 + rank bs a ?i ≤ k⟧ ⟹
        a # ys ≤ prefix (suffix s (sa s ! k)) (length (a # ys))
proof −
  fix k
  assume k < length s select (sort bs) a 0 + rank bs a ?i ≤ k


  have k < select (sort bs) a 0 + count-list s a ∨ select (sort bs) a 0 + count-list s a ≤ k
    using linorder-le-less-linear by blast
  moreover

  have select (sort bs) a 0 + count-list s a ≤ k ⟹
      a # ys ≤ prefix (suffix s (sa s ! k)) (length (a # ys))
  proof −
    assume select (sort bs) a 0 + count-list s a ≤ k
    hence ∃k'. k = select (sort bs) a 0 + k' ∧ count-list s a ≤ k'
      using nat-le-iff-add by auto
    with sa-suffix-nth-gr[of s a] assms(1)

    have a # ys < suffix s (sa s ! k)
      using ‹k < length s› ‹select (sort bs) a 0 = card -› by auto
    then show ?thesis
```

```
    by (metis le-take-n order-less-imp-le order-refl take-all-iff)
  qed
  moreover

  have k < select (sort bs) a 0 + count-list s a ⟹
      a # ys ≤ prefix (suffix s (sa s ! k)) (length (a # ys))
  proof −
    assume k < select (sort bs) a 0 + count-list s a

    from sorted-card-rank-idx[of sort s k]

    have k = card {j. j < length (sort s) ∧ sort s ! j < sort s ! k} +
             rank (sort s) (sort s ! k) k
      using ‹k < length s› by fastforce
    moreover

    from sort-nth-a[OF ‹k < length s› - ‹k < select (sort bs) a 0 + count-list s a›]
    have sort s ! k = a
      using ‹select (sort bs) a 0 + rank bs a ?i ≤ k› add-leD1 by blast

    ultimately have rank bs a ?i ≤ rank (sort s) a k
      by (metis (no-types, lifting) ‹select (sort bs) a 0 + rank bs a ?i ≤ k› rank-idx-mono
          ‹rank (sort s) a (select (sort bs) a 0 + rank bs a ?i) = rank bs a ?i›)
    moreover
    from sort-bwt-perm-nth[OF ‹k < length s› ‹sort s ! k = a›]
    obtain k′ where
      k′ < length s sa s ! k = bwt-perm s ! k′ rank (sort s) a k = rank (bwt-sa s) a k′
      by blast
    ultimately have rank bs a ?i ≤ rank bs a k′
      using assms(2) by argo

    have ?i ≤ k′
    proof (rule ccontr)
      assume ¬ ?i ≤ k′
      hence k′ < ?i
        using linorder-not-le by blast
      hence rank bs a k′ < rank bs a ?i
        by (metis (no-types, lifting) ‹k < length s› ‹k′ < length s› assms(2) rank-less
                                      ‹sa s ! k = bwt-perm s ! k′› ‹sort s ! k = a› bwt-perm-s-nth
                                      bwt-sa-length sort-sa-nth)
      with ‹rank bs a ?i ≤ rank bs a k′›
      show False
        by linarith
    qed
```

```
    then show ?thesis
      by (metis (no-types, lifting) P2' ‹k < length s› ‹k' < length s›  assms(2) bwt-perm-nth
                                    ‹sa s ! k = bwt-perm s ! k'› ‹sort s ! k = a› bwt-sa-nth
                                    sort-sa-nth)
   qed
   ultimately show a # ys ≤ prefix (suffix s (sa s ! k)) (length (a # ys))
    by blast
  qed
  hence ∀ k < length ?xs. ?i' ≤ k ⟶ a # ys ≤ prefix (?xs ! k) (length (a # ys))
   by (simp add: sa-length)
  ultimately show ?thesis
   using patsearch-idx-startpoint[OF sa-suffix-sorted[of s], of ?i' a # ys]
       backward-lookup.simps
   by presburger
qed

theorem backward-lookup-cons-success-end:
  assumes valid-list s
  and     bs = bwt-sa s
  and     patsearch-set s ys ≠ {}
  and     patsearch-set s (a # ys) ≠ {}
  and     valid-list (a # ys) ∨ bot ∉ set (a # ys)
shows patsearch-idx-end (map (suffix s) (sa s)) (a # ys) =
      backward-lookup bs a (patsearch-idx-end (map (suffix s) (sa s)) ys)
proof (cases ys)
  case Nil
  from patsearch-idx-end-nil[of map (suffix s) (sa s)]
  have patsearch-idx-end (map (suffix s) (sa s)) [] = length s
   by (simp add: sa-length)
  hence backward-lookup bs a (patsearch-idx-end (map (suffix s) (sa s)) ys) =
       backward-lookup bs a (length s)
   using local.Nil by presburger
  moreover
  have a ∈ set s
   by (meson assms(4) list.set-intros(1) patsearch-set-not-empty-subset subsetD)
  ultimately show ?thesis
   using backward-lookup-length[OF assms(2)] local.Nil by fastforce
next
  case (Cons b zs)

  let ?xs = map (suffix s) (sa s)
  let ?i = patsearch-idx-start ?xs ys
  let ?j = patsearch-idx-end ?xs ys
```

```
let ?i' = patsearch-idx-start ?xs (a # ys)
let ?j' = patsearch-idx-end ?xs (a # ys)

have a ≠ bot
  using local.Cons assms(5)
  by (metis butlast.simps(2) list.distinct(1) list.set-intros(1) valid-list-iff-butlast-app-last)

have a ∈ set s
  by (meson assms(4) list.set-intros(1) patsearch-set-not-empty-subset subsetD)

with backward-lookup-upper[OF assms(2)]
have select (sort bs) a 0 + rank bs a ?j ≤ length s
  by simp

from assms(3) patsearch-set-eq-sa-range[of s ys]
have ?i < ?j
  using atLeastLessThan-empty-iff2 by blast
moreover
have ?j ≤ length s
  using sa-length by fastforce
ultimately have ?i < length s
  using dual-order.strict-trans1 by blast

have rank bs a ?j ≤ count-list s a
  by (metis (mono-tags, lifting) assms(2) bwt-sa-perm count-list-perm rank-upper-bound-gen)

have P1: ∀ k < length s. k < ?j ⟶ prefix (suffix s (sa s ! k)) (length ys) ≤ ys
  using patsearch-idx-end-less[of - - ys] by auto

have P2: ∀ k < length s. ?j ≤ k ⟶ ys < prefix (suffix s (sa s ! k)) (length ys)
  using patsearch-idx-end-success[OF sa-suffix-sorted[of s], of ys]
  by (metis (no-types, lifting) length-map nth-map sa-length)

have P1': ⋀k. ⟦k < length s; k < ?j; bs ! k = a⟧ ⟹
        prefix (suffix s (bwt-perm s ! k)) (length (a # ys)) ≤ a # ys
proof −
  fix k
  assume k < length s k < ?j bs ! k = a
  with P1
  have prefix (suffix s (sa s ! k)) (length ys) ≤ ys
    by blast
  moreover
  {
    have s ! (bwt-perm s ! k) = a
```

