Ca' Foscari University of Venice
and
Masaryk University

Doctor of Philosophy Programme
in Computer Science

Cycle XXXIII

Final Thesis

String Analysis for Software Verification

SSD: INF/01

Programme Coordinator

Prof. Agostino Cortesi

Supervisor (Ca' Foscari University)

Prof. Agostino Cortesi

Supervisor (Masaryk University)

Prof. Vashek Matyas

Graduate Student

Martina Olliaro

Matriculation Number 834397

UČO 477767

Abstract

This thesis aims to investigate string manipulation with security implications in different programming languages and to improve the state-of-the-art by applying the Abstract Interpretation theory to string analysis. Erroneous string manipulation is a challenging problem in software verification and, in fact, it is one of the major causes of program vulnerabilities that can be exploited by malicious users, leading to severe consequences for the affected systems. By string analysis we mean statically computing the set of string values that are possibly assigned to a variable. Like for other analysis issues, this is undecidable. Thus a certain degree of approximation is necessary in order to find evidence of bugs and vulnerabilities in string manipulating code. We take advantage of the Abstract Interpretation theory, i.e., a powerful mathematical theory that enables us to define and prove the soundness of approximations. The five main contributions of this thesis are:

Abstracting shape and content of strings. We introduce a new sophisticated string abstract domain for `C` language. The domain (called M-String), is parametrized on an index (bound) domain and a character domain. Picking different constituent domains, i.e., both shape information on the array structure and value information on the contained characters, allows M-String to be tailored for specific verification tasks (e.g., detection of buffer overflows), balancing precision against complexity. We describe the concrete and the abstract semantics of basic string operations and prove their soundness formally. In addition to a selection of string functions from the standard `C` library, we provide semantics for character access and update, enabling automatic lifting of arbitrary string-manipulating code into the domain. Furthermore, we provide an executable implementation of abstract operations. Using a tool that automatically lifts existing programs into the M-String domain along with an explicit-state model checker, we evaluate the accuracy of the proposed domain experimentally on real-case test programs.

Combining string domains. We combine abstract domains resulting from the reduced product between string shape abstraction and string content abstraction, in order to improve the ability to detect inconsistent states leading to program errors without a major impact with respect to efficiency. In particular, the combinations involve some string abstract domains introduced in the literature with the segmentation domain that we instantiate for string analysis.

Completeness of string domains. Completeness, in Abstract Interpretation, ensures that the analysis does not lose information with respect to the property of interest. We provide a systematic and constructive approach for generating the completion of string domains for dynamic languages, and we apply it to the refinement of existing string abstractions. Indeed, for dynamic languages, lack of string analysis completeness is a key security issue, as poorly managed string manipulation code may easily lead to significant security flaws. We also provide an effective procedure to measure the precision improvement obtained when lifting the analysis to complete domains.

Relational abstract domains for string analysis. Almost all the existing string abstract domains track information of single variables in a program (e.g., if a string contains a certain character), without inspecting their relationship with other values, causing the loss of relevant knowledge. Thus, we introduce a generic framework that allows to formalize relational string abstract domains based on ordering relationship, and we instantiate such a framework to several domains built upon different well-know string orders (e.g., substring relationships). We implemented the domain based on substring ordering, and we provide an experimental evaluation about its effectiveness on some case studies.

String manipulation in watermarking scenarios. We manipulate string values in the context of relational database watermarking. We propose a semantic-driven watermarking approach of relational textual databases, which marks multi-word textual attributes, exploiting the synonym substitution technique for text watermarking together with notions in semantic similarity analysis, and dealing with the semantic perturbations provoked by the watermark embedding. We show the effectiveness of our approach through an experimental evaluation, highlighting the resulting capacity, robustness and imperceptibility watermarking requirements. We also prove the resilience of our approach with respect to the random synonym substitution attack.

CONTENTS

1 Introduction **1**

1.1 Motivation 1

1.2 Methodology 3

1.2.1 Static Analysis 4

1.2.2 Abstract Interpretation 4

1.3 Contribution 6

1.4 Thesis Structure 7

2 Preliminaries **9**

2.1 Sets and Sequences 9

2.2 Order Relations 10

2.3 Functions 11

2.4 Fixpoints 12

2.5 Traces 13

2.6 Abstract Interpretation 14

2.6.1 Galois Connection 14

2.6.2 Soundness and Completeness 16

2.6.3 Fixpoints Approximation 17

2.6.4 Product Operators 18

3 String Analysis for C **21**

3.1 Introduction 21

3.2 FunArray 23

3.2.1 Array Concrete Representation 24
3.2.2 Array Abstract Domain Functor 25
3.3 Syntax 26
3.4 Concrete Domain and Semantics 28
3.4.1 Character Array Concrete Representation 28
3.4.2 Concrete Domain 29
3.4.3 Concrete Semantics 30
3.5 M-String 35
3.5.1 Character Array Abstract Domain Functor 35
3.5.2 Abstract Semantics 41
3.5.3 Soundness 45
3.6 Program Abstraction 48
3.6.1 Compilation-Based Approach 49
3.6.2 Syntactic Abstraction 50
3.6.3 Aggregate Domains 51
3.6.4 Semantic Abstraction 53
3.6.5 Abstract Operations 54
3.7 Instantiating M-String 54
3.7.1 Symbolic Scalar Values 55
3.7.2 Concrete Characters, Symbolic Bounds 55
3.7.3 Symbolic Characters, Symbolic Bounds 57
3.7.4 Implementation 58
3.8 Experimental Evaluation 58
3.9 Discussion 63

4 Combining String Domains **65**
4.1 Introduction 65
4.2 Syntax 68
4.3 Concrete Domain and Semantics 69
4.3.1 Concrete Domain 69
4.3.2 Concrete Semantics 69
4.4 String Abstract Domains 71

4.4.1 String Length 71
4.4.2 Character Inclusion 72
4.4.3 Prefix and Suffix 73
4.5 Segmentation Abstract Domain 74
4.5.1 String Concrete Representation 75
4.5.2 Abstract Domain 75
4.5.3 Abstract Semantics 82
4.5.4 Soundness 83
4.6 Refined String Abstract Domains 85
4.6.1 Refinement Meaning 85
4.6.2 Combining Segmentation and String Length Domains 86
4.6.3 Combining Segmentation and Character Inclusion Domains 89
4.6.4 Combining Segmentation and Prefix Domains 94
4.7 Discussion 99

5 Completeness of String Domains 101

5.1 Introduction 101
5.2 Making Abstract Interpretation Complete 102
5.2.1 Complete Shell vs Complete Core 103
5.2.2 Domain Completion Procedure 105
5.2.3 Motivating Example 106
5.3 Core Language 108
5.3.1 Syntax 108
5.3.2 Concrete Semantics 109
5.4 Making JavaScript String Abstract Domains Complete 109
5.4.1 Completing TAJS String Abstract Domain 110
5.4.2 Completing SAFE String Abstract Domain 113
5.5 Benefits of Adopting Complete String Abstractions 117
5.5.1 Precision 117
5.5.2 Qualitative Evaluation of Complete Shells 118
5.5.3 False Positives Reduction 119
5.6 Relative Precision 120

5.6.1 Abstract Domains Precision: an Overview 120
5.6.2 Measuring Precision Gained by Complete Shells 121
5.6.3 Experimental Evaluation 123
5.7 Discussion 124

6 Relational String Abstract Domains 125

6.1 Introduction 125
6.2 Core Language 127
6.2.1 Syntax 127
6.2.2 Concrete Semantics 128
6.3 A Suite of String Relational Abstract Domains 130
6.3.1 General Relational Framework 130
6.3.2 String Length Relational Abstract Domain 132
6.3.3 Char Inclusion Relational Abstract Domain 133
6.3.4 Substring Relational Abstract Domain 134
6.3.5 Extension to String Expressions 135
6.4 Experimental Evaluation 138
6.4.1 Test Cases 138
6.4.2 Evaluating a Real World Sample 140
6.5 Discussion 141

7 String Manipulation in Watermarking Scenarios 143

7.1 Introduction 144
7.2 Motivating Examples 147
7.3 Preliminaries 148
7.3.1 Semantic Similarity Theory 149
7.3.2 Text Watermarking 149
7.4 Semantic-based Watermarking Approach 151
7.4.1 Architecture of the Proposal 151
7.4.2 Watermarking Procedure 155
7.4.3 Analysis of the Watermark Capacity 157
7.4.4 Considerations for the Adversary Model 159
7.5 Experimental Results 161

7.5.1 Improvement of the Watermark Capacity 163
7.5.2 Detectability Analysis 164
7.5.3 Watermark Imperceptibility 167
7.5.4 Technique's Robustness 168
7.5.5 Scalability and Complexity 169
7.6 Discussion 171

8 Conclusion and Future Work **173**

A String Abstract Domains **175**
A.1 String Length 175
A.2 Character Inclusion 179
A.3 Prefix 182

B Relational String Abstract Domains **185**
B.1 Abstract Semantics of Len 185
B.2 Abstract Semantics of Char 186
B.3 Abstract Semantics of Sub 187
B.4 Abstract Semantics of Sub* 189
B.5 Len* Abstract Domain 192
B.6 Char* Abstract Domain 193

Bibliography **197**

LIST OF FIGURES

3.1 Interpretation-based vs Compilation-based 50
3.2 Syntactic Abstraction 50
3.3 M-String with Symbolic Bounds 56

5.1 Coalesced Sum Abstract Domain 107
5.2 μDyn Syntax 108
5.3 μDyn Semantics 109
5.4 SAFE and TAJS Domains over μDyn 110
5.5 TAJS Complete Shell 112
5.6 SAFE Concatenation Abstract Semantics 113
5.7 SAFE Absolute Complete Shell 116
5.8 μDyn Program Example 121

6.1 SECNAME Sample 126
6.2 IMP Syntax 128
6.3 CFG Generation 128
6.4 Len Lattice Operations 133
6.5 Char Lattice Operations 134
6.6 Sub Lattice Operations 135
6.7 A^* Lattice Operations 136
6.8 $\mathsf{Sub}^\star$ Lattice Operations 137
6.9 NOTCON and REP Programs 138

7.1 Embedding Process Architecture 152

7.2 Synonyms Sets 154
7.3 Evaluating Watermark Capacity 159
7.4 WM Source Samples 161
7.5 Analysis Correlation 170

LIST OF TABLES

3.1 `C` Syntax 27
3.2 M-String Measurements 59
3.3 Benchmark Abstraction 59
3.4 `PDCLib` Verification Results 61
3.5 Veriabs Overflow Benchmarks Results 61
3.6 Measurements for Automatically Generated Parsers Analysis 62
3.6 Measurements for Automatically Generated Parsers Analysis: *cont* 62

4.1 Introductory Example 68
4.2 Shortcuts of String Constants 71

5.1 Completing SAFE Domain 115
5.2 TAJS Domain vs TAJS Complete Shell Domain 124

6.1 Analysis Results of NOTCON 139
6.2 Analyses Results of REP 139
6.3 Analysis Results of SECNAME 141

7.1 AHK Approach Notation 145
7.2 Motivating Example 147
7.3 Structure of the Dataset "Amazon Fine Food Reviews" 162
7.4 Binary Capacity Values 164
7.5 Weight-based Capacity Values 164
7.6 WSD Precision During WM Detection 165

7.7 Dected WM Quality 166
7.8 WM Detectability 166
7.9 Fixed Word Rate Values 167
7.10 Similarities for WM UTM 167
7.11 Detected WMs After Pseudo-random Tuple Deletion Attacks 169
7.12 Detected WMs After Pseudo-Random Update Attacks 169
7.13 WM Embedding Required Time 170

A.1 $\overline{\mathbf{SL}}$ abstract semantics 177
A.2 $\overline{\mathbf{CI}}$ abstract semantics 181
A.3 $\overline{\mathbf{PR}}$ abstract semantics 183

Chapter 1

INTRODUCTION

The goal of this thesis is to improve the state-of-the-art of string analysis in software verification. We take into account different scenarios of string manipulation and by applying the Abstract Interpretation theory, we introduce new abstract domains for string analysis and we discuss their ability to enhance the precision when combining them with other existing strings domains. As an additional contribution we apply a semantic approach to string manipulation in the context of watermarking for relational databases.

1.1 Motivation

Programming with strings is often prone to errors. Indeed the different ways strings are treated by the programming languages and the lack of support for string manipulation often lead to run-time errors. Depending on the language we are using, strings can be handled as arrays of characters, as primitive data types or as objects. Due to their different implementation, strings suffer of manipulation errors causing vulnerabilities that result in irreversible damages [30].

For instance, in the `C` programming language, strings are not a built-in type. As a matter of fact, they are represented as null terminating arrays of characters. Due to how arrays are managed in `C`, it is common for those that manipulate strings to suffer from errors such as: unbounded string copies, off-by-one errors, null-termination errors and string truncation [154]. Some of those errors lead to buffer overflows that can cause the program to behave unpredictably, to crash or to be subject of malicious code injection.

Injection attacks target also programs written in a memory safe language. In the `Java` programming language, strings are objects corresponding to the instances of the String class, and even thought the `Java` language is memory safe, it is possible for programmers to commit logical programming errors leading to unexpected consequences. The same happens in software written in the `JavaScript` language, where strings are a primitive data type and language features like, e.g., dynamic typing, implicit type conversion, and

reflection, compromise code readability, data consistency and make the code prone to bugs or to vulnerability exploitations [12, 145].

Nevertheless, strings play a crucial role in programming. In particular, in web-based software, strings are used any time the web application needs to communicate with and to store end-users informations, as well as, in generating dynamic code.

For the reasons above string analysis techniques, i.e., a static analysis approach that, given a particular point of a program, aims to determine in that point the literals hold by a string variable, are recognised to be an effective approach to detect string manipulation errors and to prevent catastrophic events. The relevance of string analysis may be better understood by the following two security scenarios.

Buffer Overflow Vulnerability

According to the IEEE and TIOBE rankings, `C` is currently one of the top ten programming languages in use [31, 168], and many software systems of critical importance are written in `C`.

A well-known security issue that affects `C` code is the buffer overflow vulnerability. A buffer overflow consists in an access to a buffer outside its assigned bounds and it occurs because the `C` programming language does not provide any built-in protection to over-reading or to over-writing of data in any part of the memory. For example, there is no automatic check if the insertion of data into an array (the built-in buffer type) is confined within its boundaries. Static methods able to automatic detect buffer overflows in `C` programs have been widely studied in the literature (with an emphasis also on strings) [62, 63, 129, 161, 173, 175, 181]. Those methods exploit different inference techniques based, among others, on constraint-based techniques, tainted data-flow analysis, string pattern matching analysis or annotation analysis [156].

Injection

Web applications are often affected by problems that are related to how strings are (improperly) manipulated [30, 91, 95, 164], leading to severe consequences, e.g., deleting content from a database or executing malicious scripts in the end user's browser. According to the OWASP Top Ten [141], two of the most common vulnerabilities in web applications that arise from string manipulation errors are: Cross Site Scripting (XSS) and SQL injection (SQLi). Both vulnerabilities are classified as injection attacks, i.e., when untrusted inputs are supplied to a program, and are caused by inadequate string input validation and sanitization, specifically:

- "A SQL Injection attack consists of insertion (or injection) of a SQL query via the input data from the client to the application" [163].

- "A Cross Site Scripting attack consists of injection of malicious scripts into otherwise benign and trusted websites" [182].

In the last decade, the number of web applications is increasing and, consequently, the amount of new XSS and SQLi vulnerabilities is advancing despite of both the suggestions provided by secure coding practices and the available techniques of input validation and sanitization.

To this end, precise static analysis techniques can prevent string manipulation errors that lead to security attacks in web-based software. In the literature several frameworks that use static analysis to counter injection attacks have been designed [71, 94, 100, 109, 124, 174, 177]. Furthermore, different string analysis techniques have been developed to tackle the problems mentioned above [114, 133, 167, 170, 176, 178, 186, 188]. For instance, Minimade [133] proposed to use a context free grammar to approximate strings in order to validate and guarantee the security of web pages dynamically generated by a server side program. This technique has been extended by Wasserman and Su to find SQLi [176] and XSS vulnerabilities [177]. In [188], Yu et al. applied string analysis, by means of automaton abstraction, to check the correctness of sanitization operations and later to automatically repair faulty string manipulation code [184]. Detection and verification of sanitizers has been carried out also by Tateishi et al. [167], where constraints over program variables and string operations are represented with monadic second-order logic. Finally, string analysis by means of Abstract Interpretation has been used by Tripp et al. [170] to detect `JavaScript` security vulnerabilities in the client side of web applications. Others survey contributions in this field can be found in [90, 95, 116, 157].

1.2 Methodology

As mentioned above, string analysis is a static analysis technique that, given a particular program point where a string variable occurs, aims to determine the literals possibly assigned to that variable. In general this is an undecidable problem. However, approximations can be used to prove anyway the presence or the absence of software bugs and vulnerabilities in the string manipulating code.

In the recent literature, different approximation techniques for string analysis have been developed, such as [30]: automata-based [15, 34, 187, 188], abstraction-based [8, 9, 10, 11, 49], constraint-based [1, 123, 150, 152, 172], and grammar-based [133, 177]. For instance, in [15], Arceri and Mastroeni defined a new automaton-based semantics for string analysis to handle dynamic languages string features such as dynamic typing and implicit type conversion. Amadini et al., in [8], approximate strings as a dashed string, namely a sequence of concatenated blocks that specify the number of times the character they contain must appear and the number of times the latter may appear. Samirni et al.

[150] repaired HTML generation errors in PHP programs by solving a system of string constraints.

We choose to adopt the Abstract Interpretation theory [51, 56] to approach the string analysis problem as it provides a rigorous mathematical framework within which we can design sound approximations varying the degrees of abstractions.

1.2.1 Static Analysis

Static analysis aims at automatic reasoning about the behaviour of computer programs without executing them [136]. Firstly used in compiler optimization (e.g., to produce efficient code), it is now widely applied in software verification. In particular, the verification process is about providing guarantees that a computer program complies with its requirement (e.g., correctness) [101] and as a matter of fact static program analysis has been shown useful in order to find bugs [19, 27]. That is extremely important when code bugs may be exploited to perform security attacks.

It is important to note that from the Rice's theorem [148], we know that all non trivial properties about the behaviour of programs, written in a Turing complete programming language, are undecidable. This means that automated reasoning of software generally must involve *sound* approximations. By sound approximation we mean the approximation of concrete values by means of abstract values, and an approximation of operations by corresponding operations on abstract values, preserving the semantics with respect to the observed property kept by the abstraction.

In the context of software verification, a sound verification analysis must detect all the errors, possibly producing also false positive due to the approximation.

Different static analysis techniques have been developed since the 1970s, such as: data flow analysis [113], control flow analysis [7], type and effect analysis [138], Abstract Interpretation [51], model checking [22] and symbolic execution [117]. As the years go by, for some of those techniques there have been defined several approaches. For example, the data flow analysis can be equational-based or constraint-based [139] and the model checking technique can be abstract-based [35] or symbolic-based [36].

1.2.2 Abstract Interpretation

Abstract Interpretation [51, 56] has been proposed by P. Cousot and R. Cousot in the 1970s as a general theory of sound abstraction of the uncomputable concrete semantics of computer programs, which is able to cover all the aforementioned static analysis techniques. Now it is widely integrated in software verification tools and used to rigorous mathematical proofs of approximations correctness. Static analysis by Abstract Interpretation is sound by construction, but in general it is not complete, as some loss of information may occur during the abstract computation even with respect to the property of interest. Concrete values managed by the program are represented by elements belonging to algebraic structures

called abstract domains. Abstract domains are designed taking into account the properties that need to be inferred, e.g., numerical or lexical, relational or non-relational, and they can be both language-specific or general purpose. Moreover, Abstract Interpretations allows the combination of abstract domains, through suitable operators, i.e., refinement and product operators, to enhance the accuracy of the abstract computation.

Numerical Abstract Domains

A numerical abstract domain is an abstract domain that approximates sets of numerical values [126]. Different numerical abstract domains have been designed, such as: the Sign and the Parity abstract domains [50, 51] that respectively approximate integers by their sign or by their "be even or odd", the Interval abstract domain [51] that takes into account the minimum and the maximum value that an integer variable can be assigned to. The domains just mentioned do not track relationships between variables. Conversely the following ones are relational: the Pentagon abstract domain [126] used to prove the safety of array accesses in byte-code and intermediate languages, the Octagon abstract domain [135] which allows to prove program invariants and the Trapezoid Step Function abstract domain [43] which approximates continuous functions. Recently, Bautista et al. [25] defined the Numeric Path Relations abstract domain which denotes relations over structured data containing scalar values. Moreover, combinations of numerical abstract domains have been extensively used to improve the precision of the analysis and implemented in various tools [59, 64].

String Abstract Domains

A string abstract domain approximates sets of string values. In the recent literature, different abstract domains for string analysis have been defined [11, 10, 34, 49, 57, 115, 127, 185] and different tools have been designed for their usage and combination [105, 112, 122, 162]. For instance, a basic, well-known domain is the String Set domain, which simply keeps track of a set of strings and it is a specific instance of the general (bounded) set domain. There exist general-purpose abstract domains that approximate shape and/or content information of strings, such as: the Character Inclusion and the Prefix domains [49] that track the characters possibly and certainly contained in a string and its prefix, respectively. The String Length abstract domain approximates the minimum and maximum length of a string, similarly to the interval domain for numerical values. Another general-purpose string domain is the String Hash domain proposed in [127], based on a distributive hash function. Some string abstract domains focus on specific programming languages such as `JavaScript` [16, 105] and `C` [108, 134], others target specific properties and rely on different approaches like automata [188], regular expressions [142], and grammatical summaries [115].

1.3 Contribution

The main goal of this thesis is to investigate existing string abstractions, to understand the nature of string problems, to design novel string abstract domains tailored for software security and verification and to refine the existing ones. Our contribution can be summarized as follows:

- We introduce M-String, a sophisticated `C`-specific string abstract domain. It fully approximates both the content and the shape of strings by representing them as abstractions of consecutive, non-overlapping, possibly empty segments. A segment represents a sequence of characters which share the same property and is surrounded by the segment bounds that abstracts concrete array indexes. By using M-String we may detect buffer overflows, among other verification tasks.

- We instantiate the Segmentation domain [57] for string analysis and we combine it with some existing general-purpose string abstract domains, in order to improve the accuracy of the analysis, also highlighting inconsistent states which may lead to program errors.

- We show how to compute the completion of string abstract domains, in order to obtain an optimal abstract analysis, i.e., both sound and complete. In particular, we compute the complete version of two `JavaScript`-specific string abstract domains, i.e., TAJS [105] and SAFE [122], with respect to some operations of interest. Moreover, we quantify the precision gained using a complete domain with respect to its uncomplete version.

- We design a framework which allows the definition of relational string abstract domains based on string orders. We make the analysis able to capture relations between single string variables and then between string expressions and string variables. Moreover, we instantiate our framework over three well known string orders, i.e., the length inequality, char inclusion and substring relation.

- As a last contribution, not strictly related to the previous ones, we introduce a new watermarking technique for relational textual databases based on synonyms substitution which allows the preservation of the semantic consistency between the original database and the watermarked one. Our technique achieves high level of capacity and robustness against the synonyms substitution attack, among others, while not damaging its imperceptibility requirement.

Note that most of the results we achieved are equipped with an experimental evaluation.

1.4 Thesis Structure

Chapter 2 introduces the technical notation that will be used in the following chapters.

Chapter 3 introduces the definition of the M-String abstract domain for the analysis of `C` programs handling strings. M-String detects the presence of common strings management errors that may lead to undefined behaviour, e.g., buffer overflows. The contribution of this chapter has been published in [47, 48, 121].

Chapter 4 presents the Segmentation string abstract domain, an instantiation of the FunArray domain functor by Cousot et al. [57], for the analysis of arrays content, to string analysis. We discuss the limitations of basic existing string domains and we combines them with the Segmentation in order to improve the accuracy of the analysis result. Combinations are carried out by exploiting the notions of reduced product [56] and Granger methodology [87]. Moreover, we study inconsistencies between abstract states as they may lead to program errors. The contribution of this chapter is still submitted for publication.

Chapter 5 defines a way to proceed in order to compute the completion [80] of string abstract domains and applies the completion procedure to two incomplete `JavaScript` string abstract domains to enhance their precision. Namely, we complete TAJS [105] and SAFE [122] abstract domains with respect to two operations of interest. Moreover, we provide a measure of the analysis precision increment gained using those complete domains with respect to the original ones. The contribution of this chapter has been published in [18].

Chapter 6 defines a general framework to design relational string abstract domains based on orders between string elements. The framework we present is firstly defined to track relations between single string variables and then enhanced to capture relations between string variables and string expressions. We instantiate it to a suit of string order relations of interest, i.e., length inequality, char inclusion and substring relation. In particular, we evaluate the latter substring relational abstract domain comparing it with state-of-the-art general purpose string abstract domains. The contribution of this chapter is still submitted for publication.

Chapter 7 presents a new watermarking technique, to protect ownership of textual relational databases, which takes care of semantic consistency between the original database and the watermarked one. We design it carefully balancing robustness and imperceptibility of the watermark, and with the aim of increasing both its capacity and resilience to attacks, e.g., synonyms substitution attack. Moreover, we formalize a new approach to measure watermark capacity and we prove the effectiveness of our technique through an experimental evaluation. The contribution of this chapter has been published in [85, 84].

Chapter 8 concludes and presents perspectives for future work. Appendix A recalls formal details of existing string abstract domains.

Chapter 2

PRELIMINARIES

In this chapter we introduce the mathematical background and the theoretical results on Abstract Interpretation theory that will be used throughout the thesis. In particular, we recall some basic notion on set and order theory, functions, definitions on fixpoints and traces, and the basics of Abstract Interpretation, i.e., the framework we use to conduct string analysis.

Chapter Structure

Section 2.1 presents some notation on sets and sequences. Section 2.2 and Section 2.3 recall some definitions on order theory and functions, respectively. Section 2.4 and Section 2.5 provide notions on fixpoints and traces, respectively. Section 2.6 introduces results on Abstract Interpretation.

2.1 Sets and Sequences

Sets

A set is a, possibly finite, collection of distinct elements. Let A and B be sets. $x \in A$ denotes the set membership, i.e., it says that a is an element of A. $\{x \in A \mid \phi(x)\}$ is the set of all elements of A which satisfy the formula ϕ. $A \subseteq B$ denotes the inclusion relation, meaning that all the elements of A are also elements of B or, in other words, that A is a subset of B. $\mathcal{P}(A)$ is the set of all subsets of A, namely the powerset of A. The difference between two sets is denoted by $A \setminus B$ representing the set of all elements in A but not in B, i.e., $\{x \in A \mid x \notin B\}$. The intersection between two sets is denoted by $A \cap B$ and it represents the set of elements that belong to both A and B, i.e., $\{x \mid x \in A \wedge x \in B\}$. The union between two sets is written $A \cup B$ and it denotes the set of elements that belong to either A and B, i.e., $\{x \mid x \in A \vee x \in B\}$. $A \times B$ is the Cartesian product between A and B, i.e., $\{(x, y) \mid x \in A \wedge y \in B\}$. We denote by A^n the n-ary Cartesian product of a set A, with $n \geq 2$ and by $\mid A \mid$ the cardinality of A. $\emptyset$ is the emptyset. We denote by $\exists^n$ the existential quantifier with cardinality n, which is read as "there exist exactly n objects".

As usual in mathematics, $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ and $\mathbb{R}$ denote the set of all natural, integer, rational, and real numbers, respectively.

Sequences

Let Σ be a finite set of characters, i.e., an alphabet. We denote by Σ^* the set of all possible finite sequences of characters. Let $\sigma \in \Sigma^*$, we denote by $\mid \sigma \mid$ the length of σ, by $\mathrm{char}(\sigma)$ the set of characters that occurs in the string σ. The empty string is denoted by ε. A string σ may be also denoted by $\sigma_1 \ldots \sigma_n$ in order to refer to its characters. $\sigma + \sigma'$ denotes the string concatenation, and $\sigma[x/i]$ is σ where the character occurring in position i has been substituted with the character x. Given a string σ, we denote by $\mathsf{char}(\sigma)$ the set of of the characters occurring in σ.

2.2 Order Relations

A binary relation $\sqsubseteq_L$ over a set L is a subset of $L \times L$, i.e., $\sqsubseteq_L \subseteq L \times L$. A binary relation $\sqsubseteq_L$ over a set L is a **pre-order** if and only if it is reflexive, i.e., $\forall x \in L : x \sqsubseteq_L x$, and transitive, i.e., $\forall x_1, x_2, x_3 \in L : x_1 \sqsubseteq_L x_2 \wedge x_2 \sqsubseteq_L x_3 \Rightarrow x_1 \sqsubseteq_L x_3$. A binary relation $\sqsubseteq_L$ over a set L is a **partial order** if and only if it is a pre-order, and anti-symmetric, i.e., $\forall x_1, x_2 \in L : x_1 \sqsubseteq_L x_2 \wedge x_2 \sqsubseteq_L x_1 \Rightarrow x_1 = x_2$. A binary relation $\sqsubseteq_L$ over a set L is a **total order** if and only if it is a partial order, and $\forall x_1, x_2 \in L : x_1 \sqsubseteq_L x_2 \vee x_2 \sqsubseteq_L x_1$.

A set L equipped with an order relation is denoted by $(L, \sqsubseteq_L)$. A partially ordered set is also called poset. Given a poset $(L, \sqsubseteq_L)$, the infimum $\bot_L$, when it exists, is the element in L smaller than any other element of L w.r.t. $\sqsubseteq_L$. The supremum $\top_L$, when it exists, is the element in L greater than any other element of L w.r.t. $\sqsubseteq_L$. When both $\bot_L$ and $\top_L$ exist then $(L, \sqsubseteq_L)$ is said to be bounded. Given a poset $(L, \sqsubseteq_L)$ and a $X \subseteq L$, an upper bound of X is an element $y \in L$ such that $x \sqsubseteq_L y$, $\forall x \in X$. An upper bound y for X is the least upper bound (lub) of X if and only if for every other upper bound y' of X it holds that $y \sqsubseteq_L y'$. The least upper bound, when it exists, it is unique. Dually, a lower bound of X is an element $y \in L$ such that $y \sqsubseteq_L x$, $\forall x \in X$. A lower bound y of X is the greatest lower bound (glb) of X if and only if for every other lower bound y' of X it holds that $y' \sqsubseteq_L y$. Given a poset $\langle L, \sqsubseteq_L \rangle$ and $X \subseteq L$, we denote by $\mathsf{max}^{\sqsubseteq_L}(X) = \{x \in X \mid \forall y \in X.\, x \sqsubseteq_L y \Rightarrow x = y\}$ the set of the maximal elements of X in L w.r.t. $\sqsubseteq_L$.

A subset X of a poset $(L, \sqsubseteq_L)$ is a chain if it is totally ordered w.r.t. $\sqsubseteq_L$, i.e., $\forall x_1, x_2 \in X : x_1 \sqsubseteq_L x_2 \vee x_2 \sqsubseteq_L x_1$. X is an ascending chain if it is ordered and each element is greater or equal to the previous one w.r.t. $\sqsubseteq_L$. X is an descending chain if it is ordered and each element is smaller or equal to the previous one w.r.t. $\sqsubseteq_L$. A poset $(L, \sqsubseteq_L)$ satisfies the ascending chain condition (ACC) if every ascending chain $x_1 \sqsubseteq_L x_2 \sqsubseteq_L \ldots$ of elements in L is eventually stationary, i.e., $\exists n \in \mathbb{N} : \forall m > n : x_m = x_n$. Similarly, $(L, \sqsubseteq_L)$ satisfies

the descending chain condition (DCC) if there is no infinite descending chain. Every finite poset satisfies the ACC and DCC.

Note that a poset can be graphically represented by the so-called Hasse diagram. Given a poset $(L, \sqsubseteq_L)$, its corresponding Hasse diagram has a node for each element of L and an edge links two elements $x_1, x_2 \in L$ if $x_1 \sqsubseteq_L x_2 \wedge \nexists y \in L : x_1 \sqsubseteq_L y \wedge y \sqsubseteq_L x_2$.

A poset $(L, \sqsubseteq_L)$ is a **directed set** if $\forall X \subseteq L$ such that $X \neq \emptyset$ and X is finite, then X have a least upper bound in L. Dually, a poset $(L, \sqsubseteq_L)$ is a **co-directed set** if $\forall X \subseteq L$ such that $X \neq \emptyset$ and X is finite, then X have a greatest lower bound in L. A poset $(L, \sqsubseteq_L)$ is a **complete partial order** (cpo) on directed sets if it has an infimum and for each directed set $D \subseteq L$.

Lattices

A poset $(L, \sqsubseteq_L)$ is a **join semi lattice** if for each $x_1, x_2 \in L$ there exists a least upper bound $x_1 \sqcup_L x_2$ and it belongs to L. A poset $(L, \sqsubseteq_L)$ is a **meet semi lattice** if for each $x_1, x_2 \in L$ there exists a greatest lower bound $x_1 \sqcap_L x_2$ and it belongs to L. A poset $(L, \sqsubseteq_L)$ is a **lattice** if it has an infimum, a supremum and, for each $x_1, x_2 \in L$, both the greatest lower bound $x_1 \sqcap_L x_2$ and the least upper bound $x_1 \sqcup_L x_2$ exist and belong to L. A lattice is denoted by $(L, \sqsubseteq_L, \bot_L, \top_L, \sqcap_L, \sqcup_L)$. A **complete lattice** is a lattice where each subset of L has both a least upper bound and a greatest lower bound, i.e., $\forall X \subseteq L : \bigsqcap_L X, \bigsqcup_L X \in L$. Any finite lattice is complete. A complete lattice is always a cpo.

Let L be a complete lattice. $X \subseteq L$ is a Moore family of L is $X = \mathcal{M}(X) = \{\bigsqcap_L S \mid S \subseteq X\}$, where $\bigsqcap_L \emptyset = \top_L \in \mathcal{M}(X)$. We denote by $\mathcal{M}(X)$ the Moore closure of $X \subseteq L$, that is the smallest subset of L which contains X and it is a Moore family of L.

2.3 Functions

A function is a mathematical object that relates two sets by associating each element of the first set (domain) to an element of the second set (codomain). Let f be a function with domain A and codomain B. We denote by $f : A \to B$ a function from elements of A to elements of B. Given $f : A \to B$ and $g : B \to C$ we denote with $g \circ f : A \to C$ their composition, i.e., $g \circ f = \lambda x.g(f(x))$. When f is a function of arity n, i.e., $f : A^n \to B$, with $p \in A^n$ and $i \in [0, n)$, $f_p^i = \lambda x.f(p[x/i]) : A \to B$ is the same function where all the parameters but the i-th are fixed by p.

Given two posets $(A, \sqsubseteq_A, \sqcap_A, \sqcup_A)$ and $(B, \sqsubseteq_B, \sqcap_B, \sqcup_B)$, a function $f : A \to B$ is:

- monotone if $\forall x_1, x_2 \in A : x_1 \sqsubseteq_A x_2 \Rightarrow f(x_1) \sqsubseteq_B f(x_2)$
- continuous if $\forall X \subseteq A : X$ is a chain $\wedge$ $f(\bigsqcup_A X) = \bigsqcup_B \{f(x) \mid x \in X\}$
- join preserving if $\forall x_1, x_2 \in A : f(x_1 \sqcup_A x_2) = f(x_1) \sqcup_B f(x_2)$

- complete join preserving if $\forall X \subseteq A : \bigsqcup_A X$ exists $\wedge$ $f(\bigsqcup_A X) = \bigsqcup_B f(X)$
- meet preserving if $\forall x_1, x_2 \in A : f(x_1 \sqcap_A x_2) = f(x_1) \sqcap_B f(x_2)$
- complete meet preserving if $\forall X \subseteq A : \bigsqcap_A X$ exists $\wedge$ $f(\bigsqcap_A X) = \bigsqcap_B f(X)$

The notion of function can be generalized by the one of partial function which allows a function to map some elements of its domain to an element of its codomain. Formally, a partial function $f : A \to B$ is a function $f : X \to B$ for some $X \subseteq A$.

An upper closure operator on a poset $(A, \sqsubseteq_A)$ is an operator $\rho : A \to A$ which is monotone, idempotent (i.e., $\forall x \in A : \rho(\rho(x)) = \rho(x)$), and extensive (i.e., $\forall x \in A : x \sqsubseteq_A \rho(x)$). The set of all closure operators on a poset A is denoted by $uco(A)$.

2.4 Fixpoints

Given a poset $(L, \sqsubseteq_L)$ and an operator $f : L \to L$, a fixpoint of f is an element $x \in L$ such that $f(x) = x$. The set of all fixpoints of f is $\mathsf{Fix}(f) = \{x \in L \mid f(x) = x\}$. Moreover, a pre-fixpoint of f is an element $x \in L$ such that $x \sqsubseteq_L f(x)$ and a post-fixpoint of f is an element $x \in L$ such that $f(x) \sqsubseteq_L x$. It follows that the sets of all pre- and post-fixpoints of f are $\mathsf{preFix}(f) = \{x \in L \mid x \sqsubseteq_L f(x)\}$ and $\mathsf{postFix}(f) = \{x \in L \mid f(x) \sqsubseteq_L x\}$. A fixpoint $x \in \mathsf{Fix}(f)$ is a least fixpoint of f if $\forall y \in \mathsf{Fix} : x \sqsubseteq_L y$. A fixpoint $x \in \mathsf{Fix}(f)$ is a greatest fixpoint of f if $\forall y \in \mathsf{Fix} : y \sqsubseteq_L x$. When the least and greatest fixpoints of f exist they are unique and they are denoted by $\mathsf{lfp}_x^{\sqsubseteq_L}(f)$ and $\mathsf{gfp}_x^{\sqsubseteq_L}(f)$, respectively. Precisely, $\mathsf{lfp}_x^{\sqsubseteq_L}(f)$ denotes the least fixpoint of f greater than x with respect to the order $\sqsubseteq_L$, and $\mathsf{gfp}_x^{\sqsubseteq_L}(f)$ denoter the greatest fixpoint of f smaller than x with respect to the order $\sqsubseteq_L$.

The existence of the least and greatest fixpoints of a monotonic operator defined over a complete lattice is guaranteed by the following theorems.

Theorem 2.1 (Tarski's Theorem [74])**.** Let $(L, \sqsubseteq_L, \bot_L, \top_L, \sqcap_L, \sqcup_L)$ be a complete lattice and let $f : L \to L$ be a monotonic operator on this lattice. The set of fixpoints is a non-empty complete lattice, and:

$$\mathsf{lfp}_{\bot}^{\sqsubseteq_L}(f) = \sqcap_L\{x \in L \mid f(x) \sqsubseteq_L x\}$$

$$\mathsf{gfp}_{\bot}^{\sqsubseteq_L}(f) = \sqcup_L\{x \in L \mid x \sqsubseteq_L f(x)\}$$

□

Theorem 2.2 (Constructive version of Tarski's Theorem [55])**.** Let $(L, \sqsubseteq_L, \bot_L, \top_L, \sqcap_L, \sqcup_L)$ be a complete lattice and let $f : L \to L$ be a monotonic operator on this lattice. Define the following sequence:

$$f^0 = \bot_L$$

$$f^\delta = f(f^{\delta-1}) \qquad \text{for every successor ordinal } \delta$$

$$f^\delta = \bigsqcup_{\alpha<\delta}{}_L f^\alpha \qquad \text{for every limit ordinal } \delta$$

Then the ascending chain $\{f^i \mid 0 \leq i \leq \delta\}$, where δ is an ordinal, is ultimately stationary for some $\rho \in \mathbb{N}$ that is $f^\rho = \mathsf{lfp}_\bot^{\sqsubseteq_L}(f)$.

□

2.5 Traces

Given a set X, a trace $\tau : \mathbb{N} \to X$ is a partial function such that:

$$\forall i \in \mathbb{N} : i \notin \mathsf{dom}(\tau) \Rightarrow \forall j > i : j \notin \mathsf{dom}(\tau)$$

where $\mathsf{dom}(\tau)$ denotes the domain of a trace τ. It follows that the domain of all non-empty traces is a segment of $\mathbb{N}$. The empty trace is denoted by ε_τ, i.e., the trace τ such that $\mathsf{dom}(\tau) = \emptyset$. Let be X a generic set of elements, we denote by X^+ the set of all finite traces composed by elements in X. Let $\mathsf{len}_{tr} : X^+ \to \mathbb{N}$ be the function that, given a trace, returns its length, then: $\mathsf{len}_{tr}(\tau) = i + 1$ such that $i \in \mathsf{dom}(\tau) \wedge i + 1 \notin \mathsf{dom}(\tau)$. Note that if $\tau = \varepsilon_\tau$ then $\mathsf{len}(\tau) = 0$. A trace can be represented as a sequence of states, i.e., $s_0 \to s_1 \to \ldots$. $X^+_{\xrightarrow{T}}$ denotes the set of traces in X^+ ending with a final state with respect to the transition $\xrightarrow{T}$, i.e., $X^+_{\xrightarrow{T}} = \{s_0 \to \cdots \to s_i \mid s_0 \to \cdots \to s_i \in X^+ \wedge \nexists s_j \in X \text{ such that } s_i \xrightarrow{T} s_j\}$. Given a set of initial elements X_0, a set of states S, and a transition relations $\xrightarrow{T} \subseteq \mathsf{S} \times \mathsf{S}$, the partial trace semantics [56] builds up all the traces that can be obtained by starting from traces containing only a single element from X_0 and then iteratively applying the transition relation until a fixpoint is reached.

Definition 2.1 (Partial trace semantics [56])**.** Let S be a set of states, $X_0 \subseteq \mathsf{S}$ be a set of initial elements, and $\xrightarrow{T} \subseteq \mathsf{S} \times \mathsf{S}$ a transition relation. Let $f : \mathcal{P}(\mathsf{S}) \to (\mathsf{S}^+ \to \mathsf{S}^+)$ be the function defined as:

$$F(X_0) = \lambda T.\{0 \to s_0 \mid s_0 \in X_0\} \cup \\ \{s_0 \to \cdots \to s_{i-1} \to s_i \mid s_0 \to \cdots \to s_{i-1} \in T \wedge s_{i-1} \xrightarrow{T} s_i\}$$

The partial trace semantics is defined as:

$$\mathbb{PT}[\![X_0]\!] = \mathsf{lfp}_\emptyset^{\sqsubseteq}(F)$$

△

2.6 Abstract Interpretation

Abstract Interpretation [51, 56] is a theoretical framework for sound reasoning about program semantic properties of interest. Precisely it formalizes the (concrete) semantics of a program as a fixpoint of all its execution traces, and it allows to define and prove the soundness of (possible computable) semantics at different degrees of abstraction.

2.6.1 Galois Connection

Formally, the concrete and abstract semantics are defined on a concrete $\mathbf{D}$ and abstract $\overline{\mathbf{D}}$ sets, respectively. The concrete and the abstract domains are modelled as posets, i.e., $(\mathbf{D}, \sqsubseteq_{\mathbf{D}})$ and $(\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$, respectively. The concrete and the abstract domains are related by a pair of monotone functions: the concretization $\gamma_{\overline{\mathbf{D}}} : \overline{\mathbf{D}} \to \mathbf{D}$ and the abstraction $\alpha_{\overline{\mathbf{D}}} : \mathbf{D} \to \overline{\mathbf{D}}$ functions.

Definition 2.2 (Galois Connection [51])**.** Let $(\mathbf{D}, \sqsubseteq_{\mathbf{D}})$ and $(\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$ be two partial orders. Two functions $\alpha_{\overline{\mathbf{D}}} : \mathbf{D} \to \overline{\mathbf{D}}$ and $\gamma_{\overline{\mathbf{D}}} : \overline{\mathbf{D}} \to \mathbf{D}$ form a Galois Connection if and only if

$$\forall \mathbf{d} \in \mathbf{D} : \exists \overline{\mathbf{d}} \in \overline{\mathbf{D}} : \alpha_{\overline{\mathbf{D}}}(\mathbf{d}) \sqsubseteq_{\overline{\mathbf{D}}} \overline{\mathbf{d}} \Rightarrow \mathbf{d} \sqsubseteq_{\mathbf{D}} \gamma_{\overline{\mathbf{D}}}(\overline{\mathbf{d}})$$

We denote this fact by writing

$$(\mathbf{D}, \sqsubseteq_{\mathbf{D}}) \underset{\alpha_{\overline{\mathbf{D}}}}{\overset{\gamma_{\overline{\mathbf{D}}}}{\leftrightarrows}} (\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$$

△

Theorem 2.3 ([51])**.** Let $(\mathbf{D}, \sqsubseteq_{\mathbf{D}})$ and $(\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$ be two partial orders and let $\alpha_{\overline{\mathbf{D}}} : \mathbf{D} \to \overline{\mathbf{D}}$ and $\gamma_{\overline{\mathbf{D}}} : \overline{\mathbf{D}} \to \mathbf{D}$ be two maps such that:

- $\alpha_{\overline{\mathbf{D}}}$ and $\gamma_{\overline{\mathbf{D}}}$ are monotone
- $\alpha_{\overline{\mathbf{D}}} \circ \gamma_{\overline{\mathbf{D}}}$ is reductive, i.e., $\forall \overline{\mathbf{d}} \in \overline{\mathbf{D}} : \alpha_{\overline{\mathbf{D}}} \circ \gamma_{\overline{\mathbf{D}}}(\overline{\mathbf{d}}) \sqsubseteq_{\overline{\mathbf{D}}} \overline{\mathbf{d}}$
- $\alpha_{\overline{\mathbf{D}}} \circ \gamma_{\overline{\mathbf{D}}}$ is extensive, i.e., $\forall \mathbf{d} \in \mathbf{D} : \mathbf{d} \sqsubseteq_{\mathbf{D}} \gamma_{\overline{\mathbf{D}}} \circ \alpha_{\overline{\mathbf{D}}}(\mathbf{d})$

Then, it holds that $(\mathbf{D}, \sqsubseteq_{\mathbf{D}}) \underset{\alpha_{\overline{\mathbf{D}}}}{\overset{\gamma_{\overline{\mathbf{D}}}}{\leftrightarrows}} (\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$

□

A Galois Connection can be induced by an abstraction function that is a complete lub ($\sqcup_{\overline{\mathbf{D}}}$) preserving map or by a concretization function that is a complete glb ($\sqcap_{\mathbf{D}}$) preserving map, as proved by Proposition 7 of [52].

Theorem 2.4 (Galois Connection induced by lub preserving maps)**.** Let $\alpha_{\overline{\mathbf{D}}} : \mathbf{D} \to \overline{\mathbf{D}}$ be a complete join preserving maps between posets $(\mathbf{D}, \sqsubseteq_{\mathbf{D}})$ and $(\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$. Define:

$$\gamma_{\overline{\mathbf{D}}} = \lambda \overline{\mathbf{d}}. \bigsqcup_{\mathbf{D}} \{\mathbf{d} \mid \alpha_{\overline{\mathbf{D}}}(\mathbf{d}) \sqsubseteq_{\overline{\mathbf{D}}} \overline{\mathbf{d}}\}$$

If $\gamma_{\overline{\mathbf{D}}}$ is well define, then:

$$(\mathbf{D}, \sqsubseteq_{\mathbf{D}}) \underset{\alpha_{\overline{\mathbf{D}}}}{\overset{\gamma_{\overline{\mathbf{D}}}}{\leftrightarrows}} (\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$$

□

Theorem 2.5 (Galois Connection induced by glb preserving maps)**.** Let $\gamma_{\overline{\mathbf{D}}} : \overline{\mathbf{D}} \to \mathbf{D}$ be a complete join preserving maps between posets $(\mathbf{D}, \sqsubseteq_{\mathbf{D}})$ and $(\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$. Define:

$$\alpha_{\overline{\mathbf{D}}} = \lambda \mathbf{d}. \bigsqcap_{\overline{\mathbf{D}}} \{\overline{\mathbf{d}} \mid \mathbf{y} \sqsubseteq_{\mathbf{D}} \gamma_{\overline{\mathbf{D}}}(\overline{\mathbf{d}})\}$$

If $\alpha_{\overline{\mathbf{D}}}$ is well define, then:

$$(\mathbf{D}, \sqsubseteq_{\mathbf{D}}) \underset{\alpha_{\overline{\mathbf{D}}}}{\overset{\gamma_{\overline{\mathbf{D}}}}{\leftrightarrows}} (\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$$

□

Galois Connections are compositional, i.e., the composition of two Galois Connections is still a Galois Connection.

Theorem 2.6 (Composition of Galois Connections)**.** Suppose that:

- $(\mathbf{D}, \sqsubseteq_{\mathbf{D}}) \underset{\alpha_{\overline{\mathbf{D}}}}{\overset{\gamma_{\overline{\mathbf{D}}}}{\leftrightarrows}} (\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$ and
- $(\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}}) \underset{\alpha_{\overline{\mathbf{D}}'}}{\overset{\gamma_{\overline{\mathbf{D}}'}}{\leftrightarrows}} (\overline{\mathbf{D}}', \sqsubseteq_{\overline{\mathbf{D}}'})$

Then,

$$(\mathbf{D}, \sqsubseteq_{\mathbf{D}}) \underset{\alpha_{\overline{\mathbf{D}}} \circ \alpha_{\overline{\mathbf{D}}'}}{\overset{\gamma_{\overline{\mathbf{D}}} \circ \gamma_{\overline{\mathbf{D}}'}}{\leftrightarrows}} (\overline{\mathbf{D}}', \sqsubseteq_{\overline{\mathbf{D}}'})$$

□

Note that, as it is possible to define Galois Connections also in terms of upper closure operators, the framework of Abstract Interpretation can be equivalently formalized either as Galois Connections or closure operators on a given concrete domain, which is a complete lattice $\mathbf{D}$ [56]. In particular, given $(\mathbf{D}, \sqsubseteq_{\mathbf{D}}) \underset{\alpha_{\overline{\mathbf{D}}}}{\overset{\gamma_{\overline{\mathbf{D}}}}{\leftrightarrows}} (\overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}})$, then $\gamma_{\overline{\mathbf{D}}} \circ \alpha_{\overline{\mathbf{D}}} \in uco(\mathbf{D})$. In particular, an upper closure operator (over $\mathbf{D}$) uniquely identifies a Galois Connection, and viceversa.

Lattice of Abstract Domains

If $\mathbf{D}$ is a complete lattice, then $(uco(\mathbf{D}), \sqsubseteq, \lambda x.x, \lambda x.\top, \sqcap, \sqcup)$ forms a complete lattice [51], which is the set of all possible abstractions of $\mathbf{D}$, where $\lambda x.x$ is the bottom element and $\lambda x.\top$ the top element. Let $\rho, \eta \in uco(\mathbf{D})$, $\{\rho_i\}_{i \in I} \subseteq uco(\mathbf{D})$. We say that ρ is more precise than η, i.e., $\rho \sqsubseteq \eta$ if and only if $\forall \mathbf{d} \in \mathbf{D} : \rho(\mathbf{d}) \sqsubseteq \eta(\mathbf{d})$. Let $\mathbf{d} \in \mathbf{D}$, the greatest lower bound is $(\sqcap_{i \in I} \rho_i)(\mathbf{d}) = \sqcap_{i \in I} \rho_i(\mathbf{d})$ and the least upper bound is $(\sqcup_{i \in I} \rho_i)(\mathbf{d}) = \mathbf{d}$ if and only if $\forall i \in I : \rho_i(\mathbf{d}) = \mathbf{d}$.

2.6.2 Soundness and Completeness

Two fundamental notions in Abstract Interpretation are those of soundness and completeness. A requirement of Abstract Interpretation is that the abstract semantics must be sound with respect to the concrete one, i.e., the properties verified by the abstraction process must be valid with respect to the concrete semantics. On the other hand, Abstract Interpretation accepts incompleteness of the abstract results, i.e., the abstract results can be less precise that the concrete one.

Soundness

To accomplish soundness, the concretization of the result of the abstract computation must over-approximate the result of the concrete semantics.

Definition 2.3 (Soundness)**.** Let $\mathbf{D}$ and $\overline{\mathbf{D}}$ form a Galois Connection. Moreover, let $\mathbb{S} : \mathbf{D} \to \mathbf{D}$ and $\mathbb{S}_{\overline{\mathbf{D}}} : \overline{\mathbf{D}} \to \overline{\mathbf{D}}$ be the concrete and the abstract semantics, respectively. The abstract semantics is sound if and only if $\forall \mathsf{p} \in \mathsf{preFix}(\mathbb{S}_{\overline{\mathbf{D}}}) \subseteq \overline{\mathbf{D}}$, we havee that:

$$\gamma_{\overline{\mathbf{D}}} \circ \mathbb{S}_{\overline{\mathbf{D}}}[\![\mathsf{p}]\!] \sqsupseteq_{\mathbf{D}} \mathbb{S}[\![\gamma_{\overline{\mathbf{D}}}(\mathsf{p})]\!]$$

△

Definition 2.3 presents soundness based on the computational process of the concrete domain, but we could define it also in terms of the computational process of the abstract domain.

Among all the sound abstract functions $\mathbb{S}_{\overline{\mathbf{D}}}$, we aim at the best one, namely the one that loses less information approximating $\mathbb{S}$ (cf. Definition 2.4).

Definition 2.4 (Best correct approximation)**.** Given $\mathbf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathbf{D}}$ and a concrete function $\mathbb{S} : \mathbf{D} \to \mathbf{D}$, the function $\alpha_{\overline{\mathbf{D}}} \circ \mathbb{S} \circ \gamma_{\overline{\mathbf{D}}} : \overline{\mathbf{D}} \to \overline{\mathbf{D}}$ is the best correct approximation of $\mathbb{S}$.

△

Note that there exist different ways of proving soundness of an abstract semantics [52]. Moreover, it is still acceptable if a Galois Connection between the concrete and the abstract domain cannot be induced, i.e., if does not a exist a minimal element abstracting a concrete property of interest, as we can always choose an arbitrary abstract element among all the available abstraction of a given concrete element [52] or weaken the relation between the concrete and the abstract semantics without affecting the soundness [54].

Completeness

In Abstract Interpretation, there exist two notions of completeness: backward and forward. Backward completeness property focuses on complete abstractions of the inputs, while forward completeness [75, 76, 77] focuses on complete abstractions of the outputs, both with respect to an operation of interest.

Definition 2.5 (Backward completeness)**.** Given $\mathbf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathbf{D}}$, a concrete function $\mathbb{S} : \mathbf{D} \to \mathbf{D}$ and an abstract function $\mathbb{S}_{\overline{\mathbf{D}}} : \overline{\mathbf{D}} \to \overline{\mathbf{D}}$, $\mathbb{S}_{\overline{\mathbf{D}}}$ is backward complete with respect to $\mathbb{S}$ if:

$$\alpha_{\overline{\mathbf{D}}} \circ \mathbb{S} = \mathbb{S}_{\overline{\mathbf{D}}} \circ \alpha_{\overline{\mathbf{D}}}$$

△

Definition 2.6 (Forward completeness)**.** Given $\mathbf{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathbf{D}}$, a concrete function $\mathbb{S} : \mathbf{D} \to \mathbf{D}$ and an abstract function $\mathbb{S}_{\overline{\mathbf{D}}} : \overline{\mathbf{D}} \to \overline{\mathbf{D}}$, $\mathbb{S}_{\overline{\mathbf{D}}}$ is forward complete with respect to $\mathbb{S}$ if:

$$\mathbb{S} \circ \gamma_{\overline{\mathbf{D}}} = \gamma_{\overline{\mathbf{D}}} \circ \mathbb{S}_{\overline{\mathbf{D}}}$$

△

Having backward complete abstract functions means that we have the guarantee that no loss of information arises during the input abstraction process, with respect to an operation of interest. Conversely, having forward complete abstract functions means that we have thee guarantee that no loss of information arises during the output abstraction process, with respect to an operation of interest. While forward completeness is less known, backward completeness is the one recognized ad standard notion of completeness in Abstract Interpretation, and in this thesis, we will focus on it.

2.6.3 Fixpoints Approximation

As mentioned above, the concrete and the abstract semantics are defined as the fixpoint computation of monotonic functions.

In the case in which the abstract domain respects the ACC, then the abstract computation terminates in a finite time. Otherwise a widening operator [46, 51] must be defined to force the convergence of the analysis, still maintaining the soundness.

Definition 2.7 (Widening)**.** Let $(\mathrm{L}, \sqsubseteq_{\mathrm{L}})$be a poset. A widening operator $\nabla_{\mathrm{L}} : \mathrm{L} \times \mathrm{L} \to \mathrm{L}$ satisfies the following conditions:

- $\forall \mathrm{x}, \mathrm{y} \in \mathrm{L} : \mathrm{x} \sqsubseteq_{\mathrm{L}} (\mathrm{x} \nabla_{\mathrm{L}} \mathrm{y}) \wedge \mathrm{y} \sqsubseteq_{\mathrm{L}} (\mathrm{x} \nabla_{\mathrm{L}} \mathrm{y})$
- for every increasing chain $\mathrm{x}_0 \sqsubseteq_{\mathrm{L}} \mathrm{x}_1 \sqsubseteq_{\mathrm{L}} \ldots$, the chain defined as

$$\mathrm{w}_0 = \mathrm{x}_0$$
$$\mathrm{w}_{\mathrm{n}+1} = \mathrm{w}_{\mathrm{n}} \nabla_{\mathrm{L}} \mathrm{x}_{\mathrm{n}-1}$$

  is not strictly increasing.

△

The widening often lead to a dramatic loss of information. Thus, a narrowing operator is usually defined to improve the accuracy of the widening result.

Definition 2.8 (Narrowing)**.** Let $(L, \sqsubseteq_L)$be a poset. A widening operator $\Delta_L : L \times L \to L$ satisfies the following conditions:

- $\forall x, y \in L : x \sqsubseteq_L (x \Delta_L y) \sqsubseteq_L y$
- for every decreasing chain $x_0 \sqsupseteq_L x_1 \sqsupseteq_L \ldots$, the chain defined as

$$w_0 = x_0$$
$$w_{n+1} = w_n \Delta_L x_{n-1}$$

  is not strictly decreasing.

△

2.6.4 Product Operators

Abstract Interpretation allows the combination, through *enhancing* operators, of different abstract domains within the same analysis, without affecting its soundness. Combinations aims to improve the accuracy of the analysis, also at the cost of increasing its complexity.

Enhancing operators are those who refine the information tracked by abstract domains. They can be formalized as lower closure operators on the set of all abstract interpretations of a concrete domain **D** [42, 68, 79]. The best-known enhancement operators in Abstract Interpretation theory include: reduced product, disjunctive completion, reduced cardinal power [56], tensor product [137], open product, pattern completion [40, 41], functional dependencies [78], complete shell [80], and logical product [88]. The reduced product is a refinement of the Cartesian product as it enhance the accuracy of the information tracked by one domain using the information tracked by the other, and viceversa, instead of running the two abstract analysis in parallel. The disjunctive completion [56] enhances an abstract domain by adding denotations for concrete disjunctions of its values [68], while the reduced cardinal power [56] captures disjunctive information over abstract states [42], being suitable for relational analysis. A further Cartesian product refinement is the open product, presented by Cortesi et al. in [40, 41]. It works on open abstract interpretations, which include queries and open operations. In [80], Giacobazzi et al. presented a constructive way to obtain the so-called complete shell of an abstract interpretation, i.e., a domain transformer which includes the minimal number of abstract point to a certain abstract domain $\overline{\mathbf{D}}$ in order to make it complete with respect to a certain operation of interest. The benefits of working with complete string abstract domains have been proven in Chapter 5. Finally, the logical product [88], is more precise than the reduced product, and it combines lattices which are defined over convex, stably infinite and disjoint theories.

In the following we recall the Cartesian, reduced and Granger products.

Cartesian Product

Let $\overline{\mathbf{D}}$ and $\overline{\mathbf{D}}'$ be two abstract domains representing sound approximations of the same concrete domain $\mathbf{D}$ and let $\mathfrak{C} = \overline{\mathbf{D}} \times \overline{\mathbf{D}}'$ be their Cartesian product. The abstract elements in $\mathfrak{C}$ are pairs $(\overline{\mathbf{d}}, \overline{\mathbf{d}}')$ such that $\overline{\mathbf{d}} \in \overline{\mathbf{D}}$ and $\overline{\mathbf{d}}$emd$' \in \overline{\mathbf{D}}'$. The partial order, upper and lower bound and widening operators are defined as the component-wise application of the corresponding operators of the two domains. The Cartesian product is a lattice. The abstraction function on $\mathfrak{C}$ maps a concrete element $\mathbf{d} \in \mathbf{D}$ to the pair $(\alpha_{\overline{\mathbf{D}}}(\mathbf{d}), \alpha_{\overline{\mathbf{D}}'}(\mathbf{d}))$, while the concretization function on $\mathfrak{C}$ maps an abstract element to $\gamma_{\overline{\mathbf{D}}}(\overline{\mathbf{d}}) \sqcap_{\mathbf{D}} \gamma_{\overline{\mathbf{D}}'}(\overline{\mathbf{d}}')$. Then, the Cartesian product forms a Galois Connection with the concrete domain. An abstract domain functor is a function from the parameter abstract domains $\overline{\mathbf{D}}_1, \overline{\mathbf{D}}_2, ..., \overline{\mathbf{D}}_n$ to a new abstract domain $\overline{\mathbf{D}}(\overline{\mathbf{D}}_1, \overline{\mathbf{D}}_2, ..., \overline{\mathbf{D}}_n)$. The abstract domain functor $\overline{\mathbf{D}}(\overline{\mathbf{D}}_1, \overline{\mathbf{D}}_2, ..., \overline{\mathbf{D}}_n)$ composes abstract domain properties of the parameter abstract domains to build a new class of abstract properties and operations [57].

Reduced Product

The reduced product [56] combines abstract domains mutually refining them. Informally speaking, it improves the precision of the information tracked by one domain exploiting the information tracked by the other, and vice-versa [42]. Let $(\overline{\mathbf{d}}_1, \overline{\mathbf{d}}_2) \in \mathfrak{C}$ be the Cartesian product of $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$. The reduced product is a Cartesian product equipped with a reduction operator. In particular, the reduced product search for the smallest element $(\overline{\mathbf{d}}_1^{'}, \overline{\mathbf{d}}_2^{'})$ such that: $\gamma_{\overline{\mathbf{D}}_1}(\overline{\mathbf{d}}_1^{'}) \subseteq \gamma_{\overline{\mathbf{D}}_1}(\overline{\mathbf{d}}_1)$, $\gamma_{\overline{\mathbf{D}}_2}(\overline{\mathbf{d}}_2^{'}) \subseteq \gamma_{\overline{\mathbf{D}}_2}(\overline{\mathbf{d}}_2)$, and $\gamma_{\overline{\mathbf{D}}_1}(\overline{\mathbf{d}}_1^{'}) \cap \gamma_{\overline{\mathbf{D}}_2}(\overline{\mathbf{d}}_2^{'}) = \gamma_{\overline{\mathbf{D}}_1}(\overline{\mathbf{d}}_1) \cap \gamma_{\overline{\mathbf{D}}_2}(\overline{\mathbf{d}}_2)$. Formally, the reduction operator, $\rho : \mathfrak{C} \to \mathfrak{C}$, is defined as follows: let $\mathfrak{c}$ denote the pair $(\overline{\mathbf{d}}_1, \overline{\mathbf{d}}_2)$ and $\mathfrak{c}^*$ denote the pair $(\overline{\mathbf{d}}_1^{'}, \overline{\mathbf{d}}_2^{'})$ then, $\rho(\mathfrak{c}) = \bigsqcap_{\mathfrak{C}}\{\mathfrak{c}^* \in \mathfrak{C} : \gamma_{\mathfrak{C}}(\mathfrak{c}) \sqsubseteq_{\mathfrak{C}} \gamma_{\mathfrak{C}}(\mathfrak{c}^*)\}$, where $(\overline{\mathbf{d}}_1, \overline{\mathbf{d}}_2) \sqsubseteq_{\mathfrak{C}} (\overline{\mathbf{d}}_1^{'}, \overline{\mathbf{d}}_2^{'}) \Leftrightarrow \overline{\mathbf{d}}_1 \sqsubseteq_{\overline{\mathbf{D}}_1} \overline{\mathbf{d}}_1^{'}$ and $\overline{\mathbf{d}}_2 \sqsubseteq_{\overline{\mathbf{D}}_2} \overline{\mathbf{d}}_2^{'}$. In the following we will refer to the reduced product between two abstract elements $\overline{\mathbf{d}}_1$ and $\overline{\mathbf{d}}_2$ as $\overline{\mathbf{d}}_1 \otimes \overline{\mathbf{d}}_2$.