```
    using ‹bs ! k = a› ‹k < length s› assms(2) bwt-perm-s-nth by auto
  moreover
  have bwt-perm s ! k = (sa s ! k + length s − Suc 0) mod (length s)
    using ‹k < length s› bwt-perm-nth by presburger
  moreover
  have sa s ! k > 0
    by (metis One-nat-def ‹a ≠ bot› add-diff-cancel-left' assms(1) calculation diff-less
          diff-zero gr0I last-conv-nth length-greater-0-conv less-Suc-eq-le
          less-or-eq-imp-le mod-less valid-list-def)
  ultimately have suffix s (bwt-perm s ! k) = a # suffix s (sa s ! k)
    by (metis Nat.add-diff-assoc2 Suc-leI Suc-lessD Suc-pred ‹k < length s› mod-add-self2
          mod-less sa-nth-ex suffix-cons-Suc)
 }
 ultimately show prefix (suffix s (bwt-perm s ! k)) (length (a # ys)) ≤ a # ys
  by simp
qed

have P2': ⋀k. ⟦k < length s; ?j ≤ k; bs ! k = a⟧ ⟹
        a # ys < prefix (suffix s (bwt-perm s ! k)) (length (a # ys))
proof −
 fix k
 assume k < length s ?j ≤ k bs ! k = a
 with P2
 have ys < prefix (suffix s (sa s ! k)) (length ys)
  by blast
 moreover
 {
  have s ! (bwt-perm s ! k) = a
    using ‹bs ! k = a› ‹k < length s› assms(2) bwt-perm-s-nth by auto
  moreover
  have bwt-perm s ! k = (sa s ! k + length s − Suc 0) mod (length s)
    using ‹k < length s› bwt-perm-nth by presburger
  moreover
  have sa s ! k > 0
    by (metis One-nat-def ‹a ≠ bot› add-diff-cancel-left' assms(1) calculation diff-less
          diff-zero gr0I last-conv-nth length-greater-0-conv less-Suc-eq-le
          less-or-eq-imp-le mod-less valid-list-def)
  ultimately have suffix s (bwt-perm s ! k) = a # suffix s (sa s ! k)
    by (metis Nat.add-diff-assoc2 Suc-leI Suc-lessD Suc-pred ‹k < length s› mod-add-self2
          mod-less sa-nth-ex suffix-cons-Suc)
 }
 ultimately show a # ys < prefix (suffix s (bwt-perm s ! k)) (length (a # ys))
  by simp
qed
```

have *0 < count-list (sort bs) a*
by (*metis ‹a ∈ set s› assms(2) bwt-sa-perm count-list-perm gr0I notin-count sort-perm*)

with *select-sorted-0*[*of sort bs a*] *bij-betw-sort-idx-ex*[*of sort bs s a*]
have *select (sort bs) a 0 = card {i. i < length s ∧ s ! i < a}*
by (*metis* (*mono-tags*, *lifting*) *bwt-sa-perm assms(2) bij-betw-same-card*
sorted-list-of-multiset-mset sorted-sort)

have *select (sort bs) a 0 + count-list s a ≤ length s*
by (*metis backward-lookup-upper*[*OF assms(2)*] *assms(2) backward-lookup.simps bwt-sa-perm*
count-list-perm rank-length)

have *count-list s a = count-list (sort s) a*
by (*metis count-list-perm perm-map-nths-sa sort-eq-map-nths-sa*)

have *0 < count-list s a*
by (*simp add*: *‹a ∈ set s› count-in*)

note *sort-nth-a = sort-nth-a*[*OF assms(2) ‹0 < count-list s a›*]

have *sort-bwt-perm-nth*:
⋀k. ⟦k < length s; *sort s ! k = a⟧ ⟹*
∃k′ < length s. sa s ! k = bwt-perm s ! k′ ∧ rank (sort s) a k = rank (bwt-sa s) a k′
proof −
fix *k*
assume *k < length s sort s ! k = a*
moreover
have *∃k′ < length s. sa s ! k = bwt-perm s ! k′*
by (*metis ‹k < length s› ex-bwt-perm-nth sa-nth-ex*)
then obtain *k′* **where**
k′ < length s sa s ! k = bwt-perm s ! k′
by *blast*
moreover
from *sa-bwt-perm-same-rank*[*OF assms(1) ‹k < length s› ‹k′ < length s›*]
have *rank (sort s) a k = rank (bwt-sa s) a k′*
using *‹k < length s› ‹sort s ! k = a› bwt-perm-s-nth sort-sa-nth calculation* **by** *force*
ultimately show
∃k′ < length s. sa s ! k = bwt-perm s ! k′ ∧ rank (sort s) a k = rank (bwt-sa s) a k′
by *blast*
qed

let *?j′ = select (sort bs) a 0 + rank bs a ?j*

```
have ?j' ≤ length ?xs
  using ‹?j' ≤ length s› sa-length by fastforce
moreover
have ⋀k. ⟦k < length s; k < select (sort bs) a 0 + rank bs a ?j⟧ ⟹
        prefix (suffix s (sa s ! k)) (length (a # ys)) ≤ a # ys
proof −
  fix k
  assume k < length s k < select (sort bs) a 0 + rank bs a ?j

  have k < select (sort bs) a 0 ∨ select (sort bs) a 0 ≤ k
    using linorder-le-less-linear by blast
  moreover
  have k < select (sort bs) a 0 ⟹ prefix (suffix s (sa s ! k)) (length (a # ys)) ≤ a # ys
  proof −
    assume k < select (sort bs) a 0
    with sa-suffix-nth-less[of k s a]
    have suffix s (sa s ! k) < [a]
      using ‹select (sort bs) a 0 = card {i. i < length s ∧ s ! i < a}› by force
    hence s ! (sa s ! k) < a
      by (metis Cons-less-Cons ‹k < length s› not-less-Nil sa-nth-ex suffix-cons-Suc)
    then show ?thesis
      using ‹k < length s› sa-nth-ex suffix-cons-ex by fastforce
  qed
  moreover
  have select (sort bs) a 0 ≤ k ⟹ prefix (suffix s (sa s ! k)) (length (a # ys)) ≤ a # ys
  proof −
    assume select (sort bs) a 0 ≤ k

    from sorted-card-rank-idx[of sort s k]
    have k = card {j. j < length (sort s) ∧ sort s ! j < sort s ! k} +
             rank (sort s) (sort s ! k) k
      using ‹k < length s› by fastforce
    moreover
    from sort-nth-a[OF ‹k < length s› ‹select (sort bs) a 0 ≤ k›]
    have sort s ! k = a
     using ‹k < select (sort bs) a 0 + rank bs a ?j› ‹rank bs a ?j ≤ count-list s a› by linarith
    ultimately have k = select (sort bs) a 0 + rank (sort s) a k
     using ‹select (sort bs) a 0 = -› ‹k < length s› sa-card-rank-s-idx sort-sa-nth by fastforce
    hence rank (sort s) a k < rank bs a ?j
      using ‹k < select (sort bs) a 0 + rank bs a ?j› by linarith
    moreover
    from sort-bwt-perm-nth[OF ‹k < length s› ‹sort s ! k = a›]
    obtain k' where
```