Granger Product

The Granger product [87] is an approximation of the reduction operator. It is based on two refinement operators iterated until a fixpoint is reached (or, in other words, when the smallest reduction is obtained). Each of these operators takes advantage of the information of one of the two domains involved in the product and it improves the information of the other using the information of the first one [42]. Let $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$ be two abstract domains, $\overline{\mathbf{d}}_1$ and $\overline{\mathbf{d}}_2$ be abstract elements belonging to $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$ respectively, and $\mathfrak{C}$ be their Cartesian product. The Granger operators are defined as follows: $\rho_1 : \mathfrak{C} \to \overline{\mathbf{D}}_1$ and $\rho_2 : \mathfrak{C} \to \overline{\mathbf{D}}_2$. In order to have a sound over-approximation of the reduction operator, ρ_1 and ρ_2 have to satisfy the following conditions: let $\mathfrak{c}$ denote the pair $(\overline{\mathbf{d}}_1, \overline{\mathbf{d}}_2) \in \mathfrak{C}$ then, $\rho_1(\mathfrak{c}) \sqsubseteq_{\overline{\mathbf{D}}_1} \overline{\mathbf{d}}_1 \wedge \gamma_{\mathfrak{C}}((\rho_1(\mathfrak{c}), \overline{\mathbf{d}}_2)) = \gamma_{\mathfrak{C}}(\mathfrak{c})$ and $\rho_2(\mathfrak{c}) \sqsubseteq_{\overline{\mathbf{D}}_2} \overline{\mathbf{d}}_2 \wedge \gamma_{\mathfrak{C}}((\overline{\mathbf{d}}_1, \rho_2(\mathfrak{c}))) = \gamma_{\mathfrak{C}}(\mathfrak{c})$.

Chapter 3

STRING ANALYSIS FOR C

In this chapter we present the M-String abstract domain for strings manipulating `C` code. In particular, M-String is a refinement of the segmentation approach to array content representation proposed by Cousot et al. [57] and it is tailored to detect the presence of common `C` strings managements errors. An experimental evaluation is also given, to prove its effectiveness.

The content of this chapter reports contributions published in [47, 48, 121].

Chapter Structure

Section 3.1 introduces the problem of handling `C` strings and explains our contribution. Section 3.2 recalls the array segmentation abstract domain [57] on which M-String is based. Section 3.3 gives the syntax of some operations of interest. Section 3.4 defines the concrete domain and semantics. Section 3.5 presents the M-String abstract domain for `C` character arrays and its semantics, whose soundness is formally proved. Section 3.6 presents a general approach to abstraction as a program transformation and extends it to abstraction of program strings. Sections 3.7 and 3.8 present implementation and evaluation details of M-String abstraction. Section 3.9 concludes.

3.1 Introduction

`C` is still one of the mainstream programming languages [31], and a large portion of systems of critical relevance is written in this language, such as embedded systems. Unfortunately, `C` programs suffer of bugs, due to the way they are laid out in memory, which malicious parties may exploit to drive security attacks. Ensuring the correctness of such software is of great concern. Our main interest is guaranteeing the correctness of `C` programs that manage strings, because the incorrect string manipulation may lead to several catastrophic events, ranging from loss or exposure of sensitive data to crashes in critical software components.

Strings in `C` are not a basic data type. As a matter of facts, strings in `C` are represented by zero-terminated arrays of characters and there are libraries that provide functions which

allow operating on them [30]. `C` programs that manipulate strings can suffer from buffer overflows and related issues due to the possible discrepancy between the size of the string and the size of the array (buffer). A buffer overflow is a bug that affects `C` code when a buffer is accessed out of its bounds. In particular, an out-of-bounds write is a particular (and very dangerous) case of buffer overflow. Out-of-bounds read is less critical as a bug. It is important to design methods supporting the automatic correctness verification of string management in `C` programs for the previously mentioned reasons and also because buffer overflows are usually exploitable and can easily lead to arbitrary code execution [140]. Existing bugs can be identified by enhancing tools for code analysis, which can also reduce the risk of introducing new bugs and limit the occurrence of costly security incidents.

State of the Art

Static methods tailored to automatically identify buffer overflows have been extensively studied in the literature and several inference techniques were proposed and implemented: tainted data-flow analysis, constraint solvers for various theories (including string theories) and techniques based on them (e.g., symbolic execution), annotation analysis or string pattern matching analysis [156]. Furthermore, the above mentioned techniques and a large number of bug hunting tools based on static analysis have been implemented [62, 63, 97, 129, 175, 181].

For instance, in [107] authors introduced a backward compatible method of bounds checking of `C` programs, which leaves the representation of pointers unchanged, allowing inter-operation between checked and unchecked code, with recompilation confined to the modules where problems might occur. The just mentioned feature differentiates the proposed schema from previously existing techniques. In [62] the static verifier of `C` strings CSSV is introduced. Contracts are supplied to the tool, which acts in 4 stages, reducing the problem of checking code that manipulates string to checking code that manipulates integers. Finally, Splat, described in [183], is a tool that automatically generates test inputs, symbolically reasoning about lengths of input buffers.

Most of the existing string abstract domains are general-purpose domains, which focus on the generic aspects of strings, without accounting for the specifics of string handling by the different programming languages. However, it is often beneficial to consider specific aspects of string representation when designing abstract domains for program analysis. Referring to the `C` programming language, [108] has proposed an abstract domain for `C` strings which tracks both their length and the buffer allocated size into which they are contained. Combining it with the cell abstraction [134], such domain is able to describe relations between length of variables and offsets of pointers. Amadini et al. [10] have evaluated several abstract string domains (and their combinations) for analysis of `JavaScript` programs. In [142] the simplified regular expression domain for `JavaScript` analysis was

defined too. In addition to theoretical work, a number of tools based on the above mentioned abstract domains and their combinations have been designed and implemented [105, 112, 142, 162]. While dynamic languages heavily rely on strings and their analysis benefits greatly from tailored abstract domains, the specifics of the C approach to strings also earns attention.

Contribution

In this chapter, we introduce M-String, a new abstract domain tailored for the analysis of strings in C, whose elements:

- approximate sets of C character arrays;
- allow the abstraction of both shape information on the array structure and value information on the contained characters;
- highlight the presence of well-formed strings in the approximated character arrays.

M-String refines the segmentation approach to array representation introduced in [57]. M-String's goal is to detect the presence of common string management errors that may lead to undefined behaviours or, more specifically, which may result in buffer overflows. Moreover, keeping track of the content of the characters occurring after the first null character allows us to reduce the number of false positives. In fact, rewriting the first null character in the array is not always an error, as further occurrences of the null character may follow. M-String, such as the array segmentation-based representation introduced in [57], is parametric in two ways: both with respect to the representation of the indices of the array and with respect to the abstraction of the element values.

To provide evidence of the effectiveness of M-String, we extend LART [120], a tool which performs automatic abstraction on programs, making it supporting also sophisticated (non-scalar) domains such as M-String.

We extend LART along with DIVINE 4 [23], an explicit state model checker based on LLVM. This way, we can verify the correctness of operations on strings in C programs automatically. The experimental evaluation is performed by analyzing several C programs, ranging from quite simple to moderately complex, including parsers generated by `bison`, a tool which translates context-free grammars into C parsers. The results show the actual impact of an ad-hoc segmentation-based abstract domain on model checking of C programs.

3.2 FunArray

In the following we recall the array segmentation analysis presented in [57]. Notice that we slightly modify the notation to be consistent with the whole chapter. For more details, we invite the reader to refer directly to the original paper.

3.2.1 Array Concrete Representation

Let R_a be the set of concrete array environments. A concrete array environment $\theta \in R_a$ maps array variables $\mathtt{a} \in \mathtt{A}$ to their values $\theta(\mathtt{a}) \in \mathbf{A}$, such that:

- $\theta(\mathtt{a}) = (\rho, l_{\mathtt{a}}, h_{\mathtt{a}}, A_{\mathtt{a}})$ and,
- $\theta(\mathtt{a}) \in \mathbf{A} = R_v \times \mathbf{E} \times \mathbf{E} \times (\mathbb{Z} \to (\mathbb{Z} \times \mathbf{V}))$

where

1. R_v is the set of concrete variable environments. A concrete variable environment $\rho \in R_v$ maps variables (of basic types) $\mathtt{x} \in \mathtt{X}$ to their values $\rho(\mathtt{x}) \in \mathbf{V}$.
2. $\mathbf{E}$ is the set of program expressions made up of constants, variables, mathematical unary and binary operators. $l_{\mathtt{a}}, h_{\mathtt{a}} \in \mathbf{E}$ are expressions whose value[1], given by $[\![l_{\mathtt{a}}]\!]\rho$ and $[\![h_{\mathtt{a}}]\!]\rho$, respectively represents the lower bound and the upper bound of an array $\mathtt{a}$, i.e., the lower and the upper bound of its indexes range[2]. Notice that the value of an upper bound of an array concrete value corresponds to the index immediately after the one that points to the last memory block allocated to the array when it has been initialized. As usual, array indexes are 0-based.
3. $\mathbb{Z}$ is the set of integer numbers and $\mathbf{V}$ is the set of values. Let $\mathsf{idx}(\mathtt{a})$ be the set of indexes i of an array $\mathtt{a}$, i.e., $\mathsf{idx}(\mathtt{a}) = \{i \mid i \in [[\![l_{\mathtt{a}}]\!]\rho, [\![h_{\mathtt{a}}]\!]\rho)\} \subseteq \mathbb{Z}$ and[3], let $\mathsf{pair}(\mathtt{a})$ be the set of pairs (i, v) such that v is the value of the element indexed by i in an array $\mathtt{a}$, i.e., $\mathsf{pair}(\mathtt{a}) = \{(i, v) \mid i \in \mathsf{idx}(\mathtt{a}) \wedge [\![\mathtt{a}[i]]\!]\rho = v \in \mathbf{V}\} \subseteq \mathbb{Z} \times \mathbf{V}$. Thus, $A_{\mathtt{a}} : \mathsf{idx}(\mathtt{a}) \to \mathsf{pair}(\mathtt{a})$ is a function mapping the indexes of an array $\mathtt{a}$ to their corresponding pairs (index, indexed array value).

Example 3.1. Let `a` be a `C` integer array initialized as follows: `a[5] = {5,7,9,11,13}`. The concrete value of `a` is given by the tuple $\theta(\mathtt{a}) = (\rho, 0, 5, A_{\mathtt{a}})$, where the value of the lower and the upper bound of `a` are clear from the context and the codomain of the function

[1] In the following, for simplicity, expressions are evaluated to integers.

[2] According to the denotational semantics approach [153], in [57], the value of an arithmetic expression `e` is denoted by $[\![\mathtt{e}]\!]\rho$, where: (1) the double square brackets notation denotes the semantic evaluation function and, (2) ρ is an environment mapping program variables (which also may appear in `e`) to their value. Typically, $[\![\mathtt{x}]\!]\rho$ is equivalent to $\rho(\mathtt{x})$, with $\mathtt{x} \in \mathtt{X}$, and $[\![\mathtt{n}]\!]\rho$, where n is a constant, is equivalent to `n` itself. Thus, for example, if `e` is the expression $\mathtt{x} - 1$, its semantics $[\![\mathtt{x} - 1]\!]\rho$ is defined as $[\![\mathtt{x}]\!]\rho - [\![1]\!]\rho$, which corresponds to $\rho(\mathtt{x}) - 1$.

[3] Note that, in some cases, we use the interval notation to denote numeric sets, e.g., let $x, y \in \mathbb{Z}$, the interval between x and y (not included) is denoted by $[x, y)$ which corresponds to the set $\{w \in \mathbb{Z} \mid x \leq w < y\}$.

$A_{\mathtt{a}}$ is the set $\mathsf{pair}(\mathtt{a}) = \{(0,5),(1,7),(2,9),(3,11),(4,13)\}$. Moreover, let $\mathtt{b}$ denote the sub-array of $\mathtt{a}$ from position 2 to 3 included, its concrete value is given by $\theta(\mathtt{b}) = (\rho, 2, 4, A_{\mathtt{b}})$ such that $\mathsf{pair}(\mathtt{b}) = \{(2,9),(3,11)\}$.

⬡

Observe that this array representation allows reasoning about the correspondence between shape components of an array and actual values of the array elements.

3.2.2 Array Abstract Domain Functor

According to [57], the FunArray abstract domain $\overline{\mathbf{S}}$ (shortcut for $\overline{\mathbf{S}}(\overline{\mathbf{B}}, \overline{\mathbf{A}}, \overline{\mathbf{R}})$) allows representing a sequence of consecutive, non-overlapping and possibly empty segments that over-approximate a set of concrete array values in $\mathcal{P}(\mathbf{A})$, i.e., the powerset of $\mathbf{A}$. Each segment represents a sub-array whose elements share the same property (e.g., being positive integer values) and is surrounded by the so-called segment bounds, i.e., abstractions on its lower and upper bound.

Example 3.2. Consider the integer array `a[5] = {5,7,9,10,12}`. As an abstraction of `a` we may consider $\{0\}$ `odd` $\{3\}$ `even` $\{5\}$ saying that the array contains odd numbers in the first three elements (indexed from 0 to 2) and two even elements (indexed from 3 to 4).

⬡

The elements of FunArray belong to the set $\overline{\mathbf{S}} = \{(\overline{\mathbf{B}} \times \overline{\mathbf{A}}) \times (\overline{\mathbf{B}} \times \overline{\mathbf{A}} \times \{\text{␣},?\})^{k} \times (\overline{\mathbf{B}} \times \{\text{␣},?\}) \mid k \geq 0\} \cup \{\bot_{\overline{\mathbf{S}}}\}$, and have the form $\overline{\mathbf{b}}_1\overline{\mathbf{p}}_1\overline{\mathbf{b}}_2[?_2]\overline{\mathbf{p}}_2 \ldots \overline{\mathbf{p}}_{n-1}\overline{\mathbf{b}}_n[?_n]$ where

1. $\overline{\mathbf{B}}$ is the segment bound abstract domain, approximating array indexes, with abstract properties $\overline{\mathbf{b}}_i \in \overline{\mathbf{B}}$ such that $i \in [1, n]$ and $n > 1$.

   We denote by $\overline{\mathbf{E}}$ the set of expressions of canonical form $\overline{\mathtt{x}} + k$, where $\overline{\mathtt{x}} \in \overline{\mathbf{X}}$ and $k \in \mathbb{Z}$. The segment bounds $\overline{\mathbf{b}}_i$ are sets of expressions $\{\overline{\mathbf{e}}_i^1, \ldots, \overline{\mathbf{e}}_i^m\}$, such that $\overline{\mathbf{e}}_i^j \in \overline{\mathbf{E}}$. The variable abstract domain $\overline{\mathbf{X}}$ encodes program variables, i.e., $\overline{\mathbf{X}} = \mathtt{X} \cup \{\mathtt{v}_0\}$, where $\mathtt{v}_0$ is a special variable whose value is assumed to be zero. Moreover, $\overline{\mathbf{b}}_i = \emptyset$ denotes unreachability; if $\overline{\mathbf{b}}_i \neq \emptyset$, the expressions appearing in a segment bound are all equivalent symbolic denotations of some concrete value (generally unknown in the abstract representation except when one of the $\overline{\mathbf{e}}_i^j$ is a constant). Thus, $\overline{\mathbf{B}}$ depends on the expression abstract domain $\overline{\mathbf{E}}$ which, in turn, depends on the variable abstract domain $\overline{\mathbf{X}}$.

2. $\overline{\mathbf{A}}$ is the array element abstract domain, with abstract properties $\overline{\mathbf{p}}_i \in \overline{\mathbf{A}}$. It denotes possible values of pairs (index, indexed array element) in a segment, for relational abstractions, array elements otherwise.

3. $\overline{\mathbf{R}}$ is the variable environment abstract domain, which depends on the variable abstract domain $\overline{\mathbf{X}}$, with abstract properties $\overline{\rho} \in \overline{\mathbf{R}}$.

4. the question mark, if present, expresses the possibility that the segment that precedes it may be empty. The question mark can never precede $\overline{\mathbf{b}}_1$. The space symbol ␣ in $\{␣, ?\}$ represents a non-empty segment.

Example 3.3. Let $\overline{\mathbf{A}}$ be the classical sign abstraction of numerical values. The segmentation abstract predicate $\{0\} + \{3\}? - \{5\}$ represents arrays of length 5, with either 0 or 3 positive elements followed by either 5 or 2 negative elements, respectively. For instance, it represents: $[7, 9, 10, -11, -9]$, $[6, 8, 5, -4, -2]$ and $[-2, -6, -3, -1, -4, -8]$. Please note that in the last case, the lack of positive values is justified by the presence of the question mark that says that the first segment is optional.

⬡

The *unification* algorithm, in [57], modifies two compatible segmentations[4] in order to align them with respect to the same list of bounds. The unification algorithm does not guarantee the maximality of the result, but it is always well-defined, it does terminates and it is deterministic. The partial order $\sqsubseteq_{\overline{\mathbf{S}}}$ over $\overline{\mathbf{S}}$ is defined over unified segmentations as well as the join $\sqcup_{\overline{\mathbf{S}}}$ and the meet $\sqcap_{\overline{\mathbf{S}}}$ operators. Please note that $\overline{\mathbf{S}}$ is not necessarily a lattice [73]. Moreover, $\overline{\mathbf{S}}$ does not respect the ACC, therefore, in order to ensure the convergence of the analysis, it is equipped with a widening operator $\nabla_{\overline{\mathbf{S}}}$. A narrowing operator $\Delta_{\overline{\mathbf{S}}}$, which improves the precision of the widening result, is also defined. $\nabla_{\overline{\mathbf{S}}}$ and $\Delta_{\overline{\mathbf{S}}}$ operate on unified segmentations.

Such an abstract array representation is effective for analyzing the content of arrays, but in the case of the `C` programming language where a string is defined as a null-terminating character array, it is not powerful enough to detect common string manipulation errors.

3.3 Syntax

Strings in the programming language `C` are arrays of characters, whose length is determined by the position of a terminating null character `'\0'`. Thus, for example, the string literal `"bee"` has four characters: `'b'`, `'e'`, `'e'`, `'\0'`. Moreover, `C` supports several string handling functions defined in the standard library `string.h`.

We focus on the most significant functions in the `string.h` header (see Table 3.1), manipulating null-terminated sequences of characters, plus the array elements access and update operations. Recall that `char`, `int` and `size_t` are data types in `C`, `const` is a qualifier applied to the declaration of any variable which specifies the immutability of its value, and `*str` denotes that `str` is a pointer variable.

[4] Two segmentations, $\overline{\mathbf{b}}_1^1 \ldots \overline{\mathbf{b}}_n^1[?_n^1]$ and $\overline{\mathbf{b}}_1^2 \ldots \overline{\mathbf{b}}_n^2[?_n^2]$, are compatible if $\overline{\mathbf{b}}_1^1 \cap \overline{\mathbf{b}}_1^2 \neq \emptyset$ and $\overline{\mathbf{b}}_n^1 \cap \overline{\mathbf{b}}_n^2 \neq \emptyset$.

`char *strcat(char *str1, const char *str2)`
`char *strchr(char *str, int c)`
`int strcmp(const char *str1, const char *str2)`
`char *strcpy(char *str1, const char *str2)`
`size_t strlen(const char *str)`

TABLE 3.1: String functions syntax in `C`

- `strcat` appends the null-terminated string pointed to by `str2` to the null-terminated string pointed to by `str1`. The first character of `str2` overwrites the null-terminator of `str1` and `str2` should not overlap `str1`. The string concatenation returns the pointer `str1`.
- `strchr` locates the first occurrence of `c` (converted to a `char`) in the string pointed to by `str`. The terminating null character is considered to be part of the string. The string character function returns a pointer to the located character, or a null pointer if the character does not occur in the string.
- `strcmp` lexicographically compares the string pointed to by `str1` to the string pointed to by `str2`. The string compare function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `str1` is greater than, equal to, or less than the string pointed to by `str2`.
- `strcpy` copies the null-terminated string pointed to by `str2` to the memory pointed to by `str1`. `str2` should not overlap `str1`. The string copy function returns the pointer `str1`.
- `strlen` computes the number of bytes in the string to which `str` points, not including the terminating null byte. The string length function returns the length of `str`.

Accessing an array element is possible indexing the array name. Let `i` be an index, the `i`-th element of the character array `str` is accessed by `str[i]`. On the other hand, a character array element is updated (or an assignment is performed to a character array element) by `str[i] = 'x'`, where `'x'` denotes a character literal.

As mentioned in Section 3.1, `C` does not guarantee bounds checking on array accesses and, in case of strings, the language does not ensure that the latter are null-terminated. As a consequence, improper string manipulation leads to several vulnerabilities and exploits [154]. For instance, if non null-terminated strings are passed to the functions above, the latter may return misleading results or read out of the array bound. Moreover, since `strcat` and `strcpy` do not allow the size of the destination array `str1` to be specified, they are frequent sources of buffer overflows.

3.4 Concrete Domain and Semantics

Our aim is to capture the presence of well-formed strings in `C` character arrays, to avoid undesired execution behaviours that may be security relevant. To reach our goal, we propose a character array concrete value which highlights the occurrence of null characters in it and we introduce the notion of *string of interest* of an array of chars. The concrete semantics relative to the operations presented in Section 3.3 is also given.

3.4.1 Character Array Concrete Representation

Let $\mathbf{C}$ be a finite set of characters representable by the character encoding in use equipped with a top element $\top_{\mathbf{C}}$ representing an unknown value and let $\mathtt{M}$ be the set of character array variables, such that $\mathtt{M} \subseteq \mathtt{A}$ (with $\mathtt{A}$ being the set of array variables - of any type - presented in Section 3.2.2). Then, the operational semantics of character array variables are concrete array environments $\mu \in \mathrm{R_m}$ mapping character arrays $\mathtt{m} \in \mathtt{M}$ to their values $\mu(\mathtt{m})$. Precisely:

- $\mu(\mathtt{m}) = (\rho, \mathrm{l_m}, \mathrm{h_m}, \mathrm{M_m}, \mathrm{N_m})$ and,
- $\mu(\mathtt{m}) \in \mathbf{M} = \mathrm{R_v} \times \mathbf{E} \times \mathbf{E} \times (\mathbb{Z} \to (\mathbb{Z} \times \mathbf{C})) \times \mathbb{Z}$

so that $\mathrm{R_m}$ is a map from $\mathtt{M}$ to $\mathbf{M}$, where $\mathrm{R_v}$ and $\mathbf{E}$ are the concrete variable environment and the expression domain defined in Section 3.2.2 respectively, $\mathbb{Z}$ is the integer domain and $\mathbf{C}$ is the character set introduced above. Notice that with respect to the concrete array environment θ introduced in Section 3.2.2, the function μ returns as a last component the set of indexes which map to the string terminating characters $\mathrm{N_m} = \{\mathrm{i} \mid \mathrm{i} \in \mathsf{idx}(\mathtt{m}) \wedge \mathrm{M_m}(\mathrm{i}) = (\mathrm{i},\text{'\textbackslash 0'})\}$, with $\mathsf{idx}(\mathtt{m})$ being the domain of the function $\mathrm{M_m}$. On the other hand, $\mathrm{M_m}$ behaves exactly as $\mathrm{A_a}$ in $\theta(\mathtt{a})$, mapping each index i of the considered array to the pair of the index i and the indexed array element v.

Thus, $\mathbf{M}$ extends $\mathbf{A}$ (cf. Section 3.2.2) by adding a parameter that takes into account the presence of null characters in a character array. For well-formed strings, $\mathrm{N_m}$ cannot be empty. Moreover, character array elements which have not been initialized are mapped to the top value $\top_{\mathbf{C}}$ as they may be values already present in the memory assigned to the locations array itself.

Example 3.4. Let `m` be a `C` character array initialized as follows: `m[6] = {'b','e','e','\0','b'}`. The concrete value of `m` is given by the tuple $\mu(\mathtt{m}) = (\rho, 0, 6, \mathrm{M_m}, \mathrm{N_m})$, where the codomain of the function $\mathrm{M_m}$ is the set $\mathsf{pair}(\mathtt{m}) = \{(0,\text{'b'}), (1,\text{'e'}), (2,\text{'e'}), (3,\text{'\textbackslash 0'}), (4,\text{'b'}), (5, \top_{\mathbf{C}})\}$ and $\mathrm{N_m}$ is the singleton $\{3\}$, being the array cell of index `3` the only one certainly containing a null character.

⬡

String of Interest

We formally define the string of interest of a character array as the sequence of its elements up to the first terminating one (included).

Definition 3.1 (String of interest)**.** Let $\mathtt{m} \in \mathtt{M}$ be an array of characters with concrete value $\mu(\mathtt{m}) = (\rho,\ \mathrm{l}_{\mathtt{m}}, \mathrm{h}_{\mathtt{m}}, \mathrm{M}_{\mathtt{m}}, \mathrm{N}_{\mathtt{m}})$. The string of interest of the character array described by $\mu(\mathtt{m})$ is defined as follows:

$$\mathsf{strInt}(\mu(\mathtt{m})) = \begin{cases} \langle \mathrm{v}_{\mathrm{i}} : \mathrm{i} \in [[\![\mathrm{l}_{\mathtt{m}}]\!]\rho, \mathsf{min}(\mathrm{N}_{\mathtt{m}})] \wedge \mathrm{M}_{\mathtt{m}}(\mathrm{i}) = (\mathrm{i}, \mathrm{v})\rangle & \text{if } \mathrm{N}_{\mathtt{m}} \neq \emptyset \\ \mathtt{undef} & \text{otherwise} \end{cases}$$

with v_{i} denoting the character value which occurs in the pair (i, v), and with $\mathsf{min}(\mathrm{N}_{\mathtt{m}})$ denoting the minimum element of $\mathrm{N}_{\mathtt{m}}$.

△

Example 3.5. Consider the concrete character array value of Example 3.4. Its string of interest is the sequence of characters `"bee\0"`.

⬡

Our definition of string of interest of character arrays allows us to distinguish well-formed strings and avoid bad usage of arrays of characters. If the null character appears at the first index of a character array, then we refer to its string of interest as null (`null`). In general, we refer to character arrays which contain a well-defined or null string of interest as character arrays which contain a *well-formed string*.

Moreover, when allocated memory capacity is not sufficient for a declared character array, the system writes a null character outside the array, occupying memory that is not destined for it and causing a buffer overflow. We do not represent this system behaviour, since it leads to an undefined one, so we simply consider the string of interest of such character arrays as undefined (`undef`).

3.4.2 Concrete Domain

As a concrete domain for array of characters we refer to the complete lattice $\mathcal{P}(\mathbf{M})$ defined as $(\mathcal{P}(\mathbf{M}), \subseteq_{\mathcal{P}(\mathbf{M})}, \bot_{\mathcal{P}(\mathbf{M})}, \top_{\mathcal{P}(\mathbf{M})}, \cup_{\mathcal{P}(\mathbf{M})}, \cap_{\mathcal{P}(\mathbf{M})})$ where: $\mathcal{P}(\mathbf{M})$ is the powerset of concrete character array values, the set inclusion $\subseteq_{\mathcal{P}(\mathbf{M})}$ corresponds to the partial order, the bottom element $\bot_{\mathcal{P}(\mathbf{M})}$ is the emptyset $\emptyset$, the top element $\top_{\mathcal{P}(\mathbf{M})}$ is the superset of any subset of $\mathbf{M}$ (i.e., $\mathbf{M}$ itself), the set union $\cup_{\mathcal{P}(\mathbf{M})}$ denotes the least upper bound and, the set intersection $\cap_{\mathcal{P}(\mathbf{M})}$ denotes the greatest lower bound.

We stress the fact that the concrete domain we introduce is used as a framework that helps us in creating the abstract representation, and it is not how the (concrete) character array values are actually represented in `C` programs.

3.4.3 Concrete Semantics

To formalize the concrete semantics of the `C` standard library functions from `string.h` introduced in Section 3.3, the following auxiliary functions embedding (emb), extraction (ext), comparison (cmp) and substitution (sub) over single concrete character array values need to be introduced.

Definition 3.2 (Embedding)**.** Let $\mu(\mathtt{m}_1), \mu(\mathtt{m}_2) \in \mathbf{M}$ be two concrete character array values and $[l_1, u_1] \subseteq [[\![l_{\mathtt{m}_1}]\!]\rho, [\![h_{\mathtt{m}_1}]\!]\rho)$, $[l_2, u_2] \subseteq [[\![l_{\mathtt{m}_2}]\!]\rho, [\![h_{\mathtt{m}_2}]\!]\rho)$ be two indexes ranges of the same length. The function $\mathsf{emb}(\mu(\mathtt{m}_1), [l_1, u_1], \mu(\mathtt{m}_2), [l_2, u_2])$ embeds the sequence of characters of $\mu(\mathtt{m}_2)$ which occurs from the index l_2 to the index u_2 into $\mu(\mathtt{m}_1)$ from the index l_1 to the index u_1. Formally, $\mathsf{emb}(\mu(\mathtt{m}_1), [l_1, u_1], \mu(\mathtt{m}_2), [l_2, u_2]) = \mu(\mathtt{m}_1)'$ such that:

- $[\![l_{\mathtt{m}_1'}]\!]\rho = [\![l_{\mathtt{m}_1}]\!]\rho$ and $[\![h_{\mathtt{m}_1'}]\!]\rho = [\![h_{\mathtt{m}_1}]\!]\rho$
- $M_{\mathtt{m}_1'}$:
  - $\forall i \in [[\![l_{\mathtt{m}_1'}]\!]\rho, l_1)$: $M_{\mathtt{m}_1'}(i) = (i, v)$ s.t. $k = i \wedge M_{\mathtt{m}_1}(k) = (k, v)$
  - $\forall i \in [l_1, u_1]$: $M_{\mathtt{m}_1'}(i) = (i, v)$ s.t. $k = l_2 + (i - l_1) \wedge M_{\mathtt{m}_2}(k) = (k, v)$
  - $\forall i \in (l_1, [\![h_{\mathtt{m}_1'}]\!]\rho)$: $M_{\mathtt{m}_1'}(i) = (i, v)$ s.t. $k = i \wedge M_{\mathtt{m}_1}(k) = (k, v)$
- $N_{\mathtt{m}_1'} = (N_{\mathtt{m}_1} \setminus \{i \mid i \in [l_1, u_1]\}) \cup \{i \mid i \in [l_1, u_1] \wedge k = l_2 + (i - l_1) \wedge M_{\mathtt{m}_2}(k) = (k, \text{'\textbackslash 0'})\}$

△

Example 3.6. Let $\mu(\mathtt{m}_1) = (\rho, 0, 7, M_{\mathtt{m}_1}, N_{\mathtt{m}_1})$ and $\mu(\mathtt{m}_2) = (\rho, 0, 6, M_{\mathtt{m}_2}, N_{\mathtt{m}_2})$ be two concrete character array values such that:

- $\mathsf{pair}(\mathtt{m}_1) = \{(0,\text{'a'}), (1,\text{'a'}), (2,\text{'a'}), (3,\text{'\textbackslash 0'}), (4,\text{'a'}), (5,\text{'a'}), (6,\text{'a'})\}$
- $\mathsf{pair}(\mathtt{m}_2) = \{(0,\text{'b'}), (1,\text{'b'}), (2,\text{'b'}), (3,\text{'b'}), (4,\text{'b'}), (5,\text{'\textbackslash 0'})\}$

Moreover, consider the intervals of equal length:

- $[2, 4]_{\mathtt{m}_1} \subseteq [[\![l_{\mathtt{m}_1}]\!]\rho, [\![h_{\mathtt{m}_1}]\!]\rho)$
- $[3, 5]_{\mathtt{m}_2} \subseteq [[\![l_{\mathtt{m}_2}]\!]\rho, [\![h_{\mathtt{m}_2}]\!]\rho)$

The function $\mathsf{emb}(\mu(\mathtt{m}_1), [2, 4]_{\mathtt{m}_1}, \mu(\mathtt{m}_2), [3, 5]_{\mathtt{m}_2}) = \mu(\mathtt{m}_1)'$ where:

- $[\![l_{\mathtt{m}_1'}]\!]\rho = [\![l_{\mathtt{m}_1}]\!]\rho = 0$ and $[\![h_{\mathtt{m}_1'}]\!]\rho = [\![h_{\mathtt{m}_1}]\!]\rho = 7$
- $\mathsf{pair}(\mathtt{m}_1') = \{(0,\text{'a'}), (1,\text{'a'}), (2,\text{'b'}), (3,\text{'b'}), (4,\text{'\textbackslash 0'}), (5,\text{'a'}), (6,\text{'a'})\}$
- $N_{\mathtt{m}_1'} = \{4\}$

⬡

Definition 3.3 (Extraction)**.** Let $\mu(\mathtt{m}) \in \mathbf{M}$ be a concrete character array value and $[l, u] \subseteq [[\![l_{\mathtt{m}}]\!]\rho, [\![h_{\mathtt{m}}]\!]\rho)$ be an indexes range. The function $\mathsf{ext}(\mu(\mathtt{m}), [l, u])$ extracts the sequence of characters which occurs in $\mu(\mathtt{m})$ from the index l to the index u. Formally, $\mathsf{ext}(\mu(\mathtt{m}), [l, u]) = \mu(\mathtt{m})'$ such that:

- $[\![l_{\mathtt{m}'}]\!]\rho = l$ and $[\![h_{\mathtt{m}'}]\!]\rho = u + 1$
- $M_{\mathtt{m}'} : \forall i \in [[\![l_{\mathtt{m}'}]\!]\rho, [\![h_{\mathtt{m}'}]\!]\rho)$: $M_{\mathtt{m}'}(i) = (i, v)$ s.t. $k = i \wedge M_{\mathtt{m}}(k) = (k, v)$
- $N_{\mathtt{m}'} = N_{\mathtt{m}} \setminus \{i \mid i \notin [l, u]\}$

△

Example 3.7. Let $\mu(\mathtt{m}_1)$ be the character array concrete value of Example 3.6 and $[1, 3]_{\mathtt{m}_1} \subseteq [[\![l_{\mathtt{m}_1}]\!]\rho, [\![h_{\mathtt{m}_1}]\!]\rho)$ be an indexes range of $\mu(\mathtt{m}_1)$. The function $\mathsf{ext}(\mu(\mathtt{m}_1), [1, 3]_{\mathtt{m}_1}) = \mu(\mathtt{m}_1)'$ such that:

- $[\![l_{\mathtt{m}_1'}]\!]\rho = 1$ and $[\![h_{\mathtt{m}_1'}]\!]\rho = 4$
- $\mathsf{pair}(\mathtt{m}_1') = \{(1,'a'), (2,'a'), (3,'\backslash 0')\}$
- $N_{\mathtt{m}_1'} = \{3\}$

⬡

Definition 3.4 (Comparison)**.** Let $\mu(\mathtt{m}_1)$, $\mu(\mathtt{m}_2) \in \mathbf{M}$ be two concrete character array values which contain a fully initialized well-formed string of interest, i.e., no $\top_{\mathbf{C}}$ occurs. The function $\mathsf{cmp}(\mu(\mathtt{m}_1), \mu(\mathtt{m}_2))$ (cf. Algorithm 1) lexicographically compares the strings of interest of $\mu(\mathtt{m}_1)$ and $\mu(\mathtt{m}_2)$ and it returns an integer value n which denotes the lexicographic distance between them.