```
    k' < length s sa s ! k = bwt-perm s ! k' rank (sort s) a k = rank (bwt-sa s) a k'
     by blast
   ultimately have k' < ?j
    by (metis (no-types, lifting) assms(2) leD leI rank-idx-mono)
   then show ?thesis
    by (metis (no-types, lifting) P1' ‹k < length s› ‹k' < length s› bwt-perm-nth bwt-sa-nth
                                  ‹sa s ! k = bwt-perm s ! k'› ‹sort s ! k = a› assms(2)
                                  sort-sa-nth)
  qed
  ultimately show prefix (suffix s (sa s ! k)) (length (a # ys)) ≤ a # ys
   using order-less-imp-le by blast
qed
hence ∀ k < ?j'. prefix (?xs ! k) (length (a # ys)) ≤ a # ys
 using ‹?j' ≤ length s› sa-length by fastforce
moreover
have ⋀k. ⟦k < length s; select (sort bs) a 0 + rank bs a ?j ≤ k⟧ ⟹
        a # ys < prefix (suffix s (sa s ! k)) (length (a # ys))
proof −
 fix k
 assume k < length s select (sort bs) a 0 + rank bs a ?j ≤ k

 have k < select (sort bs) a 0 + count-list s a ∨ select (sort bs) a 0 + count-list s a ≤ k
  using linorder-le-less-linear by blast
 moreover
 have select (sort bs) a 0 + count-list s a ≤ k ⟹
      a # ys < prefix (suffix s (sa s ! k)) (length (a # ys))
 proof −
  assume select (sort bs) a 0 + count-list s a ≤ k
  hence ∃ k'. k = select (sort bs) a 0 + k' ∧ count-list s a ≤ k'
   using nat-le-iff-add by auto
  hence a < sort s ! k
   by (metis (mono-tags, lifting) ‹a ∈ set s› ‹count-list s a = count-list (sort s) a›
                                  ‹k < length s› assms(2) bwt-sa-perm count-in length-sort
                                  select-sorted-0 sorted-list-of-multiset-mset
                                  sorted-nth-gr-gen sorted-sort)
  moreover
  have ∃ zs. suffix s (sa s ! k) = (sort s ! k) # zs
   by (metis ‹k < length s› sa-nth-ex sort-sa-nth suffix-cons-ex)
  ultimately show ?thesis
   by fastforce
 qed
 moreover
 have k < select (sort bs) a 0 + count-list s a ⟹
      a # ys < prefix (suffix s (sa s ! k)) (length (a # ys))
```

```
  proof −
    assume k < select (sort bs) a 0 + count-list s a

    from sorted-card-rank-idx[of sort s k]
    have k = card {j. j < length (sort s) ∧ sort s ! j < sort s ! k} +
              rank (sort s) (sort s ! k) k
      using ‹k < length s› by fastforce
    moreover
    from sort-nth-a[OF ‹k < length s›]
    have sort s ! k = a
      using ‹select (sort bs) a 0 + rank bs a ?j ≤ k› ‹k < select (sort bs) a 0 + count-list s
a›
      by linarith
    ultimately have k = select (sort bs) a 0 + rank (sort s) a k
      using ‹k < length s› ‹select (sort bs) a 0 = card {i. i < length s ∧ s ! i < a}›
          sa-card-rank-s-idx sort-sa-nth by auto
    hence rank bs a ?j ≤ rank (sort s) a k
      using ‹select (sort bs) a 0 + rank bs a ?j ≤ k› by linarith
    moreover
    from sort-bwt-perm-nth[OF ‹k < length s› ‹sort s ! k = a›]
    obtain k′ where
      k′ < length s sa s ! k = bwt-perm s ! k′ rank (sort s) a k = rank (bwt-sa s) a k′
      by blast
    ultimately have rank bs a ?j ≤ rank bs a k′
      using assms(2) by argo

    have ?j ≤ k′
    proof (rule ccontr)
      assume ¬ ?j ≤ k′
      hence k′ < ?j
        using linorder-le-less-linear by blast
      hence rank bs a k′ < rank bs a ?j
        by (metis (no-types, lifting) ‹k < length s› ‹k′ < length s› rank-less sort-sa-nth
                                  ‹sa s ! k = bwt-perm s ! k′› ‹sort s ! k = a› assms(2)
                                  bwt-perm-s-nth bwt-sa-length)
      with ‹rank bs a ?j ≤ rank bs a k′›
      show False
        using linorder-not-less by blast
    qed
    with P2′[OF ‹k′ < length s›]
    show ?thesis
      by (metis ‹k < length s› ‹k′ < length s› ‹sa s ! k = bwt-perm s ! k′› ‹sort s ! k = a›
            assms(2) bwt-perm-s-nth sort-sa-nth)
  qed
```

```
    ultimately show a # ys < prefix (suffix s (sa s ! k)) (length (a # ys))
      by blast
  qed
  hence ∀ k < length ?xs. ?j' ≤ k ⟶ a # ys < prefix (?xs ! k) (length (a # ys))
    by (simp add: sa-length)
  ultimately show ?thesis
    using patsearch-idx-endpoint[OF sa-suffix-sorted[of s], of ?j' a # ys]
        backward-lookup.simps
    by presburger
qed
```

A.8.2 Multiple backward Look-up

```
lemma backward-lookups-app:
  backward-lookups bs (xs @ ys) i = backward-lookups bs ys (backward-lookups bs xs i)
  by (induct xs arbitrary: i; simp)

lemma backward-lookups-cons:
  backward-lookups bs (y # ys) i = backward-lookups bs ys (backward-lookup bs y i)
  by force

lemma backward-lookups-idx-mono:
  j ≤ i ⟹ backward-lookups bs xs j ≤ backward-lookups bs xs i
proof (induct xs arbitrary: i j)
  case Nil
  then show ?case
    by simp
next
  case (Cons a xs)
  then show ?case
    using backward-lookup-idx-mono by force
qed

lemma backward-lookups-upper:
  assumes bs = bwt-sa s
  and     i ≤ length s
shows backward-lookups bs (rev ys) i ≤ length s
proof (induct ys)
  case Nil
  then show ?case
    by (simp add: assms(2))
next
  case (Cons a ys)
```

have *backward-lookups bs (rev (a # ys)) i =*
backward-lookups bs [a] (backward-lookups bs (rev ys) i)
by (*simp add*: *backward-lookups-app*)
moreover
from *Cons*
have *backward-lookups bs (rev ys) i ≤ length s*
by *simp*
ultimately show *?case*
using *backward-lookup-upper*[*OF assms*(*1*), *of a*] *Cons.prems* **by** *auto*
qed

Current backward lookup will fail for patterns with bot in it, e.g. *bot # xs* when *xs ≠ []*. This isn't a problem in the real world as *bot* would indicate the end of string

theorem *backward-lookups-correct*:
assumes *valid-list s*
and *bs = bwt-sa s*
shows ⟦*valid-list ys ∨ bot ∉ set ys*⟧ ⟹
patsearch-set s ys =
nth (sa s) ‘ {backward-lookups bs (rev ys) 0 ..<backward-lookups bs (rev ys) (length s)}
proof (*induct ys*)
case *Nil*
then show *?case*
using *nth-sa-upt-set patsearch-set-nil* **by** *simp*
next
case (*Cons a ys*)
note *IH = this*

let *?i = backward-lookups bs (rev ys) 0*
let *?j = backward-lookups bs (rev ys) (length s)*

have *valid-list ys ∨ bot ∉ set ys*
by (*metis IH*(*2*) *empty-iff empty-set list.set-intros*(*2*) *valid-list-consD*)

note *A = IH*(*1*)[*OF* ‹- ∨ *bot ∉ set ys*›]

let *?k = backward-lookups bs (rev (a # ys)) 0*
let *?l = backward-lookups bs (rev (a # ys)) (length s)*

have *B1*: *?k = backward-lookups bs [a] ?i ?l = backward-lookups bs [a] ?j*
by (*simp add*: *backward-lookups-app*)+
hence *B2*: *?k = backward-lookup bs a ?i ?l = backward-lookup bs a ?j*
by *simp*+

let *?i′* = *patsearch-idx-start* (*map* (*suffix s*) (*sa s*)) *ys*
let *?j′* = *patsearch-idx-end* (*map* (*suffix s*) (*sa s*)) *ys*

have *P*: *patsearch-set s ys* ≠ {} ⟹ *?i* = *?i′* ∧ *?j* = *?j′*
proof −
assume *patsearch-set s ys* ≠ {}
hence *?i* < *?j*
using *A atLeastLessThan-empty-iff* **by** *blast*

from *backward-lookups-upper*[*OF assms*(*2*)]
have *?j* ≤ *length s*
by *simp*
moreover
have *?j′* ≤ *length s*
by (*simp add*: *sa-length*)
ultimately have *inj-on* (*nth* (*sa s*)) ({*?i*..<*?j*} ∪ {*?i′*..<*?j′*})
using *inj-on-nth-sa-upt* **by** *blast*
hence {*?i*..<*?j*} = {*?i′*..<*?j′*}
by (*metis* (*no-types*, *lifting*) *patsearch-set-eq-sa-range*[*of s ys*] *A inj-on-image-eq-iff*
sup.cobounded1 sup-ge2)
then show *?thesis*
using *Ico-eq-Ico*[*of ?i′ ?j′ ?i ?j*] ‹*?i* < *?j*› **by** *presburger*
qed

show *?case*
proof (*cases patsearch-set s* (*a* # *ys*) = {})
assume *C*: *patsearch-set s* (*a* # *ys*) = {}
show *?thesis*
proof (*cases patsearch-set s ys* = {})
assume *patsearch-set s ys* = {}
hence *?j* ≤ *?i*
by (*simp add*: *A*)
hence *?l* ≤ *?k*
using *B1 backward-lookups-idx-mono* **by** *presburger*
then show *?thesis*
by (*simp add*: ‹*patsearch-set s* (*a* # *ys*) = {}›)
next
assume *D*: *patsearch-set s ys* ≠ {}
with *P backward-lookup-cons-fail*[*OF assms*(*1*,*2*) *D C*]
have *?l* ≤ *?k*
using *B2*(*1*) *B2*(*2*) *Cons.prems* **by** *argo*
then show *?thesis*
using *C* **by** *force*
qed