Notice that n will be strictly smaller than zero if $\mathsf{strInt}(\mu(\mathtt{m}_1))$ precedes $\mathsf{strInt}(\mu(\mathtt{m}_2))$ in lexicographic order, equal to zero if $\mathsf{strInt}(\mu(\mathtt{m}_1))$ and $\mathsf{strInt}(\mu(\mathtt{m}_2))$ are lexicographically equivalent, and strictly greater than zero if $\mathsf{strInt}(\mu(\mathtt{m}_1))$ follows $\mathsf{strInt}(\mu(\mathtt{m}_2))$ in lexicographic order.

△

Example 3.8. Let $\mu(\mathtt{m}_1)$ and $\mu(\mathtt{m}_2)$ be the character array concrete values of Example 3.6. Both of them contain a fully initialized well-formed string of interest and the function $\mathsf{cmp}(\mu(\mathtt{m}_1), \mu(\mathtt{m}_2))$ computes the lexicographic distance between them. Precisely, the procedure stops after the first iteration of the for loop (cf. Algorithm 1) and, assuming ASCII as the character encoding set, it returns the value –1, i.e., $n = 97 - 98$, which means that $\mathsf{strInt}(\mu(\mathtt{m}_1))$ lexicographically precedes $\mathsf{strInt}(\mu(\mathtt{m}_2))$.

⬡

Algorithm 1 Lexicographic comparison of concrete character array values.

Input: two concrete character array values $\mu(\mathtt{m}_1), \mu(\mathtt{m}_2) \in \mathbf{M}$ such that:

- both $N_{\mathtt{m}_1}$ and $N_{\mathtt{m}_2}$ are different from the emptyset and,
- for $i_1 \in [[\![l_{\mathtt{m}_1}]\!]\rho, \mathsf{min}(N_{\mathtt{m}_1})), i_2 \in [[\![l_{\mathtt{m}_2}]\!]\rho, \mathsf{min}(N_{\mathtt{m}_2}))$:

$$M(i_1) \neq (i_1, \top_{\mathbf{C}}) \text{ and } M(i_2) \neq (i_2, \top_{\mathbf{C}})$$

Output: an integer value n.

```
1: n = 0, i1 = [[l_m1]]ρ, i2 = [[l_m2]]ρ
2: while i1 ∈ [[[l_m1]]ρ, min(N_m1)] ∧ i2 ∈ [[[l_m2]]ρ, min(N_m2)] do
3:     n = v_i1 − v_i2
4:     if n ≠ 0 then
5:         return n
6:     else
7:         i1 = i1 + 1, i2 = i2 + 1
8: return n
```

Definition 3.5 (Substitution)**.** Let $\mu(\mathtt{m}) \in \mathbf{M}$ be a concrete character array value, $z \in [[\![l_{\mathtt{m}}]\!]\rho, [\![h_{\mathtt{m}}]\!]\rho)$ be an index and $\mathtt{c} \in \mathbf{C}$ be a character. The function $\mathsf{sub}(\mu(\mathtt{m}), z, \mathtt{c})$ substitutes the character which appears in $\mu(\mathtt{m})$ at the index z with the character $\mathtt{c}$. Formally, $\mathsf{sub}(\mu(\mathtt{m}), z, \mathtt{c}) = \mu(\mathtt{m})'$ such that:

- $[[\![l_{\mathtt{m}'}]\!]\rho = [[\![l_{\mathtt{m}}]\!]\rho$ and $[[\![h_{\mathtt{m}'}]\!]\rho = [[\![h_{\mathtt{m}}]\!]\rho$
- $M_{\mathtt{m}'}$:
  - $\forall i \in [[[\![l_{\mathtt{m}'}]\!]\rho, z)$: $M_{\mathtt{m}'}(i) = (i, v)$ s.t. $k = i \wedge M_{\mathtt{m}}(k) = (k, v)$
  - for $i = z$: $M_{\mathtt{m}'}(z) = (z, \mathtt{c})$
  - $\forall i \in (z, [[\![h_{\mathtt{m}'}]\!]\rho)$: $M_{\mathtt{m}'}(i) = (i, v)$ s.t. $k = i \wedge M_{\mathtt{m}}(k) = (k, v)$
- $N_{\mathtt{m}'} = \begin{cases} N_{\mathtt{m}} & \text{if } (z \in N_{\mathtt{m}} \wedge \mathtt{c} \text{ is null}) \ \vee (z \notin N_{\mathtt{m}} \wedge \mathtt{c} \text{ is not null}) \\ N_{\mathtt{m}} \setminus \{z\} & \text{if } z \in N_{\mathtt{m}} \wedge \mathtt{c} \text{ is not null} \\ N_{\mathtt{m}} \cup \{z\} & \text{otherwise} \end{cases}$

△

Example 3.9. Let $\mu(\mathtt{m}_1)$ be the character array concrete value of Example 3.6, the index z be equal to 4 and the character $\mathtt{c}$ be the null termination '\0'. The function $sub(\mu(\mathtt{m}_1), 4, \text{'\textbackslash 0'}) = \mu(\mathtt{m}_1)'$ such that:

- $[[\![l_{\mathtt{m}_1'}]\!]\rho = 0$ and $[[\![h_{\mathtt{m}_1'}]\!]\rho = 7$
- $\mathsf{pair}(\mathtt{m}_1') = \{(0,\text{'a'}), (1,\text{'a'}), (2,\text{'a'}), (3,\text{'\textbackslash 0'}), (4,\text{'\textbackslash 0'}), (5,\text{'a'}), (6,\text{'a'})\}$
- $N_{\mathtt{m}_1'} = \{3, 4\}$

⬡

Array Access

The semantic operator $\mathfrak{A}$, given the statement $\texttt{access}_j$ and a set of concrete character array values X in $\mathcal{P}(\mathbf{M})$ as parameter, returns a value in $\mathbf{C}$. In particular, $\texttt{access}_j(X)$ returns the character v which occurs at position j if all the character array values in X contain v at index j and the latter is well-defined (i.e., it ranges in the array bounds) for all the character array values in X; otherwise it returns $\top_{\mathbf{C}}$. Formally,

$$\mathfrak{A}[\![\texttt{access}_j]\!](X) = \begin{cases} v & \text{if } \forall \mu(\mathtt{m}) \in X : j \in [[\![l_{\mathtt{m}}]\!]\rho, [\![h_{\mathtt{m}}]\!]\rho) \text{ and } M_{\mathtt{m}}(j) = (j, v) \\ \top_{\mathbf{C}} & \text{otherwise} \end{cases}$$

String Concatenation

The semantic operator $\mathfrak{M}$, given a statement and some sets of concrete character array values in $\mathcal{P}(\mathbf{M})$ as parameters, returns a set of concrete character array values. When applied to $\texttt{strcat}(X_1, X_2)$, it returns all the possible embeddings in X_1 of a string of interest taken from X_2 if all the character array values (which belong to both X_1 and X_2) contain a well-formed string and the condition on the size of the destination character array values is fulfilled; otherwise it returns $\top_{\mathcal{P}(\mathbf{M})}$. Please note that the size condition ensures to perform the string concatenation only if the destination character array value is big enough to contain the string of interest of the source character array value, thus preventing undefined behaviours. Formally,

$$\mathfrak{M}[\![\texttt{strcat}]\!](X_1, X_2) = \begin{cases} X_1^{'} & \text{if } \forall \mu(\mathtt{m}_1) \in X_1 : \forall \mu(\mathtt{m}_2) \in X_2 : \\ & \quad \mathsf{strlnt}(\mu(\mathtt{m}_1)) \neq \mathsf{undef} \neq \mathsf{strlnt}(\mu(\mathtt{m}_2)) \wedge \\ & \quad\quad \texttt{size.condition} \text{ is } \texttt{true} \\ \top_{\mathcal{P}(\mathbf{M})} & \text{otherwise} \end{cases}$$

The `size.condition` is `true` if:

$$([\![h_{\mathtt{m}_1}]\!]\rho - [\![l_{\mathtt{m}_1}]\!]\rho) \geq \ [(\mathsf{min}(N_{\mathtt{m}_1}) - [\![l_{\mathtt{m}_1}]\!]\rho - 1) + (\mathsf{min}(N_{\mathtt{m}_2}) - [\![l_{\mathtt{m}_2}]\!]\rho)]$$

Moreover, $X_1^{'}$ is the set of $\mathsf{emb}(\mu(\mathtt{m}_1), [l_1, u_1], \mu(\mathtt{m}_2), [l_2, u_2])$ (cf. Definition 3.2), such that:

- $\mu(\mathtt{m}_1) \in X_1$, $l_1 = \mathsf{min}(N_{\mathtt{m}_1})$ and $u_1 = l_1 + (\mathsf{min}(N_{\mathtt{m}_2}) - [\![l_{\mathtt{m}_2}]\!]\rho)$
- $\mu(\mathtt{m}_2) \in X_2$, $l_2 = [\![l_{\mathtt{m}_2}]\!]\rho$ and $u_2 = \mathsf{min}(N_{\mathtt{m}_2})$

String Character

The semantic operator $\mathfrak{M}$, when applied to $\texttt{strchr}_v(X)$, returns the set of string of interest suffixes in X from the index corresponding to the first occurrence of the character v if all

the character array values in X contain a well-formed string containing v. Otherwise, if all the character array values in X contain a well-formed string in which does not occur the character v, it returns the emptyset (denoted by $\bot_{\mathcal{P}(\mathbf{M})}$); otherwise it returns $\top_{\mathcal{P}(\mathbf{M})}$. Formally,

$$\mathfrak{M}[\![\texttt{strchr}_{\mathrm{v}}]\!](\mathrm{X}) = \begin{cases} \mathrm{X}' & \text{if } \forall \mu(\mathtt{m}) \in \mathrm{X} : \mathsf{strlnt}(\mu(\mathtt{m})) \neq \mathtt{undef} \text{ and } \mathrm{v} \in \mathsf{strlnt}(\mu(\mathtt{m})) \\ \bot_{\mathcal{P}(\mathbf{M})} & \text{if } \forall \mu(\mathtt{m}) \in \mathrm{X} : \mathsf{strlnt}(\mu(\mathtt{m})) \neq \mathtt{undef} \text{ and } \mathrm{v} \notin \mathsf{strlnt}(\mu(\mathtt{m})) \\ \top_{\mathcal{P}(\mathbf{M})} & \text{otherwise} \end{cases}$$

In particular, X' is the set of $\mathsf{ext}(\mu(\mathtt{m}), [\mathrm{l}, \mathrm{u}])$ (cf. Definition 3.3), such that:

- $\mu(\mathtt{m}) \in \mathrm{X}$, $\mathrm{l} = \mathsf{min}(\{\mathrm{i} : \mathrm{i} \in [\![\mathrm{l}_{\mathtt{m}}]\!]\rho, \mathsf{min}(\mathrm{N}_{\mathtt{m}})] \wedge \mathrm{M}_{\mathtt{m}}(\mathrm{i}) = (\mathrm{i}, \mathrm{v})\})$ and $\mathrm{u} = \mathsf{min}(\mathrm{N}_{\mathtt{m}})$

String Compare

The semantic operator $\mathfrak{P}$, given the statement `strcmp` and two sets of concrete character array values $\mathrm{X}_1, \mathrm{X}_2$ in $\mathcal{P}(\mathbf{M})$ as parameters, returns a value in the set of integers equipped with a top element, i.e., $\mathbb{Z} \cup \top_{\mathbb{Z}}$. In particular, $\texttt{strcmp}(\mathrm{X}_1, \mathrm{X}_2)$ returns an integer value n which denotes the lexicographic distance between strings of interest in X_1 and X_2 if for all $\mu(\mathtt{m}_1) \in \mathrm{X}_1$ and $\mu(\mathtt{m}_2) \in \mathrm{X}_2$ the procedure $\mathsf{cmp}(\mu(\mathtt{m}_1), \mu(\mathtt{m}_2))$ (cf. Definition 3.4) returns n; otherwise it returns$\top_{\mathbb{Z}}$. Formally,

$$\mathfrak{P}[\![\texttt{strcmp}]\!](\mathrm{X}_1, \mathrm{X}_2) = \begin{cases} \mathrm{n} & \text{if } \forall \mu(\mathtt{m}_1) \in \mathrm{X}_1 : \forall \mu(\mathtt{m}_2) \in \mathrm{X}_2 : \mathsf{cmp}(\mu(\mathtt{m}_1), \mu(\mathtt{m}_2)) = \mathrm{n} \\ \top_{\mathbb{Z}} & \text{otherwise} \end{cases}$$

String Copy

The semantic operator $\mathfrak{M}$, when applied to $\texttt{strcpy}(\mathrm{X}_1, \mathrm{X}_2)$, behaves similarly to the string concatenation function above. Formally,

$$\mathfrak{M}[\![\texttt{strcpy}]\!](\mathrm{X}_1, \mathrm{X}_2) = \begin{cases} \mathrm{X}_1^{'} & \text{if } \forall \mu(\mathtt{m}_1) \in \mathrm{X}_1 : \forall \mu(\mathtt{m}_2) \in \mathrm{X}_2 : \\ & \quad \mathsf{strlnt}(\mu(\mathtt{m}_1)) \neq \mathtt{undef} \neq \mathsf{strlnt}(\mu(\mathtt{m}_2)) \wedge \\ & \quad \texttt{size.condition} \text{ is } \texttt{true} \\ \top_{\mathcal{P}(\mathbf{M})} & \text{otherwise} \end{cases}$$

The `size.condition` is `true` if:

$$([\![\mathrm{h}_{\mathtt{m}_1}]\!]\rho - [\![\mathrm{l}_{\mathtt{m}_1}]\!]\rho) \geq (\mathsf{min}(\mathrm{N}_{\mathtt{m}_2}) - [\![\mathrm{l}_{\mathtt{m}_2}]\!]\rho)$$

Moreover, $\mathrm{X}_1^{'}$ is the set of $\mathsf{emb}(\mu(\mathtt{m}_1), [\mathrm{l}_1, \mathrm{u}_1], \mu(\mathtt{m}_2), [\mathrm{l}_2, \mathrm{u}_2])$, such that:

- $\mu(\mathtt{m}_1) \in \mathrm{X}_1$, $\mathrm{l}_1 = [\![\mathrm{l}_{\mathtt{m}_1}]\!]\rho$ and $\mathrm{u}_1 = \mathrm{l}_1 + (\mathsf{min}(\mathrm{N}_{\mathtt{m}_2}) - [\![\mathrm{l}_{\mathtt{m}_2}]\!]\rho)$

- $\mu(\mathtt{m}_2) \in X_2$, $l_2 = [\![l_{\mathtt{m}_2}]\!]\rho$ and $u_2 = \mathsf{min}(N_{\mathtt{m}_2})$

String Length

The semantic operator $\mathfrak{L}$, given the statement `strlen` and a set of concrete character array values X in $\mathcal{P}(\mathbf{M})$ as parameter, returns a value in the set of integers equipped with a top element, i.e., $\mathbb{Z} \cup \top_{\mathbb{Z}}$. In particular, $\mathtt{strlen}(X)$ returns an integer value n which corresponds to the length of the sequence of characters before the first null one of the character arrays values in X if all the character array values in X contain a well-formed string of the same length; otherwise it returns $\top_{\mathbb{Z}}$. Formally,

$$\mathfrak{L}[\![\mathtt{strlen}]\!](X) = \begin{cases} n & \text{if } \forall \mu(\mathtt{m}) \in X : \mathsf{strInt}(\mu(\mathtt{m})) \neq \mathtt{undef} \wedge (\mathsf{min}(N_{\mathtt{m}}) - [\![l_{\mathtt{m}}]\!]\rho) = n \\ \top_{\mathbb{Z}} & \text{otherwise} \end{cases}$$

Array Update

The semantic operator $\mathfrak{M}$, when applied to $\mathtt{update}_{j,v}(X)$, returns the set of character array values in X where the character that occurs at position j has been substituted with the character v if the index j is well-defined for all the character array values in X; otherwise it returns $\top_{\mathcal{P}(\mathbf{M})}$. Formally,

$$\mathfrak{M}[\![\mathtt{update}_{j,v}]\!](X) = \begin{cases} X' & \text{if } \forall \mu(\mathtt{m}) \in X : j \in [[\![l_{\mathtt{m}}]\!]\rho, [\![h_{\mathtt{m}})]\!]\rho) \\ \top_{\mathcal{P}(\mathbf{M})} & \text{otherwise} \end{cases}$$

In particular, X' is the set of $\mathsf{sub}(\mu(\mathtt{m}), j, v)$ (cf. Definition 3.5).

3.5 M-String

In the previous section we defined the concrete value of a character array, which highlights the presence of a well-formed string in it. Moreover, we presented our concrete domain $\mathcal{P}(\mathbf{M})$, made of sets of character array values, and its concrete semantics of some operations of interest. In the following we formalize the M-String abstract domain, which approximates elements in $\mathcal{P}(\mathbf{M})$, and its semantics for which soundness is proved.

3.5.1 Character Array Abstract Domain Functor

The M-String ($\overline{\mathbf{M}}$) abstract domain approximates sets of concrete character array values with a pair of segmentations that highlights the nature of their strings of interest. The elements of the domain are split segmentation abstract predicates. As for FunArray (recalled in Section 3.2.2), segments represent sequences of characters which share the same property and are delimited by the so-called segment bounds. More precisely, the M-String abstract domain is a functor given by $\overline{\mathbf{M}}(\overline{\mathbf{B}}, \overline{\mathbf{C}}, \overline{\mathbf{R}})$ where

1. $\overline{\mathbf{B}}$ denotes the abstraction of segment bounds, equipped with the addition ($\oplus_{\overline{\mathbf{B}}}$) and subtraction ($\ominus_{\overline{\mathbf{B}}}$) operations.

2. $\overline{\mathbf{C}}$ is the abstraction of the character array elements, it is signed, it contains the value 0, and it is equipped with isNull, a special monotonic function lifting abstract elements in $\overline{\mathbf{C}}$ to a value in the set {`true`, `false`, `maybe`} and with subtraction ($\ominus_{\overline{\mathbf{C}}}$).

3. $\overline{\mathbf{R}}$ denotes the abstraction of scalar variable environments (cf. Section 3.2.2). Namely, the constant propagation domain on the set of variables `X`.

Elements of M-String belong to the set $\overline{\mathbf{M}} \triangleq (\overline{\mathbf{M}}_{\mathrm{s}}, \overline{\mathbf{M}}_{\mathrm{ns}}) \cup \{\bot_{\overline{\mathbf{M}}}, \top_{\overline{\mathbf{M}}}\}$ such that:

- $\overline{\mathbf{M}}_{\mathrm{s}}$ corresponds to $\{\{\overline{\mathbf{B}}\times\overline{\mathbf{C}}\}\times\{\overline{\mathbf{B}}\times\overline{\mathbf{C}}\times\{\sqcup, ?\}\}^{\mathrm{k}}\times\{\overline{\mathbf{B}}\times\{\sqcup, ?\}\} \mid \mathrm{k} \geq 0\}\cup\{\overline{\mathbf{B}}\}\cup\{\emptyset\}$ and it represents the segmentation of the strings of interest of a set of character arrays.

- $\overline{\mathbf{M}}_{\mathrm{ns}}$ corresponds to $\{\{\overline{\mathbf{B}} \times \overline{\mathbf{C}}\} \times \{\overline{\mathbf{B}} \times \overline{\mathbf{C}} \times \{\sqcup, ?\}\}^{\mathrm{k}} \times \{\overline{\mathbf{B}} \times \{\sqcup, ?\}\} \mid \mathrm{k} \geq 0\} \cup \{\emptyset\}$ and it represents the segmentation of the content of character arrays after their string of interests, or character arrays that do not contain the null terminating character.

- $\bot_{\overline{\mathbf{M}}}$, $\top_{\overline{\mathbf{M}}}$ are special elements denoting the bottom/top element of $\overline{\mathbf{M}}$.

The elements in $\overline{\mathbf{M}}$ are split segmentation abstract predicates of the form $\overline{\mathbf{m}} = (\mathrm{s}, \mathrm{ns})$. For instance, when $\overline{\mathbf{m}}$ is equal to $(\overline{\mathbf{b}}_1, \emptyset)$, it abstracts concrete character array values of length 1 and containing a `null` string of interest (cf. Section 3.4.1). On the other hand, when $\overline{\mathbf{m}}$ is equal to $(\overline{\mathbf{b}}_1, \overline{\mathbf{b}}_2\overline{\mathbf{p}}_2\overline{\mathbf{b}}_3[?_3] \ldots \overline{\mathbf{b}}_{\mathrm{n}}[?_{\mathrm{n}}])$, it approximates concrete character array values of length greater than or equal to 1 containing a `null` string of interest. In particular:

1. $\overline{\mathbf{b}}_{\mathrm{i}} \in \overline{\mathbf{B}}$ denotes the segment bounds, chosen in abstract domain $\overline{\mathbf{B}}$, such that $\mathrm{i} \in [1, \mathrm{n}]$ and $\mathrm{n} > 1$. A segment bound approximates a set of indexes (i.e., positive integers $\mathbb{Z}^+$), but contrary to what defined for the FunArray abstraction, the choice of $\overline{\mathbf{B}}$ is let free.

   For the sake of readability, we apply arithmetic operators on $\overline{\mathbf{b}}_{\mathrm{i}}$ directly. For instance, $\overline{\mathbf{b}} \oplus_{\overline{\mathbf{B}}} 1$ should be read as $\alpha_{\overline{\mathbf{B}}}(\{\mathrm{i} + 1 \mid \mathrm{i} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}})\})$ or $\overline{\mathbf{b}}_1 \oplus_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_2$ as $\alpha_{\overline{\mathbf{B}}}(\{\mathrm{i}_1 + \mathrm{i}_2 \mid \mathrm{i}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1) \wedge \mathrm{i}_2 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_2)\})$, where $\alpha_{\overline{\mathbf{B}}}$ and $\gamma_{\overline{\mathbf{B}}}$ are respectively the abstraction and concretization functions over the bounds abstract domain.

   Please note that $\overline{\mathbf{b}}_1$ and $\overline{\mathbf{b}}_{\mathrm{n}}$ respectively represent the segmentation lower and upper bound and in the case in which $\overline{\mathbf{m}}$ corresponds to the split segmentation $(\overline{\mathbf{b}}_1\overline{\mathbf{p}}_1\overline{\mathbf{b}}_2[?_2] \ldots \overline{\mathbf{b}}_{\mathrm{n}-1}[?_{\mathrm{n}-1}], \emptyset)$ the segmentation upper bound is hidden, due to a representative choice, and equal to $\overline{\mathbf{b}}_{\mathrm{n}-1} \oplus_{\overline{\mathbf{B}}} 1$. Moreover, in a segmentation $\ldots \overline{\mathbf{b}}_{\mathrm{i}}[?_{\mathrm{i}}]\overline{\mathbf{p}}_{\mathrm{i}+1}\overline{\mathbf{b}}_{\mathrm{i}+1}[?_{\mathrm{i}+1}] \ldots$ we always assume that $\mathsf{min}(\gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{\mathrm{i}+1})) > \mathsf{max}(\gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{\mathrm{i}}))$.

2. $\overline{\mathbf{p}}_{\mathrm{i}} \in \overline{\mathbf{C}}$ are abstract predicates, chosen in an abstract domain $\overline{\mathbf{C}}$, denoting possible values of pairs (index, character array element value) in a segment, for relational abstraction, character array elements otherwise.

3. the question mark ?, if present, indicates that the preceding segment might be empty, while ␣ indicates a non-empty segment and, as for [57], non-empty segments are not marked.

Example 3.10. Consider the split segmentation abstract predicate $\overline{\mathbf{m}} = ([0,0]\ \texttt{'a'}\ [2,5], \emptyset)$ where $\overline{\mathbf{C}}$ is the constant propagation domain for characters and $\overline{\mathbf{B}}$ the interval domain. $\overline{\mathbf{m}}$ approximates character arrays certainly containing a string of interest which is actually a sequence of `'a'`, whose length goes from 2 to 5, followed by a null character, e.g., `"aa\0"` and `"aaaaa\0"`.

In the rest of the chapter we will refer to the s and to the ns parameters of a given split segmentation abstract predicate $\overline{\mathbf{m}}$ by $\overline{\mathbf{m}}$.s and $\overline{\mathbf{m}}$.ns respectively.

M-String, like FunArray, is equipped with partial order $\sqsubseteq_{\overline{\mathbf{M}}}$, join $\sqcup_{\overline{\mathbf{M}}}$, meet $\sqcap_{\overline{\mathbf{M}}}$, widening $\nabla_{\overline{\mathbf{M}}}$ and narrowing $\Delta_{\overline{\mathbf{M}}}$ operators (cf. Section 3.2.2). We highlight the fact that the choice of $\overline{\mathbf{B}}$ is let free, so the segmentation unification algorithm presented in [57] needs to be modified accordingly, while preserving its original requirements. The unify procedure behaves as follows: given $\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2 \in \overline{\mathbf{M}}$, $\mathsf{unify}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2)$ results into the pair $\mathsf{unify}(\overline{\mathbf{m}}_1.\mathrm{s}, \overline{\mathbf{m}}_2.\mathrm{s})$ and $\mathsf{unify}(\overline{\mathbf{m}}_1.\mathrm{ns}, \overline{\mathbf{m}}_2.\mathrm{ns})$, where $\overline{\mathbf{m}}_1.\mathrm{s}$ and $\overline{\mathbf{m}}_2.\mathrm{s}$ (resp. $\overline{\mathbf{m}}_1.\mathrm{ns}$ and $\overline{\mathbf{m}}_2.\mathrm{ns}$) are compatible, leading to two abstract predicates $(\overline{\mathbf{m}}_1^{'}.\mathrm{s}, \overline{\mathbf{m}}_1^{'}.\mathrm{ns})$ and $(\overline{\mathbf{m}}_2^{'}.\mathrm{s}, \overline{\mathbf{m}}_2^{'}.\mathrm{ns})$, respectively. Given two split segmentations $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$, let $\mathrm{l}_{\overline{\mathbf{m}}_1.\mathrm{s}}$ and $\mathrm{h}_{\overline{\mathbf{m}}_1.\mathrm{s}}$ (resp. $\mathrm{l}_{\overline{\mathbf{m}}_2.\mathrm{s}}$ and $\mathrm{h}_{\overline{\mathbf{m}}_2.\mathrm{s}}$) denote the lower and upper bounds of $\overline{\mathbf{m}}_1.\mathrm{s}$ (resp. $\overline{\mathbf{m}}_2.\mathrm{s}$). $\overline{\mathbf{m}}_1.\mathrm{s}$ and $\overline{\mathbf{m}}_2.\mathrm{s}$ are compatible if $\mathrm{l}_{\overline{\mathbf{m}}_1.\mathrm{s}} \sqcap_{\overline{\mathbf{B}}} \mathrm{l}_{\overline{\mathbf{m}}_2.\mathrm{s}} \neq \bot_{\overline{\mathbf{B}}}$ and $\mathrm{h}_{\overline{\mathbf{m}}_1.\mathrm{s}} \sqcap_{\overline{\mathbf{B}}} \mathrm{h}_{\overline{\mathbf{m}}_2.\mathrm{s}} \neq \bot_{\overline{\mathbf{B}}}$. The same apply to $\overline{\mathbf{m}}_1.\mathrm{ns}$ and $\overline{\mathbf{m}}_2.\mathrm{ns}$. Definitions 3.6 and 3.7 present how the join and the meet operators over $\overline{\mathbf{M}}$ are computed. The widening and narrowing can be easily derived.

Example 3.11. Consider the following segmentations: $\overline{\mathbf{m}}_1 = ([0,0]\ \texttt{odd}\ [2,4]\ \texttt{even}\ [7,7], \emptyset)$ and $\overline{\mathbf{m}}_2 = ([0,0]\ \texttt{odd}\ [1,2]\ \top_{\mathrm{parity}}\ [3,6]\ \texttt{even}\ [7,7], \emptyset)$. Their unification leads to the abstract elements $\overline{\mathbf{m}}_1^{'} = ([0,0]\ \texttt{odd}\ [2,4]\ \texttt{even}\ [7,7], \emptyset)$ and $\overline{\mathbf{m}}_2^{'} = ([0,0]\ \texttt{odd}\ [2,2]\ \top_{\mathrm{parity}}\ [7,7], \emptyset)$. Observe that the unify yields to a pair of segmentations with the same number of segments and that is not always optimal.