```
  next
    assume C: patsearch-set s (a # ys) ≠ {}
    hence D: patsearch-set s ys ≠ {}
      by (meson patsearch-set-cons-not-empty)
    with P
    have ?i = ?i' ?j = ?j
      by blast+
    then show ?thesis
      using D P patsearch-set-eq-sa-range B2
            backward-lookup-cons-success-start[OF assms D C]
            backward-lookup-cons-success-end[OF assms D C]
            Cons.prems
      by presburger
  qed
qed
```

A.8.3 Search Pattern Backward Range

```
lemma backward-lookups-eq-search-backward:
  search-backward bs cs (i, j) = (backward-lookups bs cs i, backward-lookups bs cs j)
  by (induct cs arbitrary: i j; simp)

lemma backward-lookups-eq-fst-search-backward:
  fst (search-backward bs cs (i, j)) = backward-lookups bs cs i
  by (simp add: backward-lookups-eq-search-backward)

lemma backward-lookups-eq-snd-search-backward:
  snd (search-backward bs cs (i, j)) = backward-lookups bs cs j
  by (simp add: backward-lookups-eq-search-backward)

primrec upt-tuple-set
  where
upt-tuple-set (x, y) = {x..<y}

primrec upt-tuple-list
  where
upt-tuple-list (x, y) = [x..<y]

lemma search-backward-app:
  search-backward bs (xs @ ys) (i, j) =
   search-backward bs ys (search-backward bs xs (i, j))
  by (induct xs arbitrary: i j; simp)

lemma search-backward-idx-mono:
```

```
  j ≤ i ⟹ snd (search-backward bs xs (i, j)) ≤ fst (search-backward bs  xs (i, j))
proof (induct xs rule: rev-induct)
  case Nil
  then show ?case
    by simp
next
  case (snoc a xs)
  then show ?case
    using search-backward-app[of bs [a] xs i j]
    by (metis backward-lookup-idx-mono search-backward.simps(1,2) search-backward-app
          fst-conv prod.collapse snd-conv)
qed

theorem search-backward-correct:
  assumes valid-list s
  and     bs = bwt-sa s
  and     valid-list ys ∨ bot ∉ set ys
shows patsearch-set s ys =
      nth (sa s) ‘ upt-tuple-set (search-backward bs (rev ys) (0, (length s)))
  using backward-lookups-eq-search-backward[of bs rev ys 0 length s]
       backward-lookups-correct[OF assms]
  by auto
```

A.8.4 Search Pattern Backward Range Early Exit

```
lemma search-backward-eq-early-exit:
  upt-tuple-set (search-backward bs cs (i, j)) =
   upt-tuple-set (search-backward-early-exit bs cs (i, j))
proof (induct cs arbitrary: i j)
  case Nil
  then show ?case
    by simp
next
  case (Cons a cs i j)
  note IH = this

  let ?i = backward-lookup bs a i
  let ?j = backward-lookup bs a j

  show ?case
  proof (cases i < j)
    assume i < j
    hence upt-tuple-set (search-backward-early-exit bs (a # cs) (i, j)) =
          upt-tuple-set (search-backward-early-exit bs cs (?i, ?j))
```

```
    by simp
  moreover
  have upt-tuple-set (search-backward bs (a # cs) (i, j)) =
      upt-tuple-set (search-backward bs cs (?i, ?j))
    by simp
  moreover
  note IH[of ?i ?j]
  ultimately show ?case
    by argo
 next
  assume ¬ i < j
  hence upt-tuple-set (search-backward-early-exit bs (a # cs) (i, j)) = {}
    by force
  moreover
  have upt-tuple-set (search-backward bs (a # cs) (i, j)) = {}
    by (metis backward-lookups-eq-fst-search-backward
          backward-lookups-eq-search-backward
          ‹¬ i < j› backward-lookups-eq-snd-search-backward empty-set leI
          search-backward-idx-mono set-upt upt-eq-Nil-conv upt-tuple-set.simps)
  ultimately show ?case
    by blast
 qed
qed

theorem search-backward-early-exit-correct:
 assumes valid-list s
 and     bs = bwt-sa s
 and     valid-list ys ∨ bot ∉ set ys
shows patsearch-set s ys =
     nth (sa s) ‘ upt-tuple-set (search-backward-early-exit bs (rev ys) (0, (length s)))
 using search-backward-correct[OF assms] search-backward-eq-early-exit by presburger
```

A.8.5 Backward Search

```
theorem backward-search-correct:
 assumes valid-list s
 and     bs = bwt-sa s
 and     valid-list ys ∨ bot ∉ set ys
shows patsearch-set s ys =
     nth (sa s) ‘ upt-tuple-set (backward-search bs ys)
 by (simp add: bwt-sa-length Suffix-Array-General-axioms assms search-backward-early-exit-correct)

end
```

end

Bibliography

[1] M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. Symp. String Processing and Information Retrieval*, volume 2476 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2002. doi: 10.1007/3-540-45735-6_4.

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

[3] O. Abrahamsson, S. Ho, H. Kanabar, R. Kumar, M. O. Myreen, M. Norrish, and Y. K. Tan. Proof-producing synthesis of CakeML from monadic HOL functions. *Journal of Automated Reasoning*, 64(7):1287–1306, 2020. doi: 10.1007/S10817-020-09559-8.

[4] O. Abrahamsson, M. O. Myreen, R. Kumar, and T. Sewell. Candle: A verified implementation of HOL light. In *Proc. Conf. Interactive Theorem Proving*, volume 237 of *LIPIcs*, pages 3:1–3:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ITP.2022.3.

[5] Therauputic Goods Administration. Actual and potential harm caused by medical software, July 2020. URL https://www.tga.gov.au/resource/actual-and-potential-harm-caused-medical-software.

[6] R. Affeldt, J. Garrigue, X. Qi, and K. Tanaka. Proving tree algorithms for succinct data structures. In *Proc. Conf. Interactive Theorem Proving*, volume 141 of *LIPIcs*, pages 5:1–5:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi: 10.4230/LIPIcs.ITP.2019.5.

[7] A. Ahmed. Compositional compiler verification for a multi-language world. In *Proc. Conf. Formal Structures for Computation and Deduction*, volume 52 of *LIPIcs*, pages 1:1–1:1. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.FSCD.2016.1.

[8] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi: 10.1145/360825.360855.

[9] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. C. Murray, G. Klein, and G. Heiser. Cogent: Verifying high-assurance file system implementations. In *Proc. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 175–188. ACM, 2016. doi: 10.1145/2872362.2872404.

[10] Sidney Amani. *A Methodology for Trustworthy File Systems*. PhD thesis, University of New South Wales, August 2016. doi: 10.26190/unsworks/19093.

[11] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. CertiCoq: A verified compiler for Coq. In *Proc. Work. Coq for Programming Languages*, 2017. URL http://conf.researchr.org/event/CoqPL-2017/main-certicoq-a-verified-compiler-for-coq.

[12] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM Journal on Computing*, 15(1):98–105, 1986. doi: 10.1137/0215007.

[13] A. W. Appel, L. Beringer, A. Chlipala, B. C. Pierce, Z. Shao, S. Weirich, and S. Zdancewic. Position paper: The science of deep specification. *Philosphical Transactions of the Royal Society A*, 375(2104), 2017. doi: 10.1098/rsta.2016.0331.

[14] Andrew W. Appel. *Verified Functional Algorithms*, volume 3 of *Software Foundations*. Electronic textbook, 2024. Version 1.5.5, http://softwarefoundations.cis.upenn.edu.

[15] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. Part I: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, September 1977. doi: 10.1215/ijm/1256049011.

[16] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. Part II: Reducibility. *Illinois Journal of Mathematics*, 21(3):491–567, September 1977. doi: 10.1215/ijm/1256049012.

[17] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: A logical framework based on the λΠ-calculus modulo theory. *CoRR*, abs/2311.07185, 2023. doi: 10.48550/ARXIV.2311.07185.

[18] J. Bahne, N. Bertram, M. Böcker, J. Bode, J. Fischer, H. Foot, F. Grieskamp, F. Kurpicz, M. Löbel, O. Magiera, R. Pink, D. Piper, and C. Poeplau. SACABench: Benchmarking suffix array construction. In *Proc. Symp. String Processing and Information Retrieval*, volume 11811 of *Lecture Notes in Computer Science*, pages 407–416. Springer, 2019. doi: 10.1007/978-3-030-32686-9_29.

[19] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Proc. Conf. Formal Methods in Computer-Aided Design*, pages 35–42. IEEE Computer Society, 2010. URL https://ieeexplore.ieee.org/document/5770931/.

[20] M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. KIV 3.0 for provably correct systems. In *Proc. Work. Current Trends in Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*, pages 330–337. Springer, 1998. doi: 10.1007/3-540-48257-1_23.

[21] J. G. P. Barnes. *High Integrity Software — The SPARK Approach to Safety and Security*. Addison-Wesley, 2003. ISBN 978-0-321-13616-9.

[22] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi: 10.1007/BF00288683.

[23] B. Beckert, R. Bubel, D. Drodt, R. Hähnle, F. Lanzinger, W. Pfeifer, M. Ulbrich, and A. Weigl. The Java verification tool KeY: A tutorial. In *Proc. Symp. Formal Methods*, volume 14934 of *Lecture Notes in Computer Science*, pages 597–623. Springer, 2024. doi: 10.1007/978-3-031-71177-0_32.

[24] B. Beckert, P. Sanders, M. Ulbrich, J. Wiesler, and S. Witt. Formally verifying an efficient sorter. In *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, volume 14570 of *Lecture Notes in Computer Science*, pages 268–287. Springer, 2024. doi: 10.1007/978-3-031-57246-3_15.

[25] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi: 10.1007/978-3-662-07964-5.

[26] R. S. Bird and S. Mu. Inverting the Burrows-Wheeler transform. *Journal of Functional Programming*, 14(6):603–612, 2004. doi: 10.1017/S0956796804005118.

[27] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *Proc. Symp. Frontiers of Combining Systems*, volume 6989 of *Lecture Notes in Computer Science*, pages 12–27. Springer, 2011. doi: 10.1007/978-3-642-24364-6_2.

[28] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013. doi: 10.1007/S10817-013-9278-5.

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

[30] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda — A functional language with dependent types. In *Proc. Conf. Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009. doi: 10.1007/978-3-642-03359-9_6.

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

[32] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press, 1979. ISBN 978-0-12-122952-8.

[33] J. Breitner, A. Spector-Zabusky, Y. Li, C. Rizkallah, J. Wiegley, J. M. Cohen, and S. Weirich. Ready, set, verify! applying hs-to-coq to real-world Haskell code. *Journal of Functional Programming*, 31:e5, 2021. doi: 10.1017/S0956796820000283.

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

[35] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC-RR-124, Digital SRC Research Report, 1994.

[36] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940. doi: 10.2307/2266170.

[37] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proc. Conf. Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009. doi: 10.1007/978-3-642-03359-9_2.

[38] T2T Consortium. T2t-chm13v2.0 - genome - assembly - ncbi, jan 2022. URL https://www.ncbi.nlm.nih.gov/assembly/GCF_009914755.1/#/st. Accessed: 27/04/22.

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. ISBN 978-0-262-03384-8.

[40] A. J. Cox, M. J. Bauer, T. Jakobi, and G. Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012. doi: 10.1093/bioinformatics/bts173.

[41] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. ISBN 978-0-521-84899-2.

[42] B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice*. Addison-Wesley Publishing Company, 1st edition, 2009. ISBN 0136072240.

[43] J. S. Culpepper and A. Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems*, 29(1):1:1–1:25, 2010. doi: 10.1145/1877766.1877767.

[44] N. T. Curran, T. Kennings, and K. G. Shin. Analysis and prevention of MCAS-induced crashes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3382–3394, 2024. doi: 10.1109/TCAD.2024.3438105.

[45] S. de Gouw, F. S. de Boer, R. Bubel, R. Hähnle, J. Rot, and D. Steinhöfel. Verifying OpenJDK's sort method for generic collections. *Journal of Automated Reasoning*, 62(1):93–126, 2019. doi: 10.1007/S10817-017-9426-4.

[46] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proc. Symp. Principles of Programming Languages*, pages 689–700. ACM, 2015. doi: 10.1145/2676726.2677006.