⬡

Definition 3.6 (M-String join)**.** $\sqcup_{\overline{\mathbf{M}}}$ represents the join operator that defines a minimal upper bound between two abstract elements. Let $\mathsf{unify}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2) = \overline{\mathbf{m}}'_1, \overline{\mathbf{m}}'_2$, then $\overline{\mathbf{m}}_1^{'} \sqcup_{\overline{\mathbf{M}}} \overline{\mathbf{m}}_2^{'} = (\overline{\mathbf{m}}_1^{'}.\mathrm{s} \sqcup_{\overline{\mathbf{M}}} \overline{\mathbf{m}}_2^{'}.\mathrm{s}, \overline{\mathbf{m}}_1^{'}.\mathrm{ns} \sqcup_{\overline{\mathbf{M}}} \overline{\mathbf{m}}_2^{'}.\mathrm{ns})$ such that:

- $\overline{\mathbf{m}}_1^{'}.\mathrm{s} \sqcup_{\overline{\mathbf{M}}} \overline{\mathbf{m}}_2^{'}.\mathrm{s} = \overline{\mathbf{b}}_1^1 \sqcup_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_1^2\ \overline{\mathbf{p}}_1^1 \sqcup_{\overline{\mathbf{C}}} \overline{\mathbf{p}}_1^2\ \ldots\ \overline{\mathbf{b}}_{\mathrm{k}}^1 \sqcup_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_{\mathrm{k}}^2\ [?_{\mathrm{k}}^1]\ \curlyvee\ [?_{\mathrm{k}}^2]$
- $\overline{\mathbf{m}}_1^{'}.\mathrm{ns} \sqcup_{\overline{\mathbf{M}}} \overline{\mathbf{m}}_2^{'}.\mathrm{ns} = \overline{\mathbf{b}}_{\mathrm{k+1}}^1 \sqcup_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_{\mathrm{k+1}}^2\ \overline{\mathbf{p}}_{\mathrm{k+1}}^1 \sqcup_{\overline{\mathbf{C}}} \overline{\mathbf{p}}_{\mathrm{k+1}}^2\ \ldots\ \overline{\mathbf{b}}_{\mathrm{n}}^1 \sqcup_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_{\mathrm{n}}^2\ [?_{\mathrm{n}}^1]\ \curlyvee\ [?_{\mathrm{n}}^2]$

if $\overline{\mathbf{m}}_1^{'}.\mathrm{s}$ and $\overline{\mathbf{m}}_2^{'}.\mathrm{s}$ (resp. $\overline{\mathbf{m}}_1^{'}.\mathrm{ns}$ and $\overline{\mathbf{m}}_2^{'}.\mathrm{ns}$) are compatible; $\top_{\overline{\mathbf{M}}}$ otherwise.

Please note that $\sqcup_{\overline{\mathbf{B}}}$, $\sqcup_{\overline{\mathbf{C}}}$ and $\curlyvee$ denote the join operator of $\overline{\mathbf{B}}$, $\overline{\mathbf{C}}$ and $\{\text{␣}, ?\}$, respectively. In particular, $\text{␣} \curlyvee \text{␣} = \text{␣}$ and $\text{␣} \curlyvee ? = ? \curlyvee \text{␣} = ? \curlyvee ? = ?$.

△

Definition 3.7 (M-String meet)**.** $\sqcap_{\overline{\mathbf{M}}}$ represents the meet operator that defines a maximal lower bound between two abstract elements. Let $\mathsf{unify}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2) = \overline{\mathbf{m}}'_1, \overline{\mathbf{m}}'_2$, then $\overline{\mathbf{m}}'_1 \sqcap_{\overline{\mathbf{M}}} \overline{\mathbf{m}}'_2 = (\overline{\mathbf{m}}'_1.\mathrm{s} \sqcap_{\overline{\mathbf{M}}} \overline{\mathbf{m}}'_2.\mathrm{s}, \overline{\mathbf{m}}'_1.\mathrm{ns} \sqcap_{\overline{\mathbf{M}}} \overline{\mathbf{m}}'_2.\mathrm{ns})$ such that:

- $\overline{\mathbf{m}}'_1.\mathrm{s} \sqcap_{\overline{\mathbf{M}}} \overline{\mathbf{m}}'_2.\mathrm{s} = \overline{\mathbf{b}}^1_1 \sqcap_{\overline{\mathbf{B}}} \overline{\mathbf{b}}^2_1\ \overline{\mathbf{p}}^1_1 \sqcap_{\overline{\mathbf{C}}} \overline{\mathbf{p}}^2_1\ \ldots\ \overline{\mathbf{b}}^1_k \sqcap_{\overline{\mathbf{B}}} \overline{\mathbf{b}}^2_k\ [?^1_k] \curlywedge [?^2_k]$
- $\overline{\mathbf{m}}'_1.\mathrm{ns} \sqcap_{\overline{\mathbf{M}}} \overline{\mathbf{m}}'_2.\mathrm{ns} = \overline{\mathbf{b}}^1_{k+1} \sqcap_{\overline{\mathbf{B}}} \overline{\mathbf{b}}^2_{k+1} \overline{\mathbf{p}}^1_{k+1} \sqcap_{\overline{\mathbf{C}}} \overline{\mathbf{p}}^2_{k+1}\ \ldots\ \overline{\mathbf{b}}^1_n \sqcap_{\overline{\mathbf{B}}} \overline{\mathbf{b}}^2_n [?^1_n] \curlywedge [?^2_n]$

if $\overline{\mathbf{m}}'_1.\mathrm{s}$ and $\overline{\mathbf{m}}'_2.\mathrm{s}$ (resp. $\overline{\mathbf{m}}'_1.\mathrm{ns}$ and $\overline{\mathbf{m}}'_2.\mathrm{ns}$) are compatible; $\bot_{\overline{\mathbf{M}}}$ otherwise.

Please note that $\sqcap_{\overline{\mathbf{B}}}$, $\sqcap_{\overline{\mathbf{C}}}$ and $\curlywedge$ denote the meet operator of $\overline{\mathbf{B}}$, $\overline{\mathbf{C}}$ and $\{\text{␣}, ?\}$, respectively. In particular, $\text{␣} \curlywedge \text{␣} = \text{␣} \curlywedge ? = ? \curlywedge \text{␣} = \text{␣}$ and $? \curlywedge ? = ?$.

△

Concretization

The concretization function on the M-String abstract domain $\gamma_{\overline{\mathbf{M}}}$ maps an abstract element to a set of concrete character array values as follows: $\gamma_{\overline{\mathbf{M}}}(\bot_{\overline{\mathbf{M}}}) = \emptyset$, otherwise $\gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})$ is the set of all possible character array values represented by a split segmentation abstract predicate $\overline{\mathbf{m}}$.

Formally, we firstly define the concretization function of a generic segment $(\overline{\mathbf{b}}\overline{\mathbf{p}}\overline{\mathbf{b}}'[?])$ (regardless of what part of the split it is part of) $\gamma^*_{\overline{\mathbf{M}}}$, following [57], which corresponds to the set of character array values whose elements in the segment $[\overline{\mathbf{b}}, \overline{\mathbf{b}}'[?])$ satisfy the predicate $\overline{\mathbf{p}}$.

$$\begin{aligned}\gamma^*_{\overline{\mathbf{M}}}(\overline{\mathbf{b}}\overline{\mathbf{p}}\overline{\mathbf{b}}'[?])\overline{\rho} = \ \{(\rho, \mathrm{l}, \mathrm{h}, \mathrm{M}, \mathrm{N}) \mid \rho \in \gamma_{\overline{\mathbf{R}}}(\overline{\rho}) \wedge \forall \mathrm{b}, \mathrm{b}' : \mathrm{b} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}), \mathrm{b}' \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}') \wedge \\ [\![\mathrm{l}]\!]\rho \leq \mathrm{b} \leq \mathrm{b}' \leq [\![\mathrm{h}]\!]\rho \wedge \forall \mathrm{i} \in [\mathrm{b}, \mathrm{b}') : \mathrm{M}(\mathrm{i}) \in \gamma_{\overline{\mathbf{C}}}(\overline{\mathbf{p}}) \wedge \\ \mathrm{N} = \{\mathrm{i} \mid \mathrm{M}(\mathrm{i}) = (\mathrm{i}, \texttt{'\textbackslash 0'})\}\}\end{aligned}$$

where $\gamma_{\overline{\mathbf{R}}} \in \overline{\mathbf{R}} \to \mathcal{P}(\mathbf{R}_{\mathrm{v}})$ is the concretization function for the variable environment abstract domain, $\gamma_{\overline{\mathbf{B}}} \in \overline{\mathbf{B}} \to \mathcal{P}(\mathbb{Z}^+)$ is the concretization function for the segment bounds abstract domain, and $\gamma_{\overline{\mathbf{C}}} \in \overline{\mathbf{C}} \to \mathcal{P}(\mathbb{Z} \times \mathbf{C})$ is the concretization function for the array characters abstract domain.

We remind that the upper bound of $\overline{\mathrm{m}}.\mathrm{s}$ is not followed by a segment abstract predicate. Let $\overline{\mathbf{b}}$ be the upper bound of $\overline{\mathbf{m}}.\mathrm{s}$ (which may coincide with the lower bound of $\overline{\mathbf{m}}.\mathrm{s}$ in the case in which $\overline{\mathbf{m}}$ approximates characters arrays containing null strings of interest). $\overline{\mathbf{b}}$ is equivalent to the segment $\overline{\mathbf{b}}\overline{\mathbf{p}}\overline{\mathbf{b}}'$ such that $\overline{\mathbf{b}}' = \overline{\mathbf{b}} \oplus_{\overline{\mathbf{B}}} 1$ and $\overline{\mathbf{p}}$ is null.

An abstract element in the M-String domain is a pair of segmentations. Thus, we define the concretization function of the possible $\overline{\mathbf{m}}.\mathrm{s}$ and $\overline{\mathbf{m}}.\mathrm{ns}$ belonging to a character array

abstract predicate $\overline{\mathbf{m}}$, i.e., $\gamma^{\star}_{\overline{\mathbf{M}}} \in \overline{\mathbf{M}} \to \overline{\mathbf{R}} \to \mathcal{P}(\mathbf{M})$. Let $\underset{\mathbf{M}}{+}$ denote the concatenation of several concrete values.

$$
\begin{aligned}
\gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}.\mathrm{s})\overline{\rho} = \; & \{(\rho, \mathrm{l}, \mathrm{h}, \mathrm{M}, \mathrm{N}) \in \underset{\substack{\mathbf{M} \\ \mathrm{i}=1}}{\overset{\mathrm{k}}{+}} \gamma^{*}_{\overline{\mathbf{M}}}(\overline{\mathbf{b}}_{\mathrm{i}}\overline{\mathbf{p}}_{\mathrm{i}}\overline{\mathbf{b}}_{\mathrm{i}+1}[?_{\mathrm{i}+1}])\overline{\rho} \mid \forall \mathrm{b}_1, \mathrm{b}_{\mathrm{k}} : \; \mathrm{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1), \\
& \quad \mathrm{b}_{\mathrm{k}} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{\mathrm{k}}) \wedge \mathrm{b}_1 = [\![\mathrm{l}]\!]\rho \wedge \mathrm{b}_{\mathrm{k}} + 1 \leq [\![\mathrm{h}]\!]\rho\} \\
& \quad \text{if } \overline{\mathbf{m}}.\mathrm{s} = \overline{\mathbf{b}}_1\overline{\mathbf{p}}_1\overline{\mathbf{b}}_2[?_2] \ldots \overline{\mathbf{b}}_{\mathrm{k}-1}[?_{\mathrm{k}-1}]\overline{\mathbf{p}}_{\mathrm{k}-1}\overline{\mathbf{b}}_{\mathrm{k}}[?_{\mathrm{k}}] \\
= \; & \gamma^{*}_{\overline{\mathbf{M}}}(\overline{\mathbf{b}}_1)\overline{\rho} \\
& \quad \text{if } \overline{\mathbf{m}}.\mathrm{s} = \overline{\mathbf{b}}_1 \\
= \; & \emptyset \\
& \quad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}.\mathrm{ns})\overline{\rho} = \; & \{(\rho, \mathrm{l}, \mathrm{h}, \mathrm{M}, \mathrm{N}) \in \underset{\substack{\mathbf{M} \\ \mathrm{i}=1}}{\overset{\mathrm{n}-1}{+}} \gamma^{*}_{\overline{\mathbf{M}}}(\overline{\mathbf{b}}_{\mathrm{i}}\overline{\mathbf{p}}_{\mathrm{i}}\overline{\mathbf{b}}_{\mathrm{i}+1}[?_{\mathrm{i}+1}])\overline{\rho} \mid \forall \mathrm{b}_1, \mathrm{b}_{\mathrm{n}} : \\
& \quad \mathrm{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1), \mathrm{b}_{\mathrm{n}} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{\mathrm{n}}) \; \wedge \mathrm{b}_1 = [\![\mathrm{l}]\!]\rho \wedge \mathrm{b}_{\mathrm{n}} = [\![\mathrm{h}]\!]\rho\} \\
& \quad \text{if } \overline{\mathbf{m}}.\mathrm{ns} = \overline{\mathbf{b}}_1\overline{\mathbf{p}}_1\overline{\mathbf{b}}_2[?_2] \ldots \overline{\mathbf{b}}_{\mathrm{n}}[?_{\mathrm{n}}] \\
= \; & \{(\rho, \mathrm{l}, \mathrm{h}, \mathrm{M}, \mathrm{N}) \in \underset{\substack{\mathbf{M} \\ \mathrm{i}=\mathrm{k}+1}}{\overset{\mathrm{n}-1}{+}} \gamma^{*}_{\overline{\mathbf{M}}}(\overline{\mathbf{b}}_{\mathrm{i}}\overline{\mathbf{p}}_{\mathrm{i}}\overline{\mathbf{b}}_{\mathrm{i}+1}[?_{\mathrm{i}+1}])\overline{\rho} \mid \forall \mathrm{b}_{\mathrm{k}+1}, \mathrm{b}_{\mathrm{n}} : \\
& \quad \mathrm{b}_{\mathrm{k}+1} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{\mathrm{k}+1}), \mathrm{b}_{\mathrm{n}} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{\mathrm{n}}) \wedge [\![\mathrm{l}]\!]\rho < \mathrm{b}_{\mathrm{k}+1} \wedge \mathrm{b}_{\mathrm{n}} = [\![\mathrm{h}]\!]\rho\} \\
& \quad \text{if } \overline{\mathbf{m}}.\mathrm{ns} = \overline{\mathbf{b}}_{\mathrm{k}+1}\overline{\mathbf{p}}_{\mathrm{k}+1}\overline{\mathbf{b}}_{\mathrm{k}+2}[?_{\mathrm{k}+2}] \ldots \overline{\mathbf{b}}_{\mathrm{n}}[?_{\mathrm{n}}] \\
= \; & \emptyset \\
& \quad \text{otherwise}
\end{aligned}
$$

Finally, the concretization function of a split segmentation abstract predicate $\overline{\mathbf{m}}$ is as follows:

$$
\begin{aligned}
\gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})\overline{\rho} \triangleq \{(\rho, \mathrm{l}, \mathrm{h}, \mathrm{M}, \mathrm{N}) \in \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}.\mathrm{s})\overline{\rho} +_{\mathbf{M}} \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}.\mathrm{ns})\overline{\rho} \mid \forall \mathrm{b}_1, \mathrm{b}_{\mathrm{n}} : \; \mathrm{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1), \\
\mathrm{b}_{\mathrm{n}} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{\mathrm{n}}) \; \wedge \mathrm{b}_1 = [\![\mathrm{l}]\!]\rho \wedge \mathrm{b}_{\mathrm{n}} = [\![\mathrm{h}]\!]\rho\}
\end{aligned}
$$

where $+_{\mathbf{M}}$ returns all the possible concatenations between a concrete array value taken from $\gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}.\mathrm{s})$, and a concrete array value taken from $\gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}.\mathrm{ns})$.

Definition 3.8 (Invalid segment)**.** Given a generic segment $\overline{\mathbf{b}}\overline{\mathbf{p}}\overline{\mathbf{b}}'[?]$, it is considered invalid if its segment abstract predicate $\overline{\mathbf{p}}$ is equal to $\bot_{\overline{\mathbf{C}}}$ and its upper bound $\overline{\mathbf{b}}'$ is not followed by a question mark.

△

Theorem 3.1. Let $\overline{\mathrm{X}} \subseteq \overline{\mathbf{M}}$ such that all elements in $\overline{\mathrm{X}}$ are compatible and their meet does not result in split segmentation abstract predicates which contain invalid abstract elements. The following inference chain holds:

$$
\begin{aligned}
&\gamma_{\overline{\mathbf{M}}}\left(\bigsqcap_{\overline{\mathbf{m}}\in\overline{\mathrm{X}}}^{\overline{\mathbf{M}}}\overline{\mathbf{m}}\right)\\
&\quad= \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}') \qquad \text{where } \overline{\mathbf{m}}' \text{ is the result of the meet operation over } \overline{\mathrm{X}} \text{ (cf. Definition 3.7)}\\
&\quad= \{(\rho, \mathrm{l}, \mathrm{h}, \mathrm{M}, \mathrm{N}) \in \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}'.\mathrm{s})\overline{\rho} +_{\mathbf{M}} \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}'.\mathrm{ns})\overline{\rho} \mid \forall \mathrm{b}_1,\mathrm{b}_\mathrm{n} : \mathrm{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1),\\
&\qquad \mathrm{b}_\mathrm{n} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_\mathrm{n}) \wedge \mathrm{b}_1 = [\![\mathrm{l}]\!]\rho \wedge \mathrm{b}_\mathrm{n} = [\![\mathrm{h}]\!]\rho\} \qquad \text{by definition of } \gamma_{\overline{\mathbf{M}}}\\
&\quad= \bigcap_{\overline{\mathbf{m}}\in\overline{\mathrm{X}}}^{\mathbf{M}}\{(\rho, \mathrm{l}, \mathrm{h}, \mathrm{M}, \mathrm{N}) \in \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}.\mathrm{s})\overline{\rho} +_{\mathbf{M}} \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}.\mathrm{ns})\overline{\rho} \mid \forall \mathrm{b}_1,\mathrm{b}_\mathrm{n} :\ \mathrm{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1),\\
&\qquad \mathrm{b}_\mathrm{n} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_\mathrm{n}) \wedge \mathrm{b}_1 = [\![\mathrm{l}]\!]\rho \wedge \mathrm{b}_\mathrm{n} = [\![\mathrm{h}]\!]\rho\}\\
&\quad= \bigcap_{\overline{\mathbf{m}}\in\overline{\mathrm{X}}}^{\mathbf{M}}\gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}) \qquad \text{by definition of } \gamma_{\overline{\mathbf{M}}}
\end{aligned}
$$

□

Observe that if the hypotheses of Theorem 3.1 are not satisfied, i.e., if either the abstract predicates in $\overline{\mathrm{X}}$ are not compatible or their meet leads to invalid segmentations, then $\gamma_{\overline{\mathbf{M}}}\left(\bigsqcap_{\overline{\mathbf{m}}\in\overline{\mathrm{X}}}^{\overline{\mathbf{M}}}\overline{\mathbf{m}}\right) = \gamma_{\overline{\mathbf{M}}}(\bot_{\overline{\mathbf{M}}}) = \emptyset$, and $\bigcap_{\overline{\mathbf{m}}\in\overline{\mathrm{X}}}^{\mathbf{M}}\gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}) = \emptyset$.

In the implementation we will make use of two functions lift and lower that relate single strings to their abstraction in M-String.

Definition 3.9 (Lift)**.** Let $\mu(\mathtt{m}) \subseteq \mathbf{M}$ be a concrete character array value. Given the abstraction function $\alpha_{\overline{\mathbf{M}}}$ on M-String, we define the lift operation of X as follows:

$$\mathsf{lift}(\mu(\mathtt{m})) = \alpha_{\overline{\mathbf{M}}}(\mu(\mathtt{m})).$$

△

Definition 3.10 (Lower)**.** Let $\overline{\mathbf{m}}$ denote $\mathsf{lift}(\mu(\mathtt{m}))$ (cf. Definition 3.9). Given the concretization function $\gamma_{\overline{\mathbf{M}}}$ on M-String, we define the lower operation of $\overline{\mathbf{m}}$ as follows:

$$\mathsf{lower}(\overline{\mathbf{m}}) = \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})$$

△

Abstraction

Let X be a set of concrete character array values. The abstraction function on the M-String abstract domain $\alpha_{\overline{\mathbf{M}}}$ maps X to $\bot_{\overline{\mathbf{M}}}$ in the case in which X is empty, otherwise to an element of M-String that optimally over-approximates values in X.

3.5.2 Abstract Semantics

Let us now formalize the abstract semantics of the concrete operations defined in Section 3.4.3, over the M-String domain. In doing so, we will take advantage of the auxiliary function $\mathsf{minLen}_{\overline{\mathbf{M}}}$ which computes the minimum length of an element $\overline{\mathbf{m}} \in \overline{\mathbf{M}}$, as the upper bound of a split segmentation is possibly followed by a question mark.

Definition 3.11 ($\overline{\mathbf{m}}$ minimum length)**.** Let $\overline{\mathbf{m}} \in \overline{\mathbf{M}}$ different from $\bot_{\overline{\mathbf{M}}}$ and let $\mathrm{l}_{\overline{\mathbf{m}}}, \mathrm{h}_{\overline{\mathbf{m}}} \in \overline{\mathbf{B}}$ denote the lower and the upper bound of $\overline{\mathbf{m}}$, respectively. We define the minimum length of a split segmentation abstract predicate $\overline{\mathbf{m}}$, denoted by $\mathsf{minLen}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})$, as follows:

$$\mathsf{minLen}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}) = \begin{cases} \overline{\mathbf{b}}_{\mathrm{k}} \ominus_{\overline{\mathbf{B}}} \mathrm{l}_{\overline{\mathbf{m}}} & \text{if } \overline{\mathbf{m}}.\mathrm{ns} \neq \emptyset \wedge \mathrm{h}_{\overline{\mathbf{m}}} \text{ is followed by } ? \wedge \exists \mathrm{k} \in \overline{\mathbf{m}}.\mathrm{ns} : \\ & \qquad \mathrm{k} = \mathsf{max}(\{\mathrm{i} \in \overline{\mathbf{m}}.\mathrm{ns} \mid \overline{\mathbf{b}}_{\mathrm{i}} \text{ is not followed by } ?\}) \\ \mathrm{h}_{\overline{\mathbf{m}}} \ominus_{\overline{\mathbf{B}}} \mathrm{l}_{\overline{\mathbf{m}}} & \text{otherwise} \end{cases}$$

△

Please note that in the second case of Definition 3.11, the minimum length of a split segmentation corresponds to its length, denoted by $\mathsf{len}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})$. The $\mathsf{len}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})$ operation can be also applied over the parameters of $\overline{\mathbf{m}}$ themselves, when they are different from the emptyset and their upper bound is not question marked, which is always the case with $\overline{\mathbf{m}}$.s.

Example 3.12. Consider the split segmentation abstract predicate $\overline{\mathbf{m}} = ([0,0]\ \mathtt{'a'}\ [2,5], [3,6]\ \mathtt{'b'}\ [7,7]\ \mathtt{'c'}\ [8,8]?)$ where $\overline{\mathbf{C}}$ is the constant propagation domain for characters and $\overline{\mathbf{B}}$ the interval domain. The minimum length of $\overline{\mathbf{m}}$ is given by $\mathsf{minLen}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}) = [7,7] \ominus_{\overline{\mathbf{B}}} [0,0] = [7,7]$ as its upper bound is followed by a question mark. Logically speaking, the maximum length of $\overline{\mathbf{m}}$ is $[8,8] \ominus_{\overline{\mathbf{B}}} [0,0] = [8,8]$. The length of $\overline{\mathbf{m}}$.s is given by $\mathsf{len}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}.\mathrm{s}) = [2,5] \ominus_{\overline{\mathbf{B}}} [0,0] = [2,5]$.

⬡

Abstract Array Access

The semantic operator $\mathfrak{A}_{\overline{\mathbf{M}}}$ is the abstract counterpart of $\mathfrak{A}$ (cf. Section 3.4.3). In particular, $\mathsf{access}_{\bar{\mathrm{j}}}(\overline{\mathbf{m}})$ returns, if $\bar{\mathrm{j}}$ is valid for $\overline{\mathbf{m}}$ (i.e., there exist, and it is unique, a segment bounds interval $[\overline{\mathbf{b}}_{\mathrm{i}}[?_{\mathrm{i}}], \overline{\mathbf{b}}_{\mathrm{i}+1})$ in $\overline{\mathbf{m}}$ to which $\bar{\mathrm{j}}$ belongs), the segment abstract predicate $\overline{\mathbf{p}}_{\mathrm{i}}$; otherwise it returns $\top_{\overline{\mathbf{C}}}$. Formally,

$$\mathfrak{A}_{\overline{\mathbf{M}}}[\![\mathsf{access}_{\bar{\mathrm{j}}}]\!](\overline{\mathbf{m}}) = \begin{cases} \overline{\mathbf{p}}_{\mathrm{i}} & \text{if } \exists! \mathrm{i} \in \overline{\mathbf{m}} : \bar{\mathrm{j}} \in [\overline{\mathbf{b}}_{\mathrm{i}}[?^{\mathrm{i}}], \overline{\mathbf{b}}_{\mathrm{i}+1}) \\ \top_{\overline{\mathbf{C}}} & \text{otherwise} \end{cases}$$

where $\bar{\mathrm{j}} \in [\overline{\mathbf{b}}_{\mathrm{i}}[?^{\mathrm{i}}], \overline{\mathbf{b}}_{\mathrm{i}+1})$ if $\forall \mathrm{j} \in \gamma_{\overline{\mathbf{B}}}(\bar{\mathrm{j}}) : \forall [\mathrm{l}, \mathrm{u}) \in \{[\mathrm{l}, \mathrm{u}) \mid \mathrm{l} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{\mathrm{i}}) \wedge \mathrm{u} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{\mathrm{i}+1})\} : \mathrm{j} \in [\mathrm{l}, \mathrm{u})$.

Abstract String Concatenation

The semantic operator $\mathfrak{M}_{\overline{\mathbf{M}}}$ is the abstract counterpart of $\mathfrak{M}$. When applied to $\texttt{strcat}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2)$, it returns $\overline{\mathbf{m}}_1^{'}$ that is $\overline{\mathbf{m}}_1$ into which $\overline{\mathbf{m}}_2$.s has been embedded starting from the upper bound of $\overline{\mathbf{m}}$.s, if both the input split segmentations approximate character arrays which contain a well-formed string and the condition on the size of the destination split segmentation is fulfilled; otherwise it returns $\top_{\overline{\mathbf{M}}}$. Formally,

$$\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{strcat}]\!](\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2) = \begin{cases} \overline{\mathbf{m}}_1^{'} & \text{if } \overline{\mathbf{m}}_1.\mathrm{s} \neq \emptyset \neq \overline{\mathbf{m}}_2.\mathrm{s} \ \wedge \texttt{size.condition} \text{ is } \texttt{true} \\ \top_{\overline{\mathbf{M}}} & \text{otherwise} \end{cases}$$

The `size.condition` is `true` if $\mathsf{minLen}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_1) \geq_{\overline{\mathbf{B}}} (\mathsf{len}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_1.\mathrm{s}) \oplus_{\overline{\mathbf{B}}} \mathsf{len}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_2.\mathrm{s}) \oplus_{\overline{\mathbf{B}}} 1)$. Let:

- $\overline{\mathbf{m}}_1 = (\overline{\mathbf{b}}_1^1 \overline{\mathbf{p}}_1^1 \overline{\mathbf{b}}_2^1 [?_2^1] \ldots \overline{\mathbf{p}}_{\mathrm{k}-1}^1 \overline{\mathbf{b}}_{\mathrm{k}}^1 [?_{\mathrm{k}}^1], \overline{\mathbf{b}}_{\mathrm{k}+1}^1 \overline{\mathbf{p}}_{\mathrm{k}+1}^1 \overline{\mathbf{b}}_{\mathrm{k}+2}^1 [?_{\mathrm{k}+2}^1] \ldots \overline{\mathbf{b}}_{\mathrm{n}}^1 [?_{\mathrm{n}}^1])$
- $\overline{\mathbf{m}}_2 = (\overline{\mathbf{b}}_1^2 \overline{\mathbf{p}}_1^2 \overline{\mathbf{b}}_2^2 [?_2^2] \ldots \overline{\mathbf{p}}_{\mathrm{k}-1}^2 \overline{\mathbf{b}}_{\mathrm{k}}^2 [?_{\mathrm{k}}^2], \mathrm{ns})$

Then, $\overline{\mathbf{m}}_1^{'}.\mathrm{s} = \overline{\mathbf{b}}_1^1 \overline{\mathbf{p}}_1^1 \overline{\mathbf{b}}_2^1 [?_2^1] \ldots \overline{\mathbf{p}}_{\mathrm{k}-1}^1 \overline{\mathbf{b}}_{\mathrm{k}}^1 [?_{\mathrm{k}}^1] \overline{\mathbf{p}}_1^2 (\overline{\mathbf{b}}_{\mathrm{k}}^1 \oplus_{\overline{\mathbf{B}}} (\overline{\mathbf{b}}_2^2 \ominus_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_1^2))[?_2^2] \ldots \overline{\mathbf{p}}_{\mathrm{k}-1}^2 (\overline{\mathbf{b}}' \oplus_{\overline{\mathbf{B}}} (\overline{\mathbf{b}}_{\mathrm{k}}^2 \ominus_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_{\mathrm{k}-1}^2))[?_2^2]$ such that $\overline{\mathbf{b}}'$ denotes the immediately preceding adapted segment bound. On the other hand, $\overline{\mathbf{m}}_1^{'}$.ns is the result of removing from $\overline{\mathbf{m}}_1$.ns the sub-segmentation that goes from the lower bound of $\overline{\mathbf{m}}_1$.ns to the upper bound of $\overline{\mathbf{m}}_1^{'}$.s included.

Example 3.13. Let $([0,0]\ \texttt{a}*\ [5,7], [6,8]\ \texttt{br}*\ [13,14])$ and $([0,0]\ \texttt{a}*[3,3], \emptyset)$ be two abstract elements in $\overline{\mathbf{M}}$, such that $\overline{\mathbf{B}}$ is the interval domain over array indexes and $\overline{\mathbf{C}}$ is the prefix domain over string values. Precisely, $([0,0]\ \texttt{a}*\ [5,7], [6,8]\ \texttt{br}*\ [13,14])$ approximates all the characters arrays with as string of interest any string starting with the character `'a'` whose length goes from 5 to 7, followed by the null character and any string starting with `''br''` whose length goes from 5 to 8. On the other hand, $([0,0]\ \texttt{a}*\ [3,3], \emptyset)$ abstracts all the array of chars with string of interest equal to a string, of length 3, starting with `a` . Consider now the concatenation between them,

$$\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{strcat}]\!](([0,0]\ \texttt{a}*\ [5,7], [6,8]\ \texttt{br}*\ [13,14]), ([0,0]\ \texttt{a}*\ [3,3], \emptyset))$$

The size condition is satisfied: the minimum length of the destination split segmentation is equal to 13, which is strictly greater than $7 + 3 + 1$, i.e., the maximum length of the destination abstract array plus the maximum length of the source segmentation plus one (the null character). Their concatenation results in the following abstract element:

$$([0,0]\ \texttt{a}*\ [5,7]\ \texttt{a}*\ [8,10], [9,11]\ \texttt{br}*\ [13,14])$$

which is equivalent to $([0,0]\ \texttt{a}*\ [8,10], [9,11]\ \texttt{br}*\ [13,14])$.