[47] C. P. Devadoss and B. Sankaragomathi. Near lossless medical image compression using block BWT-MTF and hybrid fractal compression techniques. *Cluster Computing*, 22(5):12929–12937, 2019. doi: 10.1007/S10586-018-1801-3.

[48] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi: 10.1007/BF01386390.

[49] E. W. Dijkstra. On the reliability of programs. Circulated Privately, N.D. URL http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF.

[50] M. Dowson. The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997. doi: 10.1145/251880.251992.

[51] I. Dramnesc, T. Jebelean, and S. Stratulat. Certification of sorting algorithms using Theorema and Coq. In *Proc. Symp. Symbolic Computation in Software Science*, volume 14991 of *Lecture Notes in Computer Science*, pages 38–56. Springer, 2024. doi: 10.1007/978-3-031-69042-6_3.

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

[53] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. Symp. Foundations of Computer Science*, pages 137–143. IEEE Computer Society, 1997. doi: 10.1109/SFCS.1997.646102.

[54] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. Symp. Foundations of Computer Science*, pages 390–398. IEEE Computer Society, 2000. doi: 10.1109/SFCS.2000.892127.

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

[56] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):4:1–4:33, 2009. doi: 10.1145/1613676.1613680.

[57] J. Filliâtre. Proof of imperative programs in type theory. In *Proc. Int. Conf. Types for Proofs and Programs*, volume 1657 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 1998. doi: 10.1007/3-540-48167-2_6.

[58] J. Fischer and F. Kurpicz. Dismantling DivSufSort. In *Proc. Conf. Prague Stringtology*, pages 62–76. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2017. doi: arXiv:1710.01896.

[59] Y. Forster, M. Sozeau, and N. Tabareau. Verified extraction from Coq to OCaml. *Proceedings of the ACM on Programming Languages*, 8(PLDI):52–75, 2024. doi: 10.1145/3656379.

[60] R. Giancarlo, A. Restivo, and M. Sciortino. From first principles to the Burrows and Wheeler Transform and beyond, via combinatorial optimization. *Theoretical of Computer Science*, 387(3):236–248, 2007. doi: 10.1016/J.TCS.2007.07.019.

[61] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. Symp. Experimental Algorithms*,

volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi: 10.1007/978-3-319-07959-2_28.

[62] G. Gonthier. The Four Color Theorem: Engineering of a formal proof. In *Proc. Asian Symposium on Computer Mathematics*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007. doi: 10.1007/978-3-540-87827-8_28.

[63] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[64] K. Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. In *Proc. Conf. Prague Stringtology*, pages 111–125. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2019. doi: arXiv:1703.01009v5.

[65] S. Gouëzel and V. Shchur. A corrected quantitative version of the Morse lemma. *Journal of Functional Analysis*, 277(4):1258–1268, 2019. doi: 10.1016/j.jfa.2019.02.021.

[66] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *Proc. Conf. Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2012. doi: 10.1007/978-3-642-32347-8_8.

[67] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *Proc. Conf. Programming Language Design and Implementation*, pages 429–439. ACM, 2014. doi: 10.1145/2594291.2594296.

[68] S. Griebel. Binary heaps for IMP2. *Archive of Formal Proofs*, June 2019. ISSN 2150-914x. https://isa-afp.org/entries/IMP2_Binary_Heap.html, Formal proof development.

[69] D. Gusfield. *Algorithms on Strings, Trees, and Sequences — Computer Science and Computational Biology*. Cambridge University Press, 1997. doi: 10.1017/CBO9780511574931.

[70] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Proc. Symp. Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2010. doi: 10.1007/978-3-642-12251-4_9.

[71] F. Haftmann and M. Wenzel. Local theory specifications in Isabelle/Isar. In *Proc. Int. Conf. Types for Proofs and Programs*, volume 5497 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2008. doi: 10.1007/978-3-642-02444-3_10.

[72] J. Harrison. HOL light: An overview. In *Proc. Conf. Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009. doi: 10.1007/978-3-642-03359-9_4.

[73] Fabian Hellauer and Peter Lammich. The string search algorithm by Knuth, Morris and Pratt. *Archive of Formal Proofs*, December 2017. ISSN 2150-914x. https://isa-afp.org/entries/Knuth_Morris_Pratt.html, Formal proof development.

[74] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259.

[75] W. Hon, T. W. Lam, K. Sadakane, W. Sung, and S. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1): 23–36, 2007. doi: 10.1007/S00453-006-1228-8.

[76] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2003. ISBN 978-0-321-21029-6.

[77] L. Hupel and T. Nipkow. A verified compiler from Isabelle/HOL to CakeML. In *Proc. European Symposium on Programming*, volume 10801 of *Lecture Notes in Computer Science*, pages 999–1026. Springer, 2018. doi: 10.1007/978-3-319-89884-1_35.

[78] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proc. Symp. String Processing and Information Retrieval*, pages 81–88. IEEE Computer Society, 1999. doi: 10.1109/SPIRE.1999.796581.

[79] D. Jackson. Alloy: A language and tool for exploring software designs. *Communications of the ACM*, 62(9):66–76, 2019. doi: 10.1145/3338843.

[80] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Proc. Symp. NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011. doi: 10.1007/978-3-642-20398-5_4.

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

[82] F. Kammüller, M. Wenzel, and L. C. Paulson. Locales — A sectioning concept for Isabelle. In *Proc. Conf. Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 1999. doi: 10.1007/3-540-48256-3_11.

[83] J. Kang, Y. Kim, C. Hur, D. Dreyer, and V. Vafeiadis. Lightweight verification of separate compilation. In *Proc. Symp. Principles of Programming Languages*, pages 178–190. ACM, 2016. doi: 10.1145/2837614.2837642.

[84] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. Int. Conf. Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer, 2003. doi: 10.1007/3-540-45061-0_73.

[85] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. doi: 10.1147/RD.312.0249.

[86] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. Symp. Theory of Computing*, pages 125–136. ACM, 1972. doi: 10.1145/800152.804905.

[87] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. Symp. Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001. doi: 10.1007/3-540-48194-X_17.

[88] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods. Springer New York, 2000. doi: 10.1007/978-1-4615-4449-4.

[89] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing: Applicable Formal Methods*, 27(3):573–609, 2015. doi: 10.1007/S00165-014-0326-7.

[90] G. Klein. Operating system verification — An overview. *Sadhana*, 34:27–69, 2009. doi: 10.1007/s12046-009-0002-4.

[91] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. A. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proc. Symp. Operating Systems Principles*, pages 207–220. ACM, 2009. doi: 10.1145/1629575.1629596.

[92] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 2nd edition, 1998. ISBN 0201896850.

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

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

[95] J. Korkut, K. Stark, and A. W. Appel. A verified foreign function interface between Coq and C. *Proceedings of the ACM on Programming Languages*, 9 (POPL):687–717, 2025. doi: 10.1145/3704860.

[96] R. Ktistakis, P. Fournier-Viger, S. J. Puglisi, and R. Raman. Succinct BWT-based sequence prediction. In *Proc. Conf. Database and Expert Systems Applications*, volume 11707 of *Lecture Notes in Computer Science*, pages 91–101. Springer, 2019. doi: 10.1007/978-3-030-27618-8_7.

[97] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Proc. Symp. Principles of Programming Languages*, pages 179–192. ACM, 2014. doi: 10.1145/2535838.2535841.