⬡

Abstract String Character

The semantic operator $\mathfrak{M}_{\overline{\mathbf{M}}}$, when applied to $\texttt{strchr}_{\overline{\mathrm{v}}}(\overline{\mathbf{m}})$, returns a split segmentation abstract predicate $\overline{\mathbf{s}}$ with the left hand side parameter equal to the suffix segmentation of the input $\overline{\mathbf{m}}.\mathrm{s}$ from the first segment in which $\overline{\mathrm{v}}$ certainly occurs and the right hand side parameter equal to the emptyset, if $\overline{\mathbf{m}}$ approximates character arrays which contain a well-formed string and the character $\overline{\mathrm{v}}$ appears in at least one segment whose bounds are not question marked. Otherwise, if $\overline{\mathbf{m}}$ approximates character arrays which contain a well-formed string of interest and the abstract character $\overline{\mathrm{v}}$ does not occur in $\overline{\mathbf{m}}.\mathrm{s}$, it returns $\bot_{\overline{\mathbf{M}}}$; otherwise it returns $\top_{\overline{\mathbf{M}}}$. Formally,

$$\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{strchr}_{\overline{\mathrm{v}}}]\!](\overline{\mathbf{m}}) = \begin{cases} \overline{\mathbf{s}} & \text{if } \overline{\mathbf{m}}.\mathrm{s} \neq \emptyset \wedge \exists \mathrm{i} \in \overline{\mathbf{m}}.\mathrm{s} : \overline{\mathbf{p}}_{\mathrm{i}} = \overline{\mathrm{v}} \wedge \\ & \qquad \overline{\mathbf{b}}_{\mathrm{i}}, \overline{\mathbf{b}}_{\mathrm{i}+1} \text{ are not followed by ?} \\ \bot_{\overline{\mathbf{M}}} & \text{if } \overline{\mathbf{m}}.\mathrm{s} \neq \emptyset \wedge \nexists \mathrm{i} \in \overline{\mathbf{m}}.\mathrm{s} : \overline{\mathbf{p}}_{\mathrm{i}} = \overline{\mathrm{v}} \\ \top_{\overline{\mathbf{M}}} & \text{otherwise} \end{cases}$$

where $\overline{\mathbf{s}} = (\overline{\mathbf{b}}_{\mathrm{j}}\overline{\mathbf{p}}_{\mathrm{j}}\overline{\mathbf{b}}_{\mathrm{j}+1} \ldots \overline{\mathbf{b}}_{\mathrm{k}}[?_{\mathrm{k}}], \emptyset)$ such that

$$\mathrm{j} = \mathsf{min}(\{\mathrm{i} \in \overline{\mathbf{m}}.\mathrm{s} \mid \overline{\mathbf{p}}_{\mathrm{i}} = \overline{\mathrm{v}} \ \wedge \ \overline{\mathbf{b}}_{\mathrm{i}}, \overline{\mathbf{b}}_{\mathrm{i}+1} \text{ are not question marked}\})$$

Abstract String Compare

The semantic $\mathfrak{P}_{\overline{\mathbf{M}}}$ is the abstract counterpart of $\mathfrak{P}$. In particular, $\texttt{strcmp}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2)$ returns a value $\overline{\mathrm{n}}$ denoting the lexicographic distance between $\overline{\mathbf{m}}_1.\mathrm{s}$ and $\overline{\mathbf{m}}_2.\mathrm{s}$ if both the input split segmentations approximate character arrays which contain a well-formed string and they can be unified; otherwise it returns $\top_{\mathbb{Z}}$.

Notice that if $\overline{\mathrm{n}}$ is negative, this means that the strings of interest approximated by $\overline{\mathbf{m}}_1$ precede those represented by $\overline{\mathbf{m}}_2$ in lexicographic order. Conversely, if $\overline{\mathrm{n}}$ is positive, this means that the strings of interest approximated by $\overline{\mathbf{m}}_1$ follows those represented by $\overline{\mathbf{m}}_2$ in lexicographic order, and if $\overline{\mathrm{n}}$ is equal to zero they are lexicographically equal. Formally,

$$\mathfrak{P}_{\overline{\mathbf{M}}}[\![\texttt{strcmp}]\!](\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2) = \begin{cases} \overline{\mathrm{n}} & \text{if } \overline{\mathbf{m}}_1.\mathrm{s} \neq \emptyset \neq \overline{\mathbf{m}}_2.\mathrm{s} \\ \top_{\mathbb{Z}} & \text{otherwise} \end{cases}$$

where $\overline{\mathrm{n}} = \mathsf{cmp}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2)$ (cf. Algorithm 2).

Algorithm 2 Lexicographic comparison of split segmentation abstract predicates.

Input: two compatible split segmentation abstract predicates $\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2 \in \overline{\mathbf{M}}$.
Output: an integer value $\overline{\mathrm{n}}$.

1: $\overline{\mathrm{n}} = 0$, $\mathrm{i} = 1$
2: $\mathsf{unify}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2) = \overline{\mathbf{m}}_1^{'}, \overline{\mathbf{m}}_2^{'}$
3: **if** $\overline{\mathbf{m}}_1^{'}.\mathrm{s} = \overline{\mathbf{b}}_1^1 \wedge \overline{\mathbf{m}}_2^{'}.\mathrm{s} = \overline{\mathbf{b}}_1^2$ **then**
4: $\quad$ return $\overline{\mathrm{n}}$
5: **else if** $\overline{\mathbf{m}}_1^{'}.\mathrm{s} = \overline{\mathbf{b}}_1^1 \wedge \overline{\mathbf{m}}_2^{'}.\mathrm{s} \neq \overline{\mathbf{b}}_1^2$ **then**
6: $\quad$ $\overline{\mathrm{n}} = \overline{\mathrm{n}} \ominus_{\overline{\mathbf{C}}} \overline{\mathbf{p}}_1^2$
7: $\quad$ return $\overline{\mathrm{n}}$
8: **else if** $\overline{\mathbf{m}}_1^{'}.\mathrm{s} \neq \overline{\mathbf{b}}_1^1 \wedge \overline{\mathbf{m}}_2^{'}.\mathrm{s} = \overline{\mathbf{b}}_1^2$ **then**
9: $\quad$ $\overline{\mathrm{n}} = \overline{\mathbf{p}}_1^1 \ominus_{\overline{\mathbf{C}}} \overline{\mathrm{n}}$
10: $\quad$ return $\overline{\mathrm{n}}$
11: **else**
12: $\quad$ **while** $\mathrm{i} \in \overline{\mathbf{m}}_1^{'}.\mathrm{s} \wedge \mathrm{i} \in \overline{\mathbf{m}}'_2.\mathrm{s}$ **do**
13: $\quad\quad$ $\overline{\mathrm{n}} = \overline{\mathbf{p}}_\mathrm{i}^1 \ominus_{\overline{\mathbf{C}}} \overline{\mathbf{p}}_\mathrm{i}^2$
14: $\quad\quad$ **if** $\overline{\mathrm{n}} \neq 0$ **then**
15: $\quad\quad\quad$ return $\overline{\mathrm{n}}$
16: $\quad\quad$ **else**
17: $\quad\quad\quad$ $\mathrm{i} = \mathrm{i} + 1$
18: return $\overline{\mathrm{n}}$

Abstract String Copy

The semantic $\mathfrak{M}_{\overline{\mathbf{M}}}$, when applied to $\mathtt{strcpy}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2)$, it returns $\overline{\mathbf{m}}_1^{'}$ that is $\overline{\mathbf{m}}_1$ into which $\overline{\mathbf{m}}_2.\mathrm{s}$ has been embedded starting from the lower bound of $\overline{\mathbf{m}}$, if both the input split segmentations approximate character arrays which contain a well-formed string and the condition on the size of the destination split segmentation is fulfilled; otherwise it returns $\top_{\overline{\mathbf{M}}}$. Formally,

$$\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathtt{strcpy}]\!](\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2) = \begin{cases} \overline{\mathbf{m}}_1^{'} & \text{if } \overline{\mathbf{m}}_1.\mathrm{s} \neq \emptyset \neq \overline{\mathbf{m}}_2.\mathrm{s} \ \wedge \mathtt{size.condition} \text{ is } \mathtt{true} \\ \top_{\overline{\mathbf{M}}} & \text{otherwise} \end{cases}$$

The `size.condition` is `true` if $\mathsf{minLen}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_1) \geq_{\overline{\mathbf{B}}} \mathsf{len}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_2.\mathrm{s}) \oplus_{\overline{\mathbf{B}}} 1$. Let:

- $\overline{\mathbf{m}}_1 = (\overline{\mathbf{b}}_1^1 \overline{\mathbf{p}}_1^1 \overline{\mathbf{b}}_2^1 [?_2^1] \ldots \overline{\mathbf{p}}_{\mathrm{k}-1}^1 \overline{\mathbf{b}}_\mathrm{k}^1 [?_\mathrm{k}^1], \overline{\mathbf{b}}_{\mathrm{k}+1}^1 \overline{\mathbf{p}}_{\mathrm{k}+1}^1 \overline{\mathbf{b}}_{\mathrm{k}+2}^1 [?_{\mathrm{k}+2}^1] \ldots \overline{\mathbf{b}}_\mathrm{n}^1 [?_\mathrm{n}^1])$
- $\overline{\mathbf{m}}_2 = (\overline{\mathbf{b}}_1^2 \overline{\mathbf{p}}_1^2 \overline{\mathbf{b}}_2^2 [?_2^2] \ldots \overline{\mathbf{p}}_{\mathrm{k}-1}^2 \overline{\mathbf{b}}_\mathrm{k}^2 [?_\mathrm{k}^2], \mathrm{ns})$

Then, $\overline{\mathbf{m}}_1^{'}.\mathrm{s} = \overline{\mathbf{b}}_1^1 \overline{\mathbf{p}}_1^2 (\overline{\mathbf{b}}_1^1 \oplus_{\overline{\mathbf{B}}} (\overline{\mathbf{b}}_2^2 \ominus_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_1^2))[?_2^2] \ldots \overline{\mathbf{p}}_{\mathrm{k}-1}^2 (\overline{\mathbf{b}}' \oplus_{\overline{\mathbf{B}}} (\overline{\mathbf{b}}_\mathrm{k}^2 \ominus_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_{\mathrm{k}-1}^2))[?_\mathrm{k}^2]$ such that $\overline{\mathbf{b}}'$ denotes the immediately preceding adapted segment bound. On the other hand $\overline{\mathbf{m}}_1^{'}.\mathrm{ns}$ is the sub-segmentation of $\overline{\mathbf{m}}_1$ that goes from the upper bound of $\overline{\mathbf{m}}_1^{'}.\mathrm{s}$ plus one to the upper bound of $\overline{\mathbf{m}}_1$.

Abstract String Length

The semantic $\mathfrak{L}_{\overline{\mathbf{M}}}$ is the abstract counterpart of $\mathfrak{L}$. In particular, `strlen` returns a value $\overline{\mathrm{n}}$, if $\overline{\mathbf{m}}$ approximates character arrays which contain a well-formed string, the upper bound of $\overline{\mathbf{m}}.\mathrm{s}$ is not followed by a question mark and in $\overline{\mathbf{m}}.\mathrm{s}$ do not occur possibly null segment abstract predicates; otherwise it returns $\top_{\overline{\mathbf{B}}}$. Formally,

$$\mathfrak{L}_{\overline{\mathbf{M}}}[\![\texttt{strlen}]\!](\overline{\mathbf{m}}) = \begin{cases} \overline{\mathrm{n}} & \text{if } \overline{\mathbf{m}}.\mathrm{s} = \overline{\mathbf{b}}_1\overline{\mathbf{p}}_1\overline{\mathbf{b}}_2[?_2]\ldots\overline{\mathbf{b}}_\mathrm{k} \wedge \\ & \qquad \nexists \mathrm{i} \in \overline{\mathbf{m}}.\mathrm{s} : \mathsf{isNull}(\overline{\mathbf{p}}_\mathrm{i}) = \texttt{maybe} \\ \top_{\overline{\mathbf{B}}} & \text{otherwise} \end{cases}$$

where $\overline{\mathrm{n}} = \overline{\mathbf{b}}_\mathrm{k} \ominus_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_1$.

Abstract Array Update

The semantic $\mathfrak{M}_{\overline{\mathbf{M}}}$, when applied to $\texttt{update}_{\bar{\mathrm{j}},\overline{\mathrm{v}}}(\overline{\mathbf{m}})$, returns, if $\gamma_{\overline{\mathbf{B}}}(\bar{\mathrm{j}})$ corresponds to the singleton $\{\mathrm{j}\}$ and $\bar{\mathrm{j}}$ is valid for $\overline{\mathbf{m}}$ (i.e., there exists - and it is unique - a segment bounds interval $[\overline{\mathbf{b}}_\mathrm{i}[?_\mathrm{i}], \overline{\mathbf{b}}_{\mathrm{i}+1})$ in $\overline{\mathbf{m}}$ to which $\bar{\mathrm{j}}$ belongs), $\overline{\mathbf{m}}'$ that is $\overline{\mathbf{m}}$ where the segment $\overline{\mathbf{b}}_\mathrm{i}[?_\mathrm{i}]\overline{\mathbf{p}}_\mathrm{i}\overline{\mathbf{b}}_{\mathrm{i}+1}$ is split so that the segment abstract predicate at position $\bar{\mathrm{j}}$ is substituted with $\overline{\mathrm{v}}$; otherwise it returns $\top_{\overline{\mathbf{M}}}$. Formally,

$$\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{update}_{\bar{\mathrm{j}},\overline{\mathrm{v}}}]\!](\overline{\mathbf{m}}) = \begin{cases} \overline{\mathbf{m}}' & \text{if } \gamma_{\overline{\mathbf{B}}}(\bar{\mathrm{j}}) = \{\mathrm{j}\} \wedge \exists! \mathrm{i} \in \overline{\mathbf{m}} : \bar{\mathrm{j}} \in [\overline{\mathbf{b}}_\mathrm{i}[?_\mathrm{i}], \overline{\mathbf{b}}_{\mathrm{i}+1}) \\ \top_{\overline{\mathbf{M}}} & \text{otherwise} \end{cases}$$

3.5.3 Soundness

Theorem 3.2. $\mathfrak{A}_{\overline{\mathbf{M}}}$ is a sound over-approximation of $\mathfrak{A}$. Formally,

$$\gamma_{\overline{\mathbf{C}}}(\mathfrak{A}_{\overline{\mathbf{M}}}[\![\mathrm{stm}]\!](\overline{\mathbf{m}})) \supseteq \{\mathfrak{A}[\![\mathrm{stm}]\!](\mu(\mathtt{m})) : \mu(\mathtt{m}) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})\}$$

Proof.
Consider the unary operator $\texttt{access}_{\bar{\mathrm{j}}}$ and let $\overline{\mathbf{m}}$ be a split segmentation abstract predicate. We have to prove that:

$$\gamma_{\overline{\mathbf{C}}}(\mathfrak{A}_{\overline{\mathbf{M}}}[\![\texttt{access}_{\bar{\mathrm{j}}}]\!](\overline{\mathbf{m}})) \supseteq \{\mathfrak{A}[\![\texttt{access}_\mathrm{j}]\!](\mu(\mathtt{m})) : \mu(\mathtt{m}) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})\}$$

$\texttt{access}_\mathrm{j}$ of $\mu(\mathtt{m})$ returns, by definition of $\mathfrak{A}$, the character array value v that occurs at position j, if j belongs to $[[\![\mathrm{l}_\mathtt{m}]\!]\rho, [\![\mathrm{h}_\mathtt{m}]\!]\rho)$; $\top_{\overline{\mathbf{C}}}$ otherwise. Let $\alpha_{\overline{\mathbf{B}}}(\mathrm{j}) = \bar{\mathrm{j}}$. Then, v belongs to $\gamma_{\overline{\mathbf{C}}}(\mathfrak{A}_{\overline{\mathbf{M}}}[\![\texttt{access}_{\bar{\mathrm{j}}}]\!](\overline{\mathbf{m}}))$ because $\texttt{access}_{\bar{\mathrm{j}}}$ of $\overline{\mathbf{m}}$, by definition of $\mathfrak{A}_{\overline{\mathbf{M}}}$, is equal to the segment abstract predicate $\overline{\mathbf{p}}_\mathrm{i}$, if there exists - and it unique - a segment bounds interval $[\overline{\mathbf{b}}_\mathrm{i}[?_\mathrm{i}], \overline{\mathbf{b}}_{\mathrm{i}+1})$ to which $\bar{\mathrm{j}}$ belongs; $\top_{\overline{\mathbf{C}}}$ otherwise.

□

Theorem 3.3. $\mathfrak{M}_{\overline{\mathbf{M}}}$ is is a sound over-approximation of $\mathfrak{M}$. Formally,

$$\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathrm{stm}]\!](\overline{\mathbf{m}})) \supseteq \{\mathfrak{M}[\![\mathrm{stm}]\!](\mu(\mathtt{m})) : \mu(\mathtt{m}) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})\}$$

Proof.

- Consider the binary operator `strcat` and let $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$ be two split segmentation abstract predicates. We have to prove that:

$$\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathtt{strcat}]\!](\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2))$$
$$\supseteq$$
$$\{\mathfrak{M}[\![\mathtt{strcat}]\!](\mu(\mathtt{m}_1), \mu(\mathtt{m}_2)) : \mu(\mathtt{m}_1) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_1) \wedge \mu(\mathtt{m}_2) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_2)\}$$

  `strcat` of $\mu(\mathtt{m}_1)$ and $\mu(\mathtt{m}_2)$ returns, by definition of $\mathfrak{M}$, $\mu(\mathtt{m}_1)'$ where the first null-terminating memory block of $\mu(\mathtt{m}_2)$ (including the null terminator), i.e., its string of interest, is embedded into $\mu(\mathtt{m}_1)$ starting from the index to which the first null character in $\mu(\mathtt{m}_1)$ occurs, if both $\mu(\mathtt{m}_1)$ and $\mu(\mathtt{m}_2)$ contain a well-formed string and the size condition on the destination character array value is fulfilled; $\top_{\mathbf{M}}$ otherwise. Then, $\mu(\mathtt{m}_1)'$ belongs to $\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathtt{strcat}]\!](\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2))$ because `strcat` of $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$, by definition of $\mathfrak{M}_{\overline{\mathbf{M}}}$, is equal to $\overline{\mathbf{m}}_1^{'}$ that is $\overline{\mathbf{m}}_1$ into which $\overline{\mathbf{m}}_2$.s has been embedded starting from the upper bound of $\overline{\mathbf{m}}_1$.s, if both $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$ approximate character arrays which contain a well-formed string and the size condition on the destination segmentation abstract predicate is fulfilled; $\top_{\overline{\mathbf{M}}}$ otherwise.

- Consider the unary operator $\mathtt{strchr}_{\overline{\mathrm{v}}}$, and let $\overline{\mathbf{m}}$ be a split segmentation abstract predicate. We have to prove that:

$$\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathtt{strchr}_{\overline{\mathrm{v}}}]\!](\overline{\mathbf{m}})) \supseteq \{\mathfrak{M}[\![\mathtt{strchr}_{\mathrm{v}}]\!](\mu(\mathtt{m})) : \mu(\mathtt{m}) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})\}$$

  $\mathtt{strchr}_{\mathrm{v}}$ of $\mu(\mathtt{m})$ returns, by definition of $\mathfrak{M}$, $\mu(\mathtt{s})$ that corresponds to the suffix of the string of interest of $\mu(\mathtt{m})$ starting from the index to which appears the first occurrence of v, if $\mu(\mathtt{m})$ contains a well-formed string and v occurs in $\overline{\mathbf{m}}$; the emptyset (i.e., $\bot_{\mathbf{M}}$), if $\mu(\mathtt{m})$ contains a well-formed string and v does not occur in $\mu(\mathtt{m})$; $\top_{\mathbf{M}}$ otherwise. Let $\alpha_{\overline{\mathbf{C}}}(\mathrm{v}) = \overline{\mathrm{v}}$. Then, $\mu(\mathtt{s})$ belongs to $\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathtt{strchr}_{\overline{\mathrm{v}}}]\!](\overline{\mathbf{m}}))$ because $\mathtt{strchr}_{\overline{\mathrm{v}}}$ of $\overline{\mathbf{m}}$, by definition of $\mathfrak{M}_{\overline{\mathbf{M}}}$, is equal to $\overline{\mathrm{s}}$ that is the split segmentation abstract predicate with $\overline{\mathrm{s}}$.s equal to the sub-segmentation of $\overline{\mathbf{m}}$.s starting from the first segment to which $\overline{\mathrm{v}}$ certainly occurs and $\overline{\mathrm{s}}$.ns equal to the emptyset if $\overline{\mathbf{m}}$ approximates character arrays which contain a well-defined string and $\overline{\mathrm{v}}$ appears in at least one segment whose bounds are not question marked; $\bot_{\overline{\mathbf{M}}}$ if $\overline{\mathbf{m}}$ approximates character arrays which contain a well-formed string and $\overline{\mathrm{v}}$ does not appear in $\overline{\mathbf{m}}$.s; $\top_{\overline{\mathbf{M}}}$ otherwise.

- Consider the binary operator `strcpy` and let $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$ be two split segmentation abstract predicates. We have to prove that:

$$\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathtt{strcpy}]\!](\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2))$$
$$\supseteq$$
$$\{\mathfrak{M}[\![\mathtt{strcpy}]\!](\mu(\mathtt{m}_1), \mu(\mathtt{m}_2)) : \mu(\mathtt{m}_1) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_1) \wedge \mu(\mathtt{m}_2) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_2)\}$$

  `strcpy` of $\mu(\mathtt{m}_1)$ and $\mu(\mathtt{m}_2)$ returns, by definition of $\mathfrak{M}$, $\mu(\mathtt{m}_1)'$ where the first null-terminating memory block of $\mu(\mathtt{m}_2)$ (including the null terminator), i.e., its string of interest, is embedded into $\mu(\mathtt{m}_1)$ starting from the lower bound of $\mu(\mathtt{m}_1)$, if both $\mu(\mathtt{m}_1)$ and $\mu(\mathtt{m}_2)$ contain a well-formed string and the size condition on the destination character array value is fulfilled, $\top_{\mathbf{M}}$ otherwise. Then, $\mu(\mathtt{m}_1)'$ belongs to $\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathtt{strcpy}]\!](\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2))$ because `strcpy` of $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$, by definition of $\mathfrak{M}_{\overline{\mathbf{M}}}$, is equal to $\overline{\mathbf{m}}_1'$ that is $\overline{\mathbf{m}}_1$ into which $\overline{\mathbf{m}}_2$.s has been embedded starting from the lower bound of $\overline{\mathbf{m}}$, if both $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$ approximate character arrays which contain a well-formed string and the size condition on the destination segmentation abstract predicate is fulfilled; $\top_{\overline{\mathbf{M}}}$ otherwise.

- Consider the unary operator $\mathtt{update}_{\bar{\mathrm{j}},\overline{\mathrm{v}}}$ and let $\overline{\mathbf{m}}$ be a split segmentation abstract predicate. We have to prove that:

$$\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathtt{update}_{\bar{\mathrm{j}},\overline{\mathrm{v}}}]\!](\overline{\mathbf{m}})) \supseteq \{\mathfrak{M}[\![\mathtt{update}_{\mathrm{j},\mathrm{v}}]\!](\mu(\mathtt{m})) : \mu(\mathtt{m})) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})\}$$

  $\mathtt{update}_{\mathrm{j},\mathrm{v}}$ of $\mu(\mathtt{m})$ returns, by definition of $\mathfrak{M}$, $\mu(\mathtt{m})'$ that is $\mu(\mathtt{m})$ where the character at position j has been substituted with the character v, if j is a valid index for $\mu(\mathtt{m})$; $\top_{\mathbf{M}}$ otherwise. Let $\alpha_{\overline{\mathbf{B}}}(\mathrm{j}) = \bar{\mathrm{j}}$ and $\alpha_{\overline{\mathbf{C}}}(\mathrm{v}) = \overline{\mathrm{v}}$. Then, $\mu(\mathtt{m})'$ belongs to $\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\mathtt{update}_{\bar{\mathrm{j}},\overline{\mathrm{v}}}]\!](\overline{\mathbf{m}}))$ because $\mathtt{update}_{\bar{\mathrm{j}},\overline{\mathrm{v}}}$ of $\overline{\mathbf{m}}$, by definition of $\mathfrak{M}_{\overline{\mathbf{M}}}$, is equal to $\overline{\mathbf{m}}'$ that is $\overline{\mathbf{m}}$ where the segment that is valid for $\bar{\mathrm{j}}$ is split so that the segment abstract predicate which occurs at position $\bar{\mathrm{j}}$ is substituted with $\overline{\mathrm{v}}$, if $\gamma_{\overline{\mathbf{B}}}(\bar{\mathrm{j}})$ is equal to the singleton $\{\mathrm{j}\}$ and $\bar{\mathrm{j}}$ is valid for $\overline{\mathbf{m}}$; $\top_{\overline{\mathbf{M}}}$ otherwise.

□

Theorem 3.4. $\mathfrak{P}_{\overline{\mathbf{M}}}$ is a sound over-approximation of $\mathfrak{P}$. Formally,

$$\gamma_{\overline{\mathbf{B}}}(\mathfrak{P}_{\overline{\mathbf{M}}}[\![\mathrm{stm}]\!](\overline{\mathbf{m}})) \supseteq \{\mathfrak{P}[\![\mathrm{stm}]\!](\mu(\mathtt{m})) : \mu(\mathtt{m}) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})\}$$

Proof.
Consider the binary operator `strcmp` and let $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$ be two split segmentation abstract predicates. We have to prove that:

$$\gamma_{\overline{\mathbf{C}}}(\mathfrak{P}_{\overline{\mathbf{M}}}[\![\mathtt{strcmp}]\!](\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2))$$

$$\supseteq$$
$$\{\mathfrak{P}[\![\texttt{strcmp}]\!](\mu(\mathtt{m}_1), \mu(\mathtt{m}_2)) : \mu(\mathtt{m}_1) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_1) \ \wedge \ \mu(\mathtt{m}_2) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_2)\}$$

`strcmp` of $\mu(\mathtt{m}_1)$ and $\mu(\mathtt{m}_2)$ returns an integer value n, resulting from the difference between corresponding character array elements, denoting the lexicographic distance between the strings of interest of $\mu(\mathtt{m}_1)$ and $\mu(\mathtt{m}_2)$, if both contain a well-formed string, $\top_{\mathbb{Z}}$ otherwise, by definition of $\mathfrak{P}$. Then n belongs to $\gamma_{\overline{\mathbf{C}}}(\mathfrak{P}_{\overline{\mathbf{M}}}[\![\texttt{strcmp}]\!](\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2))$ because `strcmp` of $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$, by definition of $\mathfrak{P}_{\overline{\mathbf{M}}}$, is equal to $\overline{\mathrm{n}}$ that is the difference between corresponding segment abstract predicates, denoting the lexicographic distance between $\overline{\mathbf{m}}_1$.s and $\overline{\mathbf{m}}_2$.s, if $\overline{\mathbf{m}}_1$ and $\overline{\mathbf{m}}_2$ are comparable, both approximate character arrays which contain a well-formed string where $\top_{\overline{\mathbf{C}}}$ does not occur; $\top_{\mathbb{Z}}$ otherwise.

□

Theorem 3.5. $\mathfrak{L}_{\overline{\mathbf{M}}}$ is a sound over-approximation of $\mathfrak{L}$. Formally,

$$\gamma_{\overline{\mathbf{B}}}(\mathfrak{L}_{\overline{\mathbf{M}}}[\![\mathrm{stm}]\!](\overline{\mathbf{m}})) \supseteq \{\mathfrak{L}[\![\mathrm{stm}]\!](\mu(\mathtt{m})) : \mu(\mathtt{m}) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})\}$$

Proof.
Consider the unary operator `strlen` and let $\overline{\mathbf{m}}$ be a split segmentation abstract predicate. We have to prove that:

$$\gamma_{\overline{\mathbf{B}}}(\mathfrak{L}_{\overline{\mathbf{M}}}[\![\texttt{strlen}]\!](\overline{\mathbf{m}})) \supseteq \{\mathfrak{L}[\![\texttt{strlen}]\!](\mu(\mathtt{m})) : \mu(\mathtt{m}) \in \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}})\}$$

`strlen` of $\mu(\mathtt{m})$ returns, by definition of $\mathfrak{L}$, an integer value n which denotes the length of the sequence of character before the first null one in $\mu(\mathtt{m})$, if $\mu(\mathtt{m})$ contains a well-formed string; $\top_{\mathbb{Z}}$ otherwise. Then n belongs to $\gamma_{\overline{\mathbf{B}}}(\mathfrak{L}_{\overline{\mathbf{M}}}[\![\texttt{strlen}]\!](\overline{\mathbf{m}}))$ because `strlen` of $(\overline{\mathbf{m}})$, by definition of $\mathfrak{L}_{\overline{\mathbf{M}}}$ is equal to the difference between the lower and the upper bound of $\overline{\mathbf{m}}$.s if $\overline{\mathbf{m}}$ approximates character arrays which contain a well-formed string of interest; $\top_{\overline{\mathbf{B}}}$ otherwise.

□

3.6 Program Abstraction

Adapting M-String to the analysis of real-world `C` programs requires, first of all, a procedure that identifies string operations automatically. A subset of such operations then has to be performed using abstract operations, carried out on a suitable abstract representation. The technique that captures this approach is known as abstract interpretation. A typical implementation is based on an interpreter in the programming language sense: it executes the program by directly performing the operations written down in the source code. However, rather than using concrete values and concrete operations on those values,

part (or the entirety) of the computation is performed in an *abstract domain*, which over-approximates the semantics of the concrete program.

In this chapter, we mainly focus on string abstraction. Therefore we will interpret the portions of the program that do not make use of strings without abstracting values. We only apply abstraction to strings that within the program are manipulated by string operations: when the program deals with string variables that exhibit minimal variation, e.g., string literals, the M-String representation would provide no benefit, and instead it could either hurt performance or it may introduce spurious counterexamples.

Based on the considerations above, it is clear that it is beneficial to reuse and refactor existing tools that implement abstract verification in a modular way on explicit programs. A compilation-based abstraction design that follows this approach was introduced and implemented in [120]. However, such a tool is designed to abstract scalar values only. This is why we need to extend it to operate with more sophisticated domains that represent more complex objects, such as strings.

In the rest of this section, we will first summarize the general approach to abstraction as a program transformation. In Section 3.6.3, we explore the implications of aggregate (as opposed to scalar) domains within this framework. Sections 3.6.4 and 3.6.5 then go on to discuss the semantic (run-time) aspects of the abstraction and which operations we consider as primitives of the abstraction.

3.6.1 Compilation-Based Approach

Instead of (re-)interpreting instructions abstractly, in a compilation-based approach, abstract instructions are transformed into an equivalent explicit code that implements the abstract computation. The transformation takes place before the analysis of the program (e.g., model checking) during the compilation process.

Consequently, the analysis processes the program without needing special knowledge of the abstract domains in use, as the abstraction is encoded directly in the program. Figure 3.1 depicts a comparison of the compilation-based approach with respect the interpretation-based approach adopted by more conventional abstract interpreters.
In a compilation-based approach, two different abstraction perspectives are considered:

1. *static*, referencing to the syntax and the type system,

2. *dynamic*, or semantic, referencing to execution and values.

The LART tool performs syntactic (*static*) abstraction on LLVM bitcode [119]. Syntactic abstraction replaces some of the LLVM instructions that occur in the program with their abstract counterparts, as depicted in Figure 3.2.

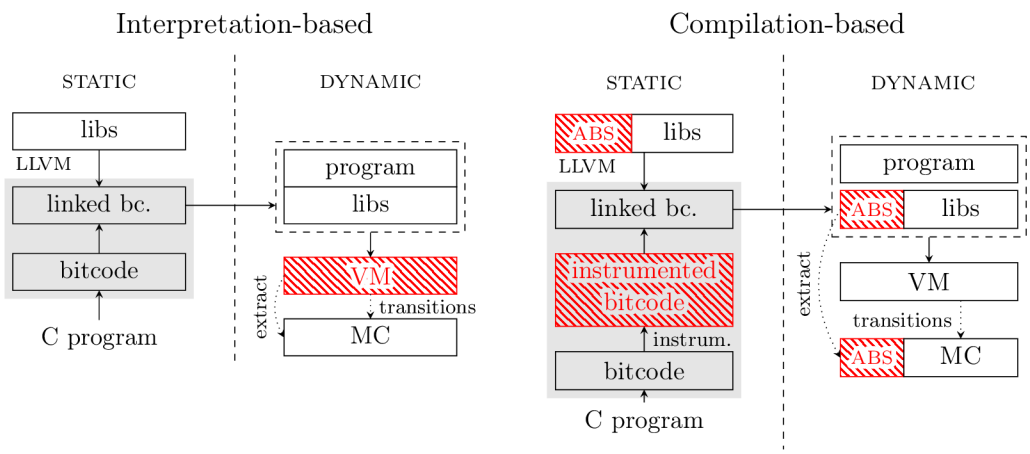


FIGURE 3.1: In the figure, we compare an abstract interpretation with a compilation-based approach. In the interpretation-based approach, the whole abstract interpretation is performed at runtime. The bitecode operations are interpreted abstractely by a virtual machine (VM) which maintains an abstract state. In this way, an abstract state-space is generated for a model-checking algorithm (MC). The compilation-based approach is different. The abstract operations are instrumented into the compiled program and their implementation is provided as a library. Then, the virtual machine executes the instrumented program as a regular bitcode [120].

```
char *a = input_string();
char *b = string();
char *c = strcat(a, b);
int l = strlen(c);
```

```
abstract a = abstract_string();
char *b = string();
abstract c = abstract_strcat(a, lift(b));
abstract l = abstract_strlen(c);
```

FIGURE 3.2: Syntactic abstraction.