[98] S. Kurtz. Reducing the space requirement of suffix trees. *Software: Practical Experience*, 29(13):1149–1171, 1999. doi: 10.1002/(SICI)1097-024X(199911)29:13<1149::AID-SPE274>3.0.CO;2-O.

[99] P. Lammich. Automatic data refinement. In *Proc. Conf. Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2013. doi: 10.1007/978-3-642-39634-2_9.

[100] P. Lammich. Generating verified LLVM from Isabelle/HOL. In *Proc. Conf. Interactive Theorem Proving*, volume 141 of *LIPIcs*, pages 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi: 10.4230/LIPICS.ITP.2019.22.

[101] P. Lammich. Refinement to Imperative HOL. *Journal of Automated Reasoning*, 62(4):481–503, 2019. doi: 10.1007/S10817-017-9437-1.

[102] P. Lammich. Efficient verified implementation of Introsort and Pdqsort. In *Proc. Int. Joint Conf. Automated Reasoning*, volume 12167 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, 2020. doi: 10.1007/978-3-030-51054-1_18.

[103] P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In *Proc. Conf. Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010. doi: 10.1007/978-3-642-14052-5_24.

[104] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X.

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

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

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

[108] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proc. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi: 10.1007/978-3-642-17511-4_20.

[109] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010. doi: 10.1007/978-3-642-12002-2_26.

[110] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. doi: 10.1007/S10817-009-9155-4.

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

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

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

[114] R. A. Lippert, C. M. Mobarry, and B. Walenz. A space-efficient construction of the Burrows-Wheeler transform for genomic data. *Journal of Computational Biology*, 12(7):943–951, 2005. doi: 10.1089/CMB.2005.12.943.

[115] D. MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust.* MIT Press, 2001. ISBN 9780262133937.

[116] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. Symp. Combinatorial Pattern Matching*, volume 3537 of *Lecture Notes in Computer Science*, pages 45–56. Springer, 2005. doi: 10.1007/11496656_5.

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

[118] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008. doi: 10.1017/CBO9780511809071.

[119] G. Manzini. An analysis of the Burrows-Wheeler Transform. *Journal of the ACM*, 48(3):407–430, 2001. doi: 10.1145/382780.382782.

[120] D. Matichuk, T. Murray, J. Andronick, R. Jeffery, G. Klein, and M. Staples. Empirical study towards a leading indicator for cost of formal software verification. In *Proc. Int. Conf. Software Engineering*, volume 1 of *ICSE*, pages 722–732. IEEE Computer Society, 2015. doi: 10.1109/ICSE.2015.85.

[121] S. McConnell. *Code Complete — A Practical Handbook of Software Construction.* Microsoft Press, 2nd edition, 2004. ISBN 9780735619678.

[122] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi: 10.1145/321941.321946.

[123] R. Milner, M. Tofte, and R. Harper. *Definition of Standard ML.* MIT Press, 1990. ISBN 978-0-262-63132-7.

[124] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml — Functional Programming for the Masses.* O'Reilly, 2013. ISBN 978-1-4493-2391-2.

[125] A. Moffat and A. Turpin. *Compression and Coding Algorithms.* Kluwer Academic Publishers, 2002. ISBN 0-7923-7668-4.

[126] A. Mohan, W. X. Leow, and A. Hobor. Functional correctness of C implementations of Dijkstra's, Kruskal's, and Prim's algorithms. In *Proc. Conf. Computer Aided Verification*, volume 12760 of *Lecture Notes in Computer Science*, pages 801–826. Springer, 2021. doi: 10.1007/978-3-030-81688-9_37.

[127] J. S. Moore and M. Martinez. 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*, volume 22 of *NATO Science for Peace*

and Security Series - D: Information and Communication Security, pages 267–284. IOS Press, 2009. doi: 10.3233/978-1-58603-976-9-267.

[128] L. Moura and N. S. Bjørner. Z3: An efficient SMT solver. In *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24.

[129] L. Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *Proc. Conf. Automatic Deduction*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

[130] E. Mullen, S. Pernsteiner, J. R. Wilcox, Z. Tatlock, and D. Grossman. Œuf: Minimizing the Coq extraction TCB. In *Proc. Conf. Certified Programs and Proofs*, pages 172–185. ACM, 2018. doi: 10.1145/3167089.

[131] M. O. Myreen. Functional programs: Conversions between deep and shallow embeddings. In *Proc. Conf. Interactive Theorem Proving*, volume 7406 of *Lecture Notes in Computer Science*, pages 412–417. Springer, 2012. doi: 10.1007/978-3-642-32347-8_29.

[132] M. O. Myreen and S. Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming*, 24(2-3):284–315, 2014. doi: 10.1017/S0956796813000282.

[133] A. Nasir. Intel i9-14900KF overclocker clinches CPU frequency world record at 9.12 GHz — Wytiwx joins Elmor as the only person to push a CPU past 9 GHz. *Tom's Hardware Guide*, 2025. URL https://www.tomshardware.com/pc-components/cpus/intel-i9-14900kf-overclocker-clinches-cpu-frequency-world-record-at-9-12-ghz-wytiwx-joins-elmor-as-the-only-person-to-push-a-cpu-past-9-ghz#:~:text=With%20a%20core%20voltage%20of,margins%20at%20just%204%20MHz.

[134] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. doi: 10.1016/J.JDA.2013.07.004.

[135] G. Navarro. *Compact Data Structures — A Practical Approach*. Cambridge University Press, 2016. ISBN 978-1-10-715238-0.

[136] G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: A compositionally verified compiler for a higher-order imperative language. In *Proc. Int. Conf. Functional Programming*, pages 166–178. ACM, 2015. doi: 10.1145/2784731.2784764.

[137] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi: 10.1007/3-540-45949-9.

[138] T. Nipkow, M. Eberl, and M. P. L. Haslbeck. Verified textbook algorithms: A biased survey. In *Proc. Symp. Automated Technology for Verification and Analysis*, volume 12302 of *Lecture Notes in Computer Science*, pages 25–53. Springer, 2020. doi: 10.1007/978-3-030-59152-6_2.

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

[140] G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Proc. Data Compression Conference*, pages 193–202. IEEE Computer Society, 2009. doi: 10.1109/DCC.2009.42.

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

[142] M. Norrish. *C Formalized in HOL*. PhD thesis, University of Cambridge, UK, 1999. URL https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.624131.

[143] L. Noschinski, C. Rizkallah, and K. Mehlhorn. Verification of certifying computations through AutoCorres and Simpl. In *Proc. Symp. NASA Formal Methods*, pages 46–61, 2014. doi: 10.1007/978-3-319-06200-6_4.

[144] D. S. N. Nunes, F. A. Louza, S. Gog, M. Ayala-Rincón, and G. Navarro. Grammar compression by induced suffix sorting. *ACM Journal of Experimental Algorithms*, 27:1.1:1–1.1:33, 2022. doi: 10.1145/3549992.

[145] S. Nurk, S. Koren, A. Rhie, M. Rautiainen, A. V. Bzikadze, A. Mikheenko, M. R. Vollger, N. Altemose, L. Uralsky, A. Gershman, S. Aganezov, S. J. Hoyt, M. Diekhans, G. A. Logsdon, M. Alonge, S. E. Antonarakis, M. Borchers, G. G. Bouffard, S. Y. Brooks, G. V. Caldas, N. Chen, H. Cheng, C. Chin, W. Chow, L. G. de Lima, P.p C. Dishuck, R. Durbin, T. Dvorkina, I. T. Fiddes, G. Formenti, R. S. Fulton, A. Fungtammasan, E. Garrison, P. G. S. Grady, T. A. Graves-Lindsay, I. M. Hall, N. F. Hansen, G. A. Hartley, M. Haukness, K. Howe, M. W. Hunkapiller, C. Jain, M. Jain, E. D. Jarvis, P. Kerpedjiev, M. Kirsche, M. Kolmogorov, J. Korlach, M. Kremitzki, H. Li, V. V. Maduro, T. Marschall, A. M. McCartney, J. McDaniel, D. E. Miller, J. C. Mullikin, E. W. Myers, N. D. Olson, B. Paten, P. Peluso, P. A. Pevzner, D. Porubsky, T. Potapova, E. I.