3.6.2 Syntactic Abstraction

The first step of program abstraction performed by LART is a syntactic abstraction. Syntactic abstraction replaces LLVM instructions or whole functions with their abstract counterparts. Since we do not want to perform all operations abstractly, we need to classify only those operations that might obtain abstract values as their arguments. The abstract values emerge in the program as input values. From these values, LART computes all operations that might come into contact with abstract values using a combination of data flow and alias analyses. Finally, as a result of analyses, LART obtains a set of possibly abstract operations that are replaced by their abstract equivalents, e.g., `strcat`, `strlen` are replaced by `abstract_strcat` and `abstract_strlen`. Abstract operations then implement the manipulation with abstract values, in our case with M-Strings as described in Section 3.4, in other words the specific meaning of abstract instructions and abstract

values then defines the semantic abstraction.

For the precise formulation of syntactic abstraction, we take advantage of the static type system of LLVM. We leverage the fact that we can assign to each variable its type, which is either concrete or abstract. In this way, we can precisely set a boundary between concrete and abstract values.

Let us consider a simplified version of LLVM. It defines a set of *concrete scalar types* S. The set of all possible types is given by a map Γ that inductively defines all finite (non-recursive) algebraic types over the set of given scalars. To be precise, the set of all possible types $\Gamma(\mathrm{T})$ derived from a set of scalars T is defined as follows:

1. $\mathrm{T} \subseteq \Gamma(\mathrm{T})$, meaning each scalar type is included in $\Gamma(\mathrm{T})$,
2. if $\mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{n} \in \Gamma(\mathrm{T})$ then also the *product type* is in $\Gamma(\mathrm{T})$: $(\mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{n}) \in \Gamma(\mathrm{T}), \mathrm{n} \in \mathbb{N}$,
3. if $\mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{n} \in \Gamma(\mathrm{T})$ then also *disjoint union* is in $\Gamma(\mathrm{T})$: $\mathrm{t}_1 \mid \mathrm{t}_2 \mid \cdots \mid \mathrm{t}_\mathrm{n} \in \Gamma(\mathrm{T}), \mathrm{n} \in \mathbb{N}$,
4. if $\mathrm{t} \in \Gamma(\mathrm{T})$ then $\mathrm{t}^* \in \Gamma(\mathrm{T})$, where t^* denotes pointer type.

In a concrete LLVM program, the set of admissible types comprise those derived from concrete scalars S, i.e., $\Gamma(\mathrm{S})$. In syntactic abstraction, we need to extend admissible types by abstract types. From these, we generate all possible types using Γ. Depending on the type of abstraction, we use a different set of basic abstract types. In the case of scalar abstraction, a set of basic abstract types contains abstract scalar types $\overline{\mathrm{S}}$. Correspondence between abstract and concrete scalars is given by a bijective map $\Lambda : \mathrm{S} \to \overline{\mathrm{S}}$. Finally, each value, which exists in the abstracted program, has an assigned type of $\Gamma(\mathrm{S} \cup \overline{\mathrm{S}})$. Specifically, this implies that the abstraction works with *mixed types*—products and unions might contain both concrete and abstract fields. Moreover, it is possible to create pointers to both abstract or mixed values.

3.6.3 Aggregate Domains

In addition to scalar values that cannot be further decomposed, programs typically operate with more complex data which can be seen as compositions—aggregates—of multiple scalar values. Depending on aggregates' nature, we can classify them as aggregates which contain a variable number of items (arrays), records that contain a fixed number of items in a fixed layout, where each of these can be of a different type. The items in such aggregates can be (and often are) scalars. However, more complex aggregates are also possible: arrays of records, records which in turn contain other records, and so on.

While scalar domains only dealt with simple values, in aggregate abstraction, we consider composite data in the spirit of the above definition. Similarly to scalar domains, abstract aggregate domains approximate concrete aggregate values by describing a particular set of aggregate properties. For example, we can describe a set of aggregates by their

length or a set of values that appear in the aggregate. In the M-String, the kept properties are in the form of segmentation, where segments are further abstracted by bounds and characters. Values in an aggregate domain then keep the representation of chosen properties and operations updates them. For instance, consider an array length property domain—the domain operations in such a case operate only with lengths of arrays, e.g., *abstract concat* of arrays adds together lengths of its arguments (abstract arrays).

In general, aggregate domains can provide arbitrary operations. However, two operations are, in some sense, universal, being elementary memory manipulation operations, namely: byte-wise `access` and `update` of the aggregate. The universality of these operations originates from the fact that all aggregate operations can be represented as accesses and updates. In a low-level representation of a program (assembly), they usually are presented in this form. LLVM allows a slightly higher level of manipulation to access and update individual scalars present in the aggregates (as opposed to bytes). For M-String, though, this distinction is not essential because the scalars stored in `C` strings are individual bytes (characters). All other operations are present in the form of sequences of elementary instructions—possibly encapsulated in functions. Moreover, as in concrete programs, the `access` and `update` represents an interface between scalars and memory, in the abstraction, they form an interface between scalar and aggregate domains (even in the case of byte-oriented access since bytes are also scalars). We refer the reader to the Section 3.5.2 for abstract semantics of `access` and `update`.

In comparison to scalar abstraction, the syntactic abstraction of aggregates does not operate directly with aggregate types. In LLVM, aggregate values are usually represented by a pointer to the underlying aggregate type. Therefore all the accesses and updates are made through the pointers to the aggregates. For instance, strings are represented as a pointer to a character array. We need to take this fact into account when we perform the syntactic abstraction. In the analysis, we consider the pointers to aggregates as base types for the abstraction. In the case of arrays, the base types are concrete pointers to those arrays: let us call them P^*, where $\mathrm{P}^* \subseteq \Gamma(\mathrm{S})$. A set of abstract pointers types P^* then describes types of abstracted aggregates (arrays). As for scalar domains, we define a natural correspondence between pointers to concrete values and abstract aggregates as a bijective map $\Lambda : \mathrm{P}^* \to \overline{\mathrm{P}^*}$. For instance, in the case of M-String abstraction, the map Λ assigns to `char*` a type of M-String value. Finally, we allow all the mixed types generated from scalars and abstract aggregates: $\Gamma(\mathrm{S} \cup \overline{\mathrm{P}^*})$.

Observe that pointers, in general, also in LLVM maintain two pieces of information about memory location: they represent both the memory *object* and an *offset* into that object. In particular, our implementation treats the first 32-bits of the pointer as an object identifier and the last 32-bits as its offset. This distinction is not very relevant in explicit programs because those two components are represented in a uniform way in a single value and often they cannot be distinguished at all. However, the distinction becomes relevant

when dealing with abstract aggregate values. In fact, in this case, the *object* component of the pointer is concrete as it determines a single specific abstract object. On the other side, the *offset* component may or may not be concrete. The choice depends on the specific abstract aggregate domain: it may be more advantageous representing the offset in an abstract way, i.e., by a 32-bit abstract scalar value. Observe that a memory access through such a pointer needs to be treated in both cases as an abstract `access` or `update` operation.

In LLVM, two basic memory access operations are defined, load and store, corresponding to the `access` and `update` operations. It is important to notice that memory access is always explicit: memory is never used in a computation directly. This observation is used in the design of aggregate abstraction, where we can assume that the access to the content of an aggregate will always go through a pointer associated with the abstract object.

3.6.4 Semantic Abstraction

In syntactic abstraction, we dealt with operations' syntax, their types, and the types of values and variables. It described how LART performs a source-to-source transformation. In contrast, semantic abstraction concerns with the values computed at runtime by a program. It defines how abstract operations modify values and how to transfer between concrete and abstract values. Therefore, similarly to syntactic abstraction that defined the maps Λ and Λ^{-1} to transfer between concrete and abstract *types*, the semantic abstraction makes use of lift and lower (cf. Definitions 3.9 and 3.10): operations (instructions) converting values between their concrete and abstract representations. They realize a runtime implementation of domain functions: abstraction ($\alpha_{\overline{\mathbf{M}}}$ in the case of M-String) and concretization ($\gamma_{\overline{\mathbf{M}}}$).

The lift operation implements abstraction of concrete values by a single over-approximating abstract value. For example, in Figure 3.2 on line 3 of the abstracted program, a concrete string `b` is lifted to the abstract domain. This allows performing `abstract_strcat` in a single abstract domain. In other words, operations do not need to consider concrete values because all their arguments are lifted to the abstract domain. This simplifies the implementation of a domain and reduces the number of possible domain interactions. In comparison to Λ, which was a purely syntactic construct, lift and lower accomplish actual conversion of values between domains during program runtime. During program execution, lowering an abstract value into multiple concrete values can be seen as nondeterministic branching in the program (and the lower operator is indeed based on a non-deterministic choice[5] operator). For further details of the program transformation performed by LART, we kindly refer the reader to [120].

[5]In a model checker, the non-deterministic choice would be typically implemented as branching in the state space (and the consequences of all possible outcomes would be explored). In a testing context, however, the choice might implemented as random (choosing one particular path).

3.6.5 Abstract Operations

As a result of syntactic abstraction, we obtain a program that temporarily contains abstract operations. These operations take abstract values as operands and return abstract values as a result. Though, after the program transformation, the resulting program is required to be a semantically valid LLVM bitcode. Therefore, we demand that each abstract operation can be realized as a sequence of concrete instructions. This allows us to obtain an abstract program that does not contain any abstract operations and executes it using standard (concrete, explicit) methods.

Thoroughly, syntactic abstraction substitutes concrete operations with their abstract counterparts: an operation with type $(\mathrm{t}_1, \ldots, \mathrm{t}_\mathrm{n}) \rightarrow \mathrm{t}_\mathrm{r}$ is substituted by an abstract operation of type $(\Lambda(\mathrm{t}_1), \ldots, \Lambda(\mathrm{t}_\mathrm{n})) \rightarrow \Lambda(\mathrm{t}_\mathrm{r})$. Furthermore, transformation inserts lift and lower operations as needed, e.g., in places where concrete values are operands of abstract operations. The implementation is free to select the operations to be abstracted and where value lifting and lowering be inserted, so long type constraints are satisfied. However, it tends to minimize the number of abstracted operations.

In addition to LLVM instructions, the M-String abstraction requires the transformation to abstract function calls to standard library functions such as `strcmp`, `strcat`. From the perspective of syntactic abstraction, we can treat function calls as single atomic operations that take abstract values and produce abstract results. Hence, the transformation substitutes them in the same way as instructions: for instance `strcmp` operation of type $(\mathrm{m}, \mathrm{m}) \rightarrow \mathrm{s}$ is replaced by `abstract_strcmp` of type $(\Lambda(\mathrm{m}), \Lambda(\mathrm{m})) \rightarrow \Lambda(\mathrm{s})$ where m is a concrete character array and s is a concrete scalar result of the string comparison. Afterwards, all abstract operations are implemented by using concrete subroutines (implementation of abstract semantics). For details, see [120].

Observe that, as an alternative approach, the standard library functions `strcat`, `strcmp`, etc. could have been transformed instruction by instruction, by using abstract access and update of a content only. However, the price to pay would have been loosing a certain degree of accuracy in the abstraction, the exact amount depending on the single operation.

3.7 Instantiating M-String

As an aggregate domain, M-String is a parametrizable by scalar domains of characters and indices (bounds). This allows us to tailor the abstraction to the needs of the analysis of string values. Depending on the precision of chosen domains, the instance of the M-String domain will inherit their properties. With more precise domains, the M-String values will maintain higher granularity of segmentation. On the other hand, simpler character representation will decrease the segmentation granularity for the cost of a higher rate of false alarms.

A particular instance of M-String is automatically derived from a parametric description given in Section 3.5, provided a suitable scalar domain $\overline{\mathbf{C}}$ for characters and scalar domain $\overline{\mathbf{B}}$ to represent segment bounds. The instantiation demands that both scalar domains $\overline{\mathbf{C}}$ and $\overline{\mathbf{B}}$ are equipped with operations that appear in the operations with the segmentation. These are mainly elementary arithmetic and relational operations. In the implementation, we provide an M-String domain template that automatically derives all the operations from provided scalar domains.

3.7.1 Symbolic Scalar Values

In program verification, it is common practice to represent certain values symbolically (for instance, inputs from the environment). The symbolic representation allows the verifier to consider all admissible values with a reasonably small overhead. In DIVINE, symbolic verification is implemented using a similar abstraction to one described in the previous section: symbolic scalar values represent their content by SMT formula expressions (terms) in form of abstract syntax trees. The input values are represented as unconstrained variables in the bit vector logic. Operations then build formulae trees from their arguments. In addition to these so-called data definitions, symbolic representation also maintains one global formula of constraints (path-condition), which is derived from the control flow of the program. A more detailed description of this symbolic representation is presented in [120].

The domain of symbolic values (we call it a term domain) requires DIVINE to be augmented with an SMT solver form a suitable theory.[6] DIVINE uses the solver to detect computations that have reached the bottom of the term domain (those are the infeasible paths through the program). Furthermore, as a model checker, it needs to identify equal states or whether the state subsumes another one. This is achieved by the equivalence check of corresponding formulae. With these prerequisites, the symbolic representation in joint with the bit-vector theory is a precise abstraction (i.e., it is not an approximation but models the program state faithfully).

3.7.2 Concrete Characters, Symbolic Bounds

In the evaluation, we instantiate the M-String domain in two ways. The first simpler instantiation sets the domain of characters $\overline{\mathbf{C}}$ to be the concrete domain (i.e., we let the characters be represented by themselves). We let the domain of segment bounds $\overline{\mathbf{B}}$ to be a symbolic 32b integers. This instantiation balances between simplicity on the one hand (both domains we used for parameters were already present in DIVINE) and the ability to describe strings with undetermined length and structure.

[6] For scalars in `C` programs, we use the bitvector theory.

At the implementation level (as described in more detail in the following section), the domain remains generic: the particular domains we picked can be easily substituted by other domains. Compared to the theoretical description of M-String, the implementation uses a slightly simplified representation of segmentation by a pair of arrays (cf. Figure 3.3). The elements of these arrays are characters and bounds, whose type is derived from parametrization, i.e., from the scalar domains $\overline{\mathbf{C}}$ and $\overline{\mathbf{B}}$. The modification of the representation is just optimization for the implementation and does not affect the operations' semantics. The analysis with this representation is presented in Example 3.14.

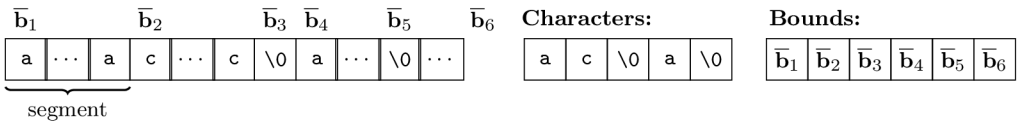


FIGURE 3.3: M-String value with symbolic bounds, where string of interest is from $\overline{\mathbf{b}}_1$ to $\overline{\mathbf{b}}_3$.

This instantiation of M-String is particularly suitable for representing strings with sequences of a single character of variable length, i.e., the strings of the form $a^k b^l c^m \ldots$ where relationships between $k, l, m, \ldots$ can be specified using standard arithmetic and relational operators and each of a, b, c is a concrete letter. This, in turn, allows M-String to be used for the analysis of program behavior on broad classes of input strings described this way. A more detailed description of this approach can be found in Section 3.8.

Example 3.14. Simple program analysis with symbolic bounds and concrete characters:

```
mstring str = abstract_string(x,b̄1,\0,b̄2,y,b̄3,\0,b̄4);
symbolic idx = abstract_int();
if (idx < b̄4) {
    str[idx] = 'y';
    symbolic len = abstract_strlen(str);
}
```

Imagine we are given symbolic bounds $\overline{\mathbf{b}}_1 < \overline{\mathbf{b}}_2 < \overline{\mathbf{b}}_3 < \overline{\mathbf{b}}_4$, then the first line of the transformed program creates `mstring` value with characters $[\mathtt{x}, \mathtt{\backslash 0}, \mathtt{y}, \mathtt{\backslash 0}]$ and bounds $[0, \overline{\mathbf{b}}_1, \overline{\mathbf{b}}_2, \overline{\mathbf{b}}_3, \overline{\mathbf{b}}_4]$. In the following, we describe mstring values as pairs of these two arrays. The second line creates a symbolic index of arbitrary value. On line 3, the program constraints the index to be smaller than mstring maximal length. Otherwise, the update on the next line would yield an error. Next the program assigns to the position of abstract index a character `y`. The assignment is implemented as update operation on mstring value. Depending on the value of the `idx`, the operations results in the following strings $\overline{\mathrm{str}_\mathrm{x}}$, as result we join all possibilities:

1. if $\texttt{idx} < \overline{\mathbf{b}}_1$: $\texttt{idx}$ falls to the first segment: $\overline{\mathrm{str}_1} = ([\texttt{x}, \texttt{y}, \texttt{x}, \backslash\texttt{0}, \texttt{y}, \backslash\texttt{0}], [\texttt{0}, \texttt{idx}, \texttt{idx} + \texttt{1}, \overline{\mathbf{b}}_1, \overline{\mathbf{b}}_2, \overline{\mathbf{b}}_3, \overline{\mathbf{b}}_4])$ and creates a new segment between $\texttt{idx}$ and $\texttt{idx} + \texttt{1}$ containing character $\texttt{y}$. Notice that if $\texttt{idx} = 0$ the first segment is empty, similarly the third segment for $\texttt{idx} + \texttt{1} = \overline{\mathbf{b}}_1$. The string of interest for $\overline{\mathrm{str}_1}$ is of form $\mathrm{x}^{\texttt{idx}}\mathrm{y}^1\mathrm{x}^{\overline{\mathbf{b}}_1 - \texttt{idx} - 1}$.

2. if $\overline{\mathbf{b}}_1 \leq \texttt{idx} < \overline{\mathbf{b}}_2$: than $\overline{\texttt{str}_2} = ([\texttt{x}, \backslash\texttt{0}, \texttt{y}, \backslash\texttt{0}, \texttt{y}, \backslash\texttt{0}], [\texttt{0}, \overline{\mathbf{b}}_1, \texttt{idx}, \texttt{idx} + \texttt{1}, \overline{\mathbf{b}}_2, \overline{\mathbf{b}}_3, \overline{\mathbf{b}}_4])$, with string of interest as join of following forms:

   - if the update is performed right after the first segment, i.e., $\texttt{idx} = \overline{\mathbf{b}}_1$:
     - if and $\left|\overline{\mathbf{b}}_1 - \overline{\mathbf{b}}_2\right| > 1$, i.e., the segment of zeros contains more elements, then the string has form $\mathrm{x}^{\overline{\mathbf{b}}_1}\mathrm{y}$,
     - otherwise the update overwrites the single zero character, hence extends the string of interest by segment of $\texttt{y}$ characters: $\mathrm{x}^{\overline{\mathbf{b}}_1}\mathrm{y}^{\overline{\mathbf{b}}_3 - \overline{\mathbf{b}}_1}$.
   - otherwise between first segment and $\texttt{idx}$ is a terminating zero, hence the string of interest remains unchanged: $\mathrm{x}^{\overline{\mathbf{b}}_1}$.

3. if $\overline{\mathbf{b}}_2 \leq \texttt{idx} < \overline{\mathbf{b}}_3$: then $\overline{\texttt{str}_3} = \overline{\texttt{str}}$, because update stores the same character as is already present in the segment.

4. if $\overline{\mathbf{b}}_3 \leq \texttt{idx} < \overline{\mathbf{b}}_4$: then update creates a new segment inside of sequence of last zeros: $\overline{\mathrm{str}_4} = ([\texttt{x}, \backslash\texttt{0}, \texttt{y}, \backslash\texttt{0}, \texttt{y}, \backslash\texttt{0}], [\texttt{0}, \overline{\mathbf{b}}_1, \overline{\mathbf{b}}_2, \overline{\mathbf{b}}_3, \texttt{idx}, \texttt{idx} + \texttt{1}, \overline{\mathbf{b}}_4])$.

Consequently, the `abstract_strlen` operation on the last line of the program computes the join of all possible lengths of strings of interest, i.e., $\overline{\mathbf{b}}_1 \sqcup_{\overline{\mathbf{B}}} \overline{\mathbf{b}}_3$.

⬡

3.7.3 Symbolic Characters, Symbolic Bounds

The second instantiation is used in benchmarks, where the computation with M-String values encountered abstract scalars (characters). This occurs when the program obtains some character as input from the environment and tries to store it into the M-String value. Therefore, we instantiated the M-String domain with an abstract representation of characters by setting the domain $\overline{\mathbf{C}}$ to be the term domain, which keeps track of symbolic 8b bitvectors (characters in `C` language). In this way, we do not need to lower abstract characters before storing them to the M-Strings, what was needed for the concrete domain used in the previous instantiation. However, we pay the price for more expensive computation with symbolic characters.

3.7.4 Implementation

Finally, we implemented the M-String abstraction as a LART domain.[7] The LART domain is a C++ library that implements abstract semantics of M-String operations presented in Section 3.5. Such a library is then linked to the transformed program allowing the program to perform abstract analysis with model-checker DIVINE. An abstract domain definition in LART consists of a C++ class that describes both the representation (in terms of data) and the operations (in terms of code) of the abstract domain.

In the case of M-String domain, this class contains 2 attributes: an array of *bounds* and an array of *characters*, as outlined in Section 3.7.2 and depicted in Figure 3.3. The class has two type parameters: the domain to use for representing segment bounds and the domain to represent individual characters (i.e., the content of segments). A specific instantiation is then automatically derived by the C++ compiler from the classes which represent the type parameters and the parametric class which represents M-String values.

As a minimal set of operations, the M-String domain implements all requisite aggregate operations: these are `lift`, `update` and `access`. Furthermore, the implementation provides an optimized version of string operations described in Sections 3.5: `strlen`, `strcpy`, `strcat`, `strcmp` and `strchr`. These operations reduce the loss of abstraction precision that would arise if only the abstraction of accesses and updates from strings were used.

Since C strings are stored, in fact, as shared, mutable character arrays, the implementation of the M-String domain needs to reflect the sharing semantics of such arrays. If multiple pointers exist into the same abstract string, modifications through one such pointer must be also visible when the string is accessed through another pointer. Moreover, the pointers do not have to be equal: they may point to different suffixes of the same string. Therefore, the representation of pointers to abstract strings must treat the *object* and the *offset* components separately (see also Section 3.6.3), and the representation of the *offset* component must be compatible with the bound domain $\overline{\mathbf{B}}$.

3.8 Experimental Evaluation

In the evaluation, we chose a few scenarios to demonstrate the properties of the abstraction. In the first scenario, we show that using abstract versions of standard functions is more efficient than if concrete versions were transformed using only abstract string accesses and updates. The second scenario investigates several implementations of standard library functions: we transform them automatically in the means of accesses and updates, and we show that their results agree with results generated by M-String library operations. In the third scenario, we evaluate M-String instantiation with symbolic characters on the set of

[7] The implementation with examples and documentation of domain usage can be found online on the supplementary page: https://divine.fi.muni.cz/2020/mstring

	Word						Sequence					
		Verification(s)						Verification(s)				
	States	8	64	1024	4096	LART(s)	States	8	64	1024	4096	LART(s)
`strcmp`	3562	480	498	472	481	1.70	70	0.26	0.24	0.21	0.25	1.76
`strcpy`	368	9.8	9.1	9.3	9.4	1.70	48	0.20	0.20	0.21	0.20	1.71
`strcat`	7398	898	873	865	843	1.72	105	0.51	0.52	0.53	0.51	1.72
`strchr`	49	0.3	0.4	0.3	0.3	1.71	15	0.04	0.04	0.03	0.04	1.70
`strlen`	78	1.1	1.2	1.0	1.3	1.70	16	0.05	0.04	0.05	0.06	1.81

TABLE 3.2: Measurements of M-String operations on two types of inputs: *Word* and *Sequence* described in Section 3.8. Each benchmark measures a size of state space and verification time for input M-Strings of a given length. Lastly, the table shows an average transformation time (LART). All measurements of time are in seconds. The size of state space does not change for different lengths of input—for more details, see discussion in Section 3.8.

	8		64		1024	
	Time(s)	States	Time(s)	States	Time(s)	States
`strcmp`	1.24	197	260	1597	T	–
`strcpy`	0.7	122	61.5	962	T	–
`strcat`	15.8	1102	T	–	T	–
`strchr`	0.04	16	0.05	16	0.05	16
`strlen`	0.19	46	9.57	326	T	–

TABLE 3.3: Benchmark of standard library functions abstracted using only the M-String definitions of *access* and *update* operations for *Sequence* inputs of size 8, 64 and 1024 characters. Verification for *Word* strings times out in most of the instances.

benchmarks from real software that contain buffer-overflow errors. Here we show that M-String can efficiently detect real-world bugs as well as to prove that program does not contain them after they are fixed. The last benchmark shows the use of abstractions on more complex `C` programs. As an example, we analyze automatically generated parsers from `bison` and `flex` tools on abstract (M-String) inputs. The resource limits for all scenarios were the same: each verification run was limited to 4 processing units (cores), 80 GB of memory, and 1 hour of CPU time. The processor used to run benchmarks was AMD EPYC 7371 clocked at 2.60GHz.

M-String Operations

The first group of benchmarks focuses on the use of resources by abstraction. Benchmarks compare the effectiveness of abstract domain operations with the automatically abstracted implementation of standard library functions from `PDCLib`, a public-domain `libc` implementation, using only essential abstract operations: `lift`, `update` and `access`. The results

depicted in Table 3.2 were measured with parametrized M-String inputs of two kinds (l is a parametric length of the input):

- *Word* w is a string of the form: $w = c_1^{i_1} \cdot c_2^{i_2} \cdot \ldots \cdot c_l^{i_l}$ where $\sum_{k=1}^{l} i_k \leq l$ and c_x is an arbitrary character from domain $\overline{\mathbf{C}}$.
- *Sequence* w is a string of the form $w = c^i$, where $i \leq l$ and c is a character from domain $\overline{\mathbf{C}}$.

For each standard library function and input type, we created an isolated benchmark in two variants: one using an abstract semantics of M-String operations (see Table 3.2) and the other variant (Table 3.3) only with an automatic abstraction of essential aggregate operations.

The first notable difference between automatically abstracted implementations of library functions and M-String operations is that the analysis of the former timeouts for input strings longer than 64 characters. The main cause of the lifted implementation's inefficiency is that it has to iterate over all characters, while M-String operations leverage iteration over larger segments. This difference also causes a blow-up of the model checker's state space for the lifted implementations while the state space size does not change for M-String operations. The reason for this is the fact that the number of segments does not change with the length of the input. Therefore M-String operations always perform the same computation independently of the M-String length.

C Standard Libraries

In the second set of benchmarks, we investigate whether the implementation from several standard libraries matches the expected results of abstract implementation. In other words, we perform an equivalence check of results obtained from M-String operations with the results of the automatically abstracted (originally concrete) standard library functions. We expect that both give the same results. For the evaluation, we picked three open-source libraries: `PDClib`, `musl-libc` and µCLibc. Since results for the libraries are rather similar, we present here only an evaluation of `PDClib` functions. The remaining results are provided in the Supplementary Material. All benchmarks showed that our implementation matches the standard one.

Similarly, as in the previous case, these benchmarks suffer from the state space blow up caused by an exponential number of possible character combinations. For this reason, we decreased the size of the input strings. In addition to large state space, many string accesses and updates of concrete implementations result in a large SMT formulae, causing a long time spent in solvers.

Furthermore, notice that the computation analysis with *Word* input, which has more segments, results in longer execution times than the analysis with *Sequence*. The reason

	Word						Sequence					
	4		8		16		4		8		16	
	Time(s)	States	Time(s)	States	Time(s)	States	Time(s)	States	Time(s)	States	Time(s)	States
`strcmp`	14.3	1005	105	2989	1350	9741	2.17	204	5.09	376	16.5	720
`strcpy`	5.15	515	57.4	1823	912	6935	0.83	183	2.49	347	9.14	675
`strcat`	468	5748	T	–	T	–	8.56	751	113	2535	1940	9463
`strchr`	0.08	22	0.08	22	0.08	22	0.3	17	0.3	17	0.4	17
`strlen`	0.66	91	4.13	259	68.8	883	0.15	34	0.28	54	0.65	94

TABLE 3.4: Verification results of functions from `PDCLib` with timeout of 1 hour. Measurements show the size of state space and verification time for the parametric length of the input.

is that the more segments naturally also causes overhead for the analyses. For example, The M-String needs to consider cases when some segments have zero length: this causes a hard SMT queries because, in the worst case, it needs to check all possible strings for given segment bounds and characters.

Veriabs Overflow Benchmarks

In this scenario, we show that the domain is capable of efficient overflow bug finding. Veriabs benchmarks exhibit overflow errors and fixed variants of real-world software. To soundly prove correctness of these benchmarks, we instantiate M-string with term domain also for characters. Hence we can reason about arbitrary strings of a symbolic length. However, as a drawback of this instantiation is that whenever the length of the string bounds a loop, we might have to unroll the loop infinitely in the analysis—these cases timeouts in the correct benchmarks.

	Correct		Error Found		
	Tests	Time(s)	Tests	Time(s)	Timeout
`apache`	0	–	26	384.26	24
`openser`	43	234.13	45	105.93	6
`wu-ftpd`	8	35.78	14	2461.27	19
`libgd`	4	9.01	4	1.85	0
`madwifi`	5	0.51	5	0.55	0
`gxine`	1	0.53	1	0.25	0

TABLE 3.5: Veriabs overflow benchmarks depict a few categories of programs exhibiting an overflow error and their fixed variants. The table shows the number of solved benchmarks (*tests*) and accumulated time for each category. For each category, the table depicts correctly verified benchmarks, benchmarks where the verifier was able to find an error and number of timeouts.

Parsers

Lastly, we evaluate our implementation on more complex programs: automatically generated parsers. For the generation, we use a tool `Bison`. It reads a language specification in the form of context-free grammar and produces a C parser that accepts the language. In the benchmarks, we generate two such parsers. The first one accepts a language of numerical expressions (mathematical expressions that consist of numbers and binary operators). The second parser is of a simple programming language with variables and branching. We present a evaluation for both parsers in Table 3.6. As with the previous benchmark sets, the M-String inputs with a smaller number of segments outperformed other analyses.

	Numeric Expressions Grammar					
	10		**20**		**35**	
	Time(s)	**States**	**Time(s)**	**States**	**Time(s)**	**States**
`add`	40.2	416	319	3548	T	–
`ones`	5.54	62	8.12	196	189	2186
`alter`	708	105	1582	11k	T	–

TABLE 3.6: Measurements of time and size of state space for analyses of automatically generated parsers. In these benchmarks, we use specifically hand-crafted M-String inputs for parsers. For parsing of mathematical expressions, it was: `addition` input had a form of two arbitrary numbers with a plus sign between them, `ones` was a simple input of a single digit sequence, and lastly, `alternation` was input that produced complicated M-Strings by alternating digits inside of expressions. The other parser of simple programming language was evaluated on: `value` was in input that created a variable and assigned a constant to it, `loop` was a short program with some control flow and `wrong` was a program that contained a syntax error.

	Simple Programming Language Grammar					
	10		**100**		**1000**	
	Time(s)	**States**	**Time(s)**	**States**	**Time(s)**	**States**
`value`	6.58	38	90.4	488	1100	4988
`loop`	1.53	23	4.88	23	33.3	23
`wrong`	7.34	82	67.7	892	311	8992

TABLE 3.6: *cont*

3.9 Discussion

A new segmentation-based abstract domain for approximating `C` strings has been introduced, whose main novelty lies in abstracting both index bounds and substrings while managing strings as a pair of two string buffers: the string of interest itself, and a tail of allocated and possibly initialized but unused memory.