Rogaev, J. A. Rosenfeld, S. L. Salzberg, V. A. Schneider, F. J. Sedlazeck, K. Shafin, C. J. Shew, A. Shumate, Y. Sims, A. F. A. Smit, D. C. Soto, I. Sović, J. M. Storer, A. Streets, B. A. Sullivan, F. Thibaud-Nissen, J. Torrance, J. Wagner, B. P. Walenz, A. Wenger, J. M. D. Wood, C. Xiao, S. M. Yan, A. C. Young, S. Zarate, U. Surti, R. C. McCoy, M. Y. Dennis, I. A. Alexandrov, J. L. Gerton, R. J. O'Neill, W.Timp, J. M. Zook, M. C. Schatz, E. E. Eichler, K. H. Miga, and A. M. Phillippy. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022. doi: 10.1126/science.abj6987.

[146] L. O'Connor. *Type Systems for Systems Types*. PhD thesis, University of New South Wales, 2019.

[147] L. O'Connor, Z. Chen, C. Rizkallah, S. Amani, J. Lim, T. C. Murray, Y. Nagashima, T. Sewell, and G. Klein. Refinement through restraint: Bringing down the cost of verification. In *Proc. Int. Conf. Functional Programming*, pages 89–102. ACM, 2016. doi: 10.1145/2951913.2951940.

[148] L. O'Connor, Z. Chen, C. Rizkallah, V. Jackson, S. Amani, G. Klein, T. Murray, T. Sewell, and G. Keller. Cogent: Uniqueness types and certifying compilation. *Journal of Functional Programming*, 31:e25, 2021. doi: 10.1017/S095679682100023X.

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

[150] D. Okanohara and K. Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In *Proc. Symp. String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2009. doi: 10.1007/978-3-642-03784-9_9.

[151] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. Conf. Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996. doi: 10.1007/3-540-61474-5_91.

[152] D. Patterson and A. Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proceedings of the ACM on Programming Languages*, 3(ICFP): 85:1–85:29, 2019. doi: 10.1145/3341689.

[153] L. C. Paulson. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science*, 5(3):73–87, 1999. doi: 10.3217/JUCS-005-03-0073.

[154] L. C. Paulson. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *Proc. Work. Practical Aspects of Automated Reasoning*, volume 9 of *EPiC Series in Computing*, pages 1–10. EasyChair, 2010. doi: 10.29007/TNFD.

[155] L. C. Paulson. Knuth–Morris–Pratt string search. *Archive of Formal Proofs*, November 2023. ISSN 2150-914x. https://isa-afp.org/entries/KnuthMorrisPratt.html, Formal proof development.

[156] C. Pit-Claudel, P. Wang, B. Delaware, J. Gross, and A. Chlipala. Extensible extraction of efficient imperative programs with foreign functions, manually managed memory, and proofs. In *Proc. Int. Joint Conf. Automated Reasoning*, volume 12167 of *Lecture Notes in Computer Science*, pages 119–137. Springer, 2020. doi: 10.1007/978-3-030-51054-1_7.

[157] C. Pit-Claudel, J. Philipoom, D. Jamner, A. Erbsen, and A. Chlipala. Relational compilation for performance-critical applications: Extensible proof-producing translation of functional models into low-level code. In *Proc. Conf. Programming Language Design and Implementation*, pages 918–933. ACM, 2022. doi: 10.1145/3519939.3523706.

[158] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998. doi: 10.1007/BFB0054170.

[159] C. Preston, Z. Arnavut, and B. Koc. Lossless compression of medical images using Burrows-Wheeler Transformation with inversion coder. In *Proc. Conf. IEEE Engineering in Medicine and Biology Society*, pages 2956–2959. IEEE Computer Society, 2015. doi: 10.1109/EMBC.2015.7319012.

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

[161] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, Center for Research in Computing Technology, 1981.

[162] M. O. Rabin and D. S. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959. doi: 10.1147/RD.32.0114.

[163] J. Reason. *Human Error*. Cambridge University Press, 1990. ISBN 978-1-13906236-7. doi: 10.1017/CBO9781139062367.

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

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

[166] N. Robertson, D. P. Sanders, P. D. Seymour, and R. Thomas. The four-color theorem. *Journal of Combinatorial Theory, Series B*, 70(1):2–44, 1997. doi: 10.1006/JCTB.1997.1750.

[167] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9.

[168] G. Rosone and M. Sciortino. The Burrows-Wheeler Transform between data compression and combinatorics on words. In *Proc. Conf. Nature of Computation, Logic, Algorithms, Applications*, volume 7921 of *Lecture Notes in Computer Science*, pages 353–364. Springer, 2013. doi: 10.1007/978-3-642-39053-1_42.

[169] J. M. Rushby. Design and verification of secure systems. In *Proc. Symp. Operating Systems Principles*, pages 12–21. ACM, 1981. doi: 10.1145/800216.806586.

[170] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle-HOL*. PhD thesis, Technical University Munich, Germany, 2006.

[171] N. Schirmer. A sequential imperative programming language syntax, semantics, hoare logics and verification environment. *Archive of Formal Proofs*, February 2008. ISSN 2150-914x. https://isa-afp.org/entries/Simpl.html, Formal proof development.

[172] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011. ISBN 032157351X.

[173] J. Seward. bzip2 homepage. https://sourceware.org/bzip2/index.html, 1996. Accessed: 2023-09-12.

[174] T. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Proc. Conf. Programming Language Design and Implementation*, pages 471–482. ACM, 2013. doi: 10.1145/2491956.2462183.

[175] V. Shchur. A quantitative version of the Morse lemma and quasi-isometries fixing the ideal boundary. *Journal of Functional Analysis*, 264(3):815–836, 2013. doi: https://doi.org/10.1016/j.jfa.2012.11.014.

[176] K. Slind and M. Norrish. A brief overview of HOL4. In *Proc. Conf. Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi: 10.1007/978-3-540-71067-7_6.

[177] C. Sternagel. Imperative insertion sort. *Archive of Formal Proofs*, September 2014. ISSN 2150-914x. https://isa-afp.org/entries/Imperative_Insertion_Sort.html, Formal proof development.

[178] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968. doi: 10.1145/363347.363387.

[179] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *Proc. Symp. Principles of Programming Languages*, pages 97–108. ACM, 2007. doi: 10.1145/1190216.1190234.

[180] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi: 10.1007/BF01206331.

[181] A. Voronkov. The anatomy of vampire implementing bottom-up procedures with code trees. *Journal of Automated Reasoning*, 15(2):237–265, 1995. doi: 10.1007/BF00881918.

[182] P. Wadler. Linear types can change the world! In *Proc. Conf. Programming Concepts and Methods*, page 561. North-Holland, 1990.

[183] P. Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015. doi: 10.1145/2699407.

[184] P. Wang, S. Cuéllar, and A. Chlipala. Compiler verification meets cross-language linking via data abstraction. In *Proc. Conf. Object Oriented Programming Systems Language and Applications*, pages 675–690. ACM, 2014. doi: 10.1145/2660193.2660201.

[185] F. Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006. ISBN 3-540-30704-4. doi: 10.1007/11542384.

[186] R. Wilson. *Four Colors Suffice: How the Map Problem Was Solved*. Princeton University Press, 2002. ISBN 9780691115267.

[187] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, and M. Norrish. Mind the gap: A verification framework for low-level C. In *Proc. Conf. Theorem Proving in Higher Order Logics*, pages 500–515. Springer, August 2009. doi: 10.1007/978-3-642-03359-9_34.

[188] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999. ISBN 1-55860-570-3.

[189] K. Yang, C. Tian, N. Zhang, Z. Duan, and Hongwei Du. A CEGAR-based static-dynamic approach to verifying full regular properties of C programs. *IEEE Transactions on Reliability*, 70(4):1455–1467, 2021. doi: 10.1109/TR.2021.3118877.

[190] B. Zhan and M. P. L. Haslbeck. Verifying asymptotic time complexity of imperative programs in Isabelle. In *Proc. Int. Joint Conf. Automated Reasoning*, volume 10900 of *Lecture Notes in Computer Science*, pages 532–548. Springer, 2018. doi: 10.1007/978-3-319-94205-6_35.

[191] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Survey*, 38(2):6, 2006. doi: 10.1145/1132956.1132959.