The presented approach enables a more precise modelling of the functions in the standard `C` library for strings, considering also the known weaknesses for the management of terminating null characters and buffer bounds. The M-string domain results effective for identifying security leaks caused by string manipulation errors, e.g., buffer overflows.

After theoretically describing the domain and the basic operations on strings, we implemented (using `C++` language) the abstract semantics combining them with a tool that starting from string-manipulating `C` programs lifts them to the M-String domain.

Our experimental results also focused on tuning the parameters of M-String (the domains for both segment content and segment bounds) by instantiating them by both concrete and symbolic characters and by symbolic (bitvector) bounds.

Chapter 4

COMBINING STRING DOMAINS

In this chapter we specialize the segmentation domain FunArray, introduced in Section 3.2, for array content analysis [57] and we combine it, through reduced product, with less sophisticated existing string abstract domains in order to improve the precision of the analysis. Note that if the M-String abstract domain (cf. Section 3) is inspired by FunArray and it is tailored for string analysis of `C` programs, in the following we use FunArray "as is", and we combine it with general-purpose existing string abstract domains.

The contribution of this chapter is still submitted for publication.

Chapter Structure

Section 4.1 highlights some limits of existing string abstract domains and explains our contribution. Section 4.2 introduce the syntax of some string operations of interest. Section 4.3 defines the concrete domain and semantics. Section 4.4 introduces the basic abstract domains for string analysis. Section 4.5 specializes the Segmentation domain [57], summarised in Section 3.2, to the approximation of strings. Section 4.6 defines our refined string abstract domains. Section 4.7 concludes.

4.1 Introduction

Existing string abstract domains partially track content and/or shape information of string values. Let us consider some basic domain for strings [49]. The String Length domain approximates the length of a string through an interval (e.g., the interval $[2, 5]$ represents all the possible strings whose length ranges from 2 to 5). The Character Inclusion domain approximates strings through a pair of sets highlighting the characters definitely and possibly contained (e.g., the pair of sets $(\{\mathrm{a}, \mathrm{b}\}, \{\mathrm{a}, \mathrm{b}, \mathrm{c}\})$ represents all the possible strings definitely containing the characters 'a' and 'b' and probably containing the character 'c'). The String Length and the Character Inclusion domains respectively approximate string shape and string content information. On the other hand, the Prefix domain partially detect shape and content information of the strings it approximates. For example, the

prefix ab$*$ represents all the possible strings sharing the starting sequence "ab", whose minimal length is 2. The Segmentation domain [57], summarised in Section 3.2, is a more sophisticated abstract domain that was originally proposed to abstract arrays content by bounded, consecutive, non-overlapping and possibly empty segments. It allows both the abstraction of array elements values and relational abstractions between arrays content and their indexes.

Each abstract domain represents a trade-off between the precision and the efficiency of the analysis, since usually more precise analyses require less efficient abstract domains, slowing down the convergence of the computation of the fixpoint algorithm. However, different abstract domains can be composed in various ways (such as the Cartesian and the reduced product) within the Abstract Interpretation framework [38, 42, 56].

State of the Art

While the compositional operators introduced in Section 2.6.4 have been extensively used to improve the precision of numerical analyses and implemented in various tools [59, 64], when we talk about string analysis, their usages and instantiations are short.

In particular, Yu et al. [185] conducted an analysis of strings on Web applications, with the purpose of verifying properties on strings which are relevant for security by combining (using the Cartesian product) the relation and alphabet abstractions. Elements in the product lattice are the so-called *abstraction classes* which are chosen heuristically. Amadini et al. [10] designed a framework that allows a flexible combination of several string abstract domains for JavaScript analysis. Such combination was implemented as an extension of the SAFE tool [122]. Since such approach relied on the Cartesian product, the resulting analysis exposed more precise results than the single abstract domains, but such information could have been reconstructed from the individual execution of the analyses. A further improvement of such approach [11] introduced a general and modular framework to combine several string abstract domains through the reduced product, by introducing the concept of *reference domain*. Let $\mathbf{D}$ be a concrete domain, and $\overline{\mathbf{D}}_1, ..., \overline{\mathbf{D}}_\mathrm{n}$ be abstract domains. A reference domain soundly approximates $\mathbf{D}$ and captures any information expressible in each of the abstract domains $\overline{\mathbf{D}}_1, ..., \overline{\mathbf{D}}_\mathrm{n}$. The reference domain can be applied as a medium for systematically transferring information from all these abstract domains. The information is exchanged through a *strengthening function*, i.e., a closure operator on $\overline{\mathbf{D}}_1, ..., \overline{\mathbf{D}}_\mathrm{n}$. Depending on the considered domains, the strengthening function may be computational expensive. Therefore, the authors introduced a *weak strengthening function* that is less precise but more efficient.

Contribution

In this chapter we combine through the reduced product some well-known string abstract domains (such as the String Length [11], the Character Inclusion and the Prefix [49] domains) with the Segmentation domain proposed by Cousot et al. in [57] in order to get more sophisticated and precise abstractions. On the one hand, string domains track information about the content of string values. On the other hand, the Segmentation domain behaves like a functor that allows to lift basic domains through structural information, and since it represents the string values by distinct segments, it allows string domains to keep track information only on a part of the string.

The main contribution of the chapter can be summarized as follows:

- We highlight the limits of the basic existing string abstract domains that deal with string shape and content properties separately or partially integrated, and the need to overcome these limits by providing a systematic construction of domains where string shape information is balanced with respect to string content information.

- We specialize the Segmentation domain for string analysis.
  For instance, the string segmentation 'a' $[2, 4]$ 'b' $[0, 2]$ has two consecutive segments (i.e., 'a' $[2, 4]$ and 'b' $[0, 2]$), each of which is composed by the character representation followed by a numerical interval. The latter indicates how many times the character approximation that precedes it is repeated. Thus, the abstract string 'a' $[2, 4]$ 'b' $[0, 2]$ represents all the possible strings having the character 'a' repeated from 2 to 4 times followed by the character 'b' repeated from 0 to 2 times.

- We combine basic string abstract domains with the Segmentation domain, where the tracked string information is homogeneous by exploiting the notions of Granger methodology and refinement operators [87], and we show how such reduced products can be actually computed.
  Consider the segmentation a$*$ $[2, 4]$ b$*$ $[0, 2]$ where character values are abstracted within the Prefix domain. This segmentation represents all the strings of length ranging from 2 to 4 and beginning with the character 'a' concatenated with strings of length ranging from 0 to 2 and beginning with the character 'b'. Note that this combination of domains is in position to fully cover both string shape and content information.

- We show how inconsistency conditions can be associated with abstract values, enabling the analysis to detect potential vulnerabilities within the source code.

To better understand the motivation and the results of this work, consider the example below.

String domain	Abstract elements
$\overline{\mathbf{CI}}$	({a,d,0,x,y,8}, {a,d,0,x,y,8,4}) ({a,d}, {a...z,0...9}) ($\emptyset$, a,d,x,y,z,0...9) ...
$\overline{\mathbf{PR}}$	a∗, aad∗, aaddd∗ ...
$\overline{\mathbf{SL}}$	$[14, 14]$, $[5, 16]$, $[0, 20]$ ...
$\overline{\mathbf{S}}$	'a' $[2, 2]$ 'd' $[3, 3]$ '0' $[2, 2]$ 'x' $[2, 2]$ 'y' $[3, 3]$ '8' $[2, 2]$ literal $[5, 5]$, cypher $[2, 2]$ literal $[5, 5]$ cypher $[2, 2]$ **[a,d]**$[5, 5]$ **[0,0]**$[2, 2]$ **[x,y]**$[5, 5]$ **[7,10]**$[2, 2]$

TABLE 4.1: Introductory example.

Example 4.1. Let $\overline{\mathbf{SL}}$, $\overline{\mathbf{CI}}$, $\overline{\mathbf{PR}}$ and $\overline{\mathbf{S}}$ be the String Length, Character Inclusion, Prefix and Segmentation string abstract domains respectively. Table 4.1 depicts possible elements of these abstract domains. Notice that all the abstractions in Table 4.1 approximate a set of string values that contains, for example, the literal "aaddd00xxyyy88".

Consider in particular the following abstract values:

(i) aaddd∗ ($\overline{\mathbf{PR}}$)

(ii) **[a,d]**$[5, 5]$ **[0,0]**$[2, 2]$ **[x,y]**$[5, 5]$ **[7,10]**$[2, 2]$ ($\overline{\mathbf{S}}$)

Prefix (i) approximates all the strings starting with the sequence of characters "aaddd", while segmentation (ii) approximates all the strings starting with a character from 'a' to 'd' repeated 5 times concatenated to the cypher '0' repeated two times and so on. Note that we darken the segment character representations in order to distinguish it from the segment bounds. The reduced product between these two values leads to an improvement of the information precision tracked by each of the abstract elements. In particular, we can safely add to the end of the prefix aaddd∗ the sequence of cyphers 00 as the segmentation approximates strings where the integer 0 occurs at position 5 and 6, obtaining aaddd00∗. On the other hand, the prefix clearly abstracts strings starting with a sequence of two 'a' followed by a sequence of three 'd'. Thus the first segment of the segmentation value can be split accordingly, yielding to:

$$\langle \text{aaddd00}*, \textbf{[a,a]}[2, 2]\ \textbf{[d,d]}[3, 3]\ \textbf{[0,0]}[2, 2]\ \textbf{[x,y]}[5, 5]\textbf{[7,10]}[2, 2]\rangle \quad (\overline{\mathbf{PR}} \otimes \overline{\mathbf{S}})$$

Observe that in this way both abstract components have been lifted to a more accurate representation in the resulting reduced product.

⬡

4.2 Syntax

We briefly recall the string operators defined in [49]. While mainstream programming languages support a wider set of operators, we will focus our discussion and approach on this minimal set since they support the most important computations over string values.

In particular, let `str` be a sequence of characters. `new String(str)` is the operator that generates a new constant string. Instead, let `s1` and `s2` be two strings; `concat(s1,s2)` concatenates `s1` and `s2`. Then, let `s` be a string and, let `b` and `e` be two integer values. $\texttt{substring}_{\texttt{b}}^{\texttt{e}}\texttt{(s)}$ returns the substring of `s` from the index `b` to the index `e`. Finally, let `s` be a string and let `c` be a character. $\texttt{contains}_{\texttt{c}}\texttt{(s)}$ returns true if and only if the character `c` appears in `s`.

4.3 Concrete Domain and Semantics

In the following, we recall the concrete domain and semantics of [49]. We appropriately modify the original notation to be consistent with the rest of this chapter.

4.3.1 Concrete Domain

Formally, given a finite set of characters Σ, i.e., an alphabet, and a set of all possible finite sequences of characters Σ^* (cf. Section 2.1), a concrete domain is the complete lattice $\big(\mathcal{P}(\Sigma^*), \subseteq, \emptyset, \Sigma^*, \cap, \cup\big)$ where: $\mathcal{P}(\Sigma^*)$ denotes the powerset of Σ^* (that is, the set of all string values), $\subseteq$ denotes the set inclusion (i.e., the partial order between elements in $\mathcal{P}(\Sigma^*)$), the emptyset $\emptyset$ is the bottom element of the lattice $\mathcal{P}(\Sigma^*)$, Σ^* is the top element of the lattice $\mathcal{P}(\Sigma^*)$, the set intersection $\cap$ is the greatest lower bound operand in $\mathcal{P}(\Sigma^*)$, and the set union $\cup$ is the least upper bound operand in $\mathcal{P}(\Sigma^*)$.

4.3.2 Concrete Semantics

We now define the concrete semantics $\mathfrak{S}$ and $\mathfrak{B}$. Formally:

$$\mathfrak{S} : \texttt{Stm} \times \Sigma^* \cup [\mathcal{P}(\Sigma^*)]^{\mathrm{k}} \rightarrow \mathcal{P}(\Sigma^*)$$

$$\mathfrak{B} : \texttt{Stm} \times \mathcal{P}(\Sigma^*) \rightarrow \{\texttt{true}, \texttt{false}, \top_{\mathbf{B}}\}$$

where `Stm` denotes a generic string operators. In particular, the semantics $\mathfrak{S}$ applies to `new String`, `concat` and `substring`. Note that we deal with unary and binary operations then k might be only 1 or 2. On the other hand, the semantics $\mathfrak{B}$ applies to `contains`. The concrete semantics is defined on the string operators introduced in Section 4.2 as follows:

- New String

  $\mathfrak{S}[\![\texttt{new String}(\sigma)]\!]() = \{\sigma\}$, that is, the semantics $\mathfrak{S}$, when applied to `new String(σ)`, returns the singleton $\{\sigma\}$.

- String concatenation

  $\mathfrak{S}[\![\texttt{concat}]\!](\mathrm{S}_1, \mathrm{S}_2) = \{\sigma_1 + \sigma_2 \mid \sigma_1 \in \mathrm{S}_1 \wedge \sigma_2 \in \mathrm{S}_2\}$. In this case, let $\mathrm{S}_1, \mathrm{S}_2 \in \mathcal{P}(\Sigma^*)$ and let $\sigma_1 + \sigma_2$ denote the concatenation between the strings σ_1 and σ_2. The

semantics $\mathfrak{S}$ returns a set containing all the possible concatenations between the strings which belong to S_1 and S_2.

- Substring

  $\mathfrak{S}[\![\mathtt{substring}_{\mathtt{b}}^{\mathtt{e}}]\!](S) = \{\sigma_{\mathtt{b}}...\sigma_{\mathtt{e}} \mid \sigma_1...\sigma_n \in S \wedge n \geq \mathtt{e} \wedge \mathtt{b} \leq \mathtt{e}\}$ where $S \in \mathcal{P}(\Sigma^*)$. In this case, the semantics $\mathfrak{S}$ returns a set containing all the substrings from the b-th to the e-th character ($\sigma_{\mathtt{b}}$ and $\sigma_{\mathtt{e}}$ respectively) of the strings which belong to S. If a string is too short the resulting set will not contain any element related to it.

- Is contained

  $$\mathfrak{B}[\![\mathtt{contains}_{\mathtt{c}}]\!](S) = \begin{cases} \mathtt{true} & \text{if } \forall \sigma \in S : \mathtt{c} \in \mathrm{char}(\sigma) \\ \mathtt{false} & \text{if } \forall \sigma \in S : \mathtt{c} \notin \mathrm{char}(\sigma) \\ \top_{\mathbf{B}} & \text{otherwise} \end{cases}$$

  Let $S \in \mathcal{P}(\Sigma^*)$. In this case, $\mathfrak{B}$ returns `true` if all the strings in S contain the character `c`, `false` if `c` is not contained by any of the strings in S, otherwise it returns $\top_{\mathbf{B}}$.

Example

```
1  ResultSet getPerishablePrices(String lowerBound) {
2  String query = "SELECT '$' || (RETAIL/100) FROM INVENTORY WHERE ";
3  if (lowerBound != null)
4     query += "WHOLESALE > " + lowerBound + " AND ";
5
6  query += "TYPE IN (" + getPerishableTypeCode() + ");";
7  return statement.executeQuery(query);
8  }
9  String getPerishableTypeCode() {
10 return "SELECT TYPECODE, TYPEDESC FROM TYPES WHERE NAME = 'fish' OR NAME
     ='meat'"
11 }
```

Listing 4.1: Java code building up and executing an SQL query

Listing 4.1 reports the source code of the example that will be used to explain how the analysis with the basic string abstract domains works. This example is taken from Gould et al. [86]. In particular, method `getPerishablePrices` aims at executing a SQL query that selects, given a string representing a `lowerBound`, all the items from an inventory that are either `fish` or `meat` and whose `WHOLESALE` is greater than the `lowerBound` if such parameter is not null. Such SQL query contains several errors:

1. `'$'|| (RETAIL/100)` concatenates the character `$` with the numeric expression `RETAIL/100`,

2. `lowerBound` is an arbitrary string that might contain non-numeric characters, and therefore the comparison between `WHOLESALE` and its value might cause a runtime error, and

3. the subquery returned by `getPerishableTypeCode` returns two columns instead of one, and this could cause an error.

For the sake of readability we assign a shortcut to the string constants appearing in Listing 4.1 (cf. Table 4.2).

Name	String constant
σ_1	`"SELECT '$' || (RETAIL/100) FROM INVENTORY WHERE "`
σ_2	`"WHOLESALE > "`
σ_3	`" AND "`
σ_4	`"TYPE IN ("`
σ_5	`"SELECT TYPECODE, TYPEDESC FROM TYPES WHERE NAME = 'fish' OR NAME = 'meat'"`
σ_6	`");"`

TABLE 4.2: Shortcuts of string constants in Listing 4.1.

4.4 String Abstract Domains

In this section we provide an overview of a suite of basic string abstract domains [11, 49] (cf. Appendix A for more details), which collect the set of possible string values. In particular, we highlight and compare how the following string abstract domains track content and shape information of strings.

4.4.1 String Length

The first domain we consider is the String Length abstract domain $\overline{\mathbf{SL}}$, as presented in [11, 127]. This domain tracks, through a numerical interval $[m, M]$ (s.t. $m \in \mathbb{N}$ and $M \in \mathbb{N} \cup \{\infty\}$), the minimum (i.e., m) and the maximum (i.e., M) length of the concrete strings it represents.

The String Length abstract domain detects information about the shape of the concrete strings it represents, i.e., their lengths. In particular, it is able to precisely approximate only the empty string. The abstraction, in all the other cases, totally loses the information about the strings content (e.g., the knowledge about characters order, repetitions, etc.). Any abstract element different from the top element ($\top_{\overline{\mathbf{SL}}} = [0, \infty]$) leads to a finite set of concrete strings whose cardinality depends on the cardinality of the alphabet Σ and on the width of the abstract interval. $\overline{\mathbf{SL}}$ can be implemented in a simple and efficient way

as operations on it run in constant time. $\overline{\mathbf{SL}}$ is an infinite lattice, and it does not respect the ACC, which is the reason why it has been equipped with a widening operator.

Appendix A.1 reports the complete formalization and proof of soundness of this domain and its semantics.

Example 4.2. The result of the analysis of the program in Listing 4.1, using $\overline{\mathbf{SL}}$, is given below.

Program point	Variable	$\overline{\mathbf{SL}}$
`pp.2`	`query`	$[\lvert\sigma_1\rvert, \lvert\sigma_1\rvert]$
`pp.3`	`lowerBound`	$[0, \infty]$
`pp.4`	`query`	$[\lvert\sigma_1\rvert + \lvert\sigma_2\rvert + \lvert\sigma_3\rvert, \infty]$
`pp.5`	`query`	$[\lvert\sigma_3\rvert, \infty]$
`pp.6`	`query`	$[\lvert\sigma_3\rvert + \lvert\sigma_4\rvert + \lvert\sigma_5\rvert + \lvert\sigma_6\rvert, \infty]$

At `pp.2`, the variable `query` is associated to a state containing the abstraction of σ_1. The latter is approximated by the length interval $[\lvert\sigma_1\rvert, \lvert\sigma_1\rvert]$. The input variable `lowerBound` appears at `pp.3` and, as it is unknown, it is abstracted by the top element of the $\overline{\mathbf{SL}}$ lattice, i.e., $\top_{\overline{\mathbf{SL}}} = [0, \infty]$. At `pp.4`, the variable `query` is associated to a state containing the concatenation of the abstractions of σ_1, σ_2, `lowerBound` and σ_3, i.e., $[\lvert\sigma_1\rvert + \lvert\sigma_2\rvert + 0 + \lvert\sigma_3\rvert, \lvert\sigma_1\rvert + \lvert\sigma_2\rvert + \infty + \lvert\sigma_3\rvert] = [\lvert\sigma_1\rvert + \lvert\sigma_2\rvert + \lvert\sigma_3\rvert, \infty]$. Then, at `pp.5`, the least upper bound ($\sqcup_{\overline{\mathbf{SL}}}$) between the abstract value of `query` after `pp.2` and after `pp.4` is computed, i.e., $[\lvert\sigma_1\rvert, \lvert\sigma_1\rvert] \sqcup_{\overline{\mathbf{SL}}} [\lvert\sigma_1\rvert + \lvert\sigma_2\rvert + \lvert\sigma_3\rvert, \infty] = [\mathsf{min}(\lvert\sigma_1\rvert, \lvert\sigma_2\rvert, \lvert\sigma_3\rvert), \max(\lvert\sigma_1\rvert, \infty)] = [\lvert\sigma_3\rvert, \infty]$. Finally, at `pp.6`, `query` is associated to a state containing the concatenation of the abstractions of itself after `pp.5` and the strings σ_4, σ_5 and σ_6. Thus, at the end, `query` will have a length between $\lvert\sigma_3\rvert + \lvert\sigma_4\rvert + \lvert\sigma_5\rvert + \lvert\sigma_6\rvert$ and ∞.

⬡

4.4.2 Character Inclusion

The Character Inclusion abstract domain $\overline{\mathbf{CI}}$, as defined in [49], tracks, through a *pair of sets* $(\mathrm{C}, \mathrm{MC})$, the characters that are certainly contained (i.e., C) and the characters that might be contained (i.e., MC) by the concrete strings it represents.

The Character Inclusion abstract domain detects information about the content of the concrete strings it represents. Like the String Length abstract domain, also $\overline{\mathbf{CI}}$ is able to precisely approximate the empty string only, and it makes possible to infer the minimum length of the strings it abstracts, which corresponds to the cardinality of the set C. The abstraction, in all the other cases, loses the information about concrete strings shape. Indeed, similarly to $\overline{\mathbf{SL}}$, $\overline{\mathbf{CI}}$ does not preserve the information about the order of appearance of the characters, characters repetitions, and other relevant string properties. Any abstract element different from the bottom element ($\bot_{\overline{\mathbf{CI}}}$) leads to an infinite set of concrete strings. $\overline{\mathbf{CI}}$ is not computationally expensive and it has finite height; consequently,

the termination of the analysis is guaranteed by its least upper bound (used as widening operator).

Appendix A.2 reports the complete formalization and proof of soundness of this domain and its semantics.

Example 4.3. The result of the analysis of the program in Listing 4.1, using $\overline{\mathbf{CI}}$, is given below.

Program point	Variable	$\overline{\mathbf{CI}}$
`pp.2`	`query`	(C_1, MC_1)
`pp.3`	`lowerBound`	$(\emptyset, \Sigma)$
`pp.4`	`query`	$(C_1 \cup C_2 \cup C_3, \Sigma)$
`pp.5`	`query`	(C_1, Σ)
`pp.6`	`query`	$(C_1 \cup C_4 \cup C_5 \cup C_6, \Sigma)$

At `pp.2`, the variable `query` is associated to a state containing the abstraction of σ_1. The latter is approximated by a pair of set of characters, i.e., (C_1, MC_1) with $C_1 = MC_1$. The input variable `lowerBound` is abstracted by the top element of the $\overline{\mathbf{CI}}$ lattice, i.e., $\top_{\overline{\mathbf{CI}}} = (\emptyset, \Sigma)$. At `pp.4`, the variable `query` is associated to a state containing the concatenation of the abstractions of σ_1, σ_2, `lowerBound` and σ_3, i.e., $(C_1 \cup C_2 \cup \emptyset \cup C_3, MC_1 \cup MC_2 \cup \Sigma \cup MC_3) = (C_1 \cup C_2 \cup C_3, \Sigma)$. Then, at `pp.5`, the least upper bound ($\sqcup_{\overline{\mathbf{CI}}}$) between the abstract value of `query` after `pp.2` and after `pp.4` is computed, i.e., $(C_1, MC_1) \sqcup_{\overline{\mathbf{CI}}} (C_1 \cup C_2 \cup C_3, \Sigma) = (C_1 \cap (C_1 \cup C_2 \cup C_3), MC_1 \cup \Sigma) = (C_1, \Sigma)$. Finally, at `pp.6`, `query` is associated to a state containing the concatenation of the abstractions of itself after `pp.5` and the strings σ_4, σ_5 and σ_6. Thus, at the end, `query` will surely contain the characters in σ_1, σ_4, σ_5 and σ_6 and it will probably contain any character.

⬡

4.4.3 Prefix and Suffix

The Prefix abstract domain $\overline{\mathbf{PR}}$, presented in [49], approximates a set of concrete strings through a sequence of characters whose last element is an asterisk $*$. Such a symbol denotes any possible suffix string (the empty string ε is included). On the other hand, the Suffix abstract domain $\overline{\mathbf{SU}}$ for string values mirrors the Prefix domain and its notation and all its operators are dual to those of $\overline{\mathbf{PR}}$. The suffix domain abstracts strings through their suffix preceded by an asterisk $*$, which denotes any possible prefix string, ε included. Moreover, Amadini et al. [11] discussed the Prefix-Suffix abstract domain $\overline{\mathbf{PS}}$. The latter domain approximates string values by their prefix and suffix simultaneously. Again, the notation and the operators of this domain can be easily induced by the Prefix and the Suffix domains. Indeed, the Prefix-Suffix domain abstracts strings through a pair of strings (p, s) which concretizes to the set of all possible strings having p as prefix and s as suffix (notice that, here, the asterisk $*$ is not included in the definition).

The domains discussed above partially detect both content and shape of the concrete string they represent. Indeed, $\overline{\mathbf{PR}}$, $\overline{\mathbf{SU}}$ and $\overline{\mathbf{PS}}$ can track part of the strings structure such as the initial part, the ending one or both of them, the minimum strings length, the characters surely contained, etc. Again, any abstract element different from the bottom element ($\bot_{\overline{\mathbf{PR}}}$, $\bot_{\overline{\mathbf{SU}}}$ and $\bot_{\overline{\mathbf{PS}}}$ respectively) represents an infinite set of strings. Even though operations on these domains can be computed in linear time, they suffer from having an infinite height, e.g., given any prefix we can always add a character at the end of it, obtaining a new prefix. However the domains respect the ACC and the termination of the analysis is ensured.

Appendix A.3 reports the complete formalization and proof of soundness of the Prefix domain. Costantini et al. [49] provides this on the Suffix domain, while the one for $\overline{\mathbf{PS}}$ can be obtained by the pointwise application of the operators and semantics on the Cartesian product of $\overline{\mathbf{PR}}$ and $\overline{\mathbf{SU}}$.

Example 4.4. The result of the analysis of the program in Listing 4.1, using $\overline{\mathbf{PR}}$, is given below.

Program point	Variable	$\overline{\mathbf{PR}}$
`pp.2`	`query`	σ_1*
`pp.3`	`lowerBound`	$*$
`pp.4`	`query`	σ_1*
`pp.5`	`query`	σ_1*
`pp.6`	`query`	σ_1*

At `pp.2`, the variable `query` is associated to a state containing the abstraction of σ_1. The latter is approximated by the prefix σ_1*. The input variable `lowerBound` is abstracted by the top element of the $\overline{\mathbf{PR}}$ lattice, i.e., $\top_{\overline{\mathbf{PR}}} = *$. At `pp.4`, the variable `query` is associated to a state containing the concatenation of the abstractions of σ_1, σ_2, `lowerBound` and σ_3, i.e., $\sigma_1* +_{\overline{\mathbf{PR}}} \sigma_2* +_{\overline{\mathbf{PR}}} * +_{\overline{\mathbf{PR}}} \sigma_3* = \sigma_1*$. Then, at `pp.5`, the least upper bound ($\sqcup_{\overline{\mathbf{PR}}}$) between the abstract value of `query` after `pp.2` and after `pp.4` is computed, i.e., $\sigma_1* \sqcup_{\overline{\mathbf{PR}}} \sigma_1* = \sigma_1*$. Finally, at `pp.6`, `query` is associated to a state containing the concatenation of the abstractions of itself after `pp.5` and the strings σ_4, σ_5 and σ_6. Thus, at the end, `query` will for sure begin with σ_1 followed by any possible suffix string $*$.

⬡

4.5 Segmentation Abstract Domain

In Section 3.2 we recalled FunArray, i.e., the array segmentation abstract domain functor by Cousot et al. [57]. Since the order of characters in strings is fundamental in order to track precise information on these values, we instantiate the FunArray abstract domain

for string analysis. In the following, we slightly modify the notation introduced in Section 3.2 to highlight the fact that we are instantiating FunArray over strings.

4.5.1 String Concrete Representation

Let $\varsigma \in \mathrm{R_S} = \mathtt{S} \to \mathbf{S}$ be concrete string environments mapping string variables $\mathtt{s} \in \mathtt{S}$ to their instrumented values $\varsigma(\mathtt{s}) \in \mathbf{S} \triangleq \mathrm{R_V} \times \mathbf{E} \times \mathbf{E} \times (\mathbb{Z} \to (\mathbb{Z} \times \Sigma))$. Thus, a string variable $\mathtt{s}$ is represented by a quadruple $\varsigma(\mathtt{s}) = (\rho, \mathrm{l_s}, \mathrm{h_s}, \mathrm{A_s}) \in \mathbf{S}$, such that:

- $\rho \in \mathrm{R_V} \triangleq \mathtt{X} \to \mathbf{V}$ are concrete scalar variable environments mapping variables $\mathtt{x} \in \mathtt{X}$ to their values $[\![\mathtt{x}]\!]\rho \in \mathbf{V}$.
- $\mathbf{E}$ is the expressions domain (containing constants, scalar variables, mathematical unary and binary operators), and $\mathrm{l_s}, \mathrm{h_s} \in \mathbf{E}$. The values of $\mathrm{l_s}$ and $\mathrm{h_s}$ ($[\![\mathrm{l_s}]\!]\rho$ and $[\![\mathrm{h_s}]\!]\rho$) denote respectively the lower and the upper limit of the string variable $\mathtt{s}$.
- $\mathrm{A_s}$ is a function mapping an index $\mathrm{i} \in [[\![\mathrm{l_s}]\!]\rho, [\![\mathrm{h_s}]\!]\rho)$ to the pair $\langle \mathrm{i}, \mathrm{c} \rangle$ of the index i and the corresponding string character $\mathrm{c} \in \Sigma$.

Example 4.5. Let $\mathtt{s}$ be a string variable initialized to the value `''bunny''`. The concrete value of $\mathtt{s}$ is given by the tuple $\varsigma(\mathtt{s}) = (\rho, 0, 5, \mathrm{A_s})$, where the value of the lower and the upper bound $\mathtt{s}$ is inferred from the context and the function $\mathrm{A_s}$ maps an index $\mathrm{i} \in [0, 5)$ to the pair (index, indexed character value). Thus, the codomain of $\mathrm{A_s}$ is the set $\{(0, \mathtt{'b'}), (1, \mathtt{'u'}), (2, \mathtt{'n'}), (3, \mathtt{'n'}), (4, \mathtt{'y'})\}$.

⬡

4.5.2 Abstract Domain

The Segmentation abstract domain functor $\overline{\mathbf{S}}$ for strings is a function from the parameter abstract domains $\overline{\mathbf{B}}$, $\overline{\mathbf{C}}$ and $\overline{\mathbf{R}}$, where $\overline{\mathbf{B}}$ and $\overline{\mathbf{R}}$ are abstract domain functors in their turn. The segment bound abstract domain functor $\overline{\mathbf{B}}$ is a function of the expression abstract domain $\overline{\mathbf{E}}(\overline{\mathbf{X}})$ which depends on the variable abstract domain $\overline{\mathbf{X}}$, leading to the instantiated segment bound abstract domain $\overline{\mathbf{B}}(\overline{\mathbf{E}}(\overline{\mathbf{X}}))$. $\overline{\mathbf{C}}$ is the string element abstract domain. Finally, $\overline{\mathbf{R}}$ is the variable environment abstract domain functor which depends on $\overline{\mathbf{X}}$ too, leading to the variable environment abstract domain $\overline{\mathbf{R}}(\overline{\mathbf{X}})$. Precisely:

- The variable abstract domain $\overline{\mathbf{X}}$ encodes program variables.
- The variable environment abstract domain functor $\overline{\mathbf{R}}$ depends on $\overline{\mathbf{X}}$, leading to the variable environment abstract domain $\overline{\mathbf{R}}(\overline{\mathbf{X}})$. Elements in $\overline{\mathbf{R}}$ (shorthand for $\overline{\mathbf{R}}(\overline{\mathbf{X}})$) are abstract variable environments $\overline{\rho} \in \overline{\mathbf{R}} = \mathtt{X} \to \overline{\mathbf{V}}$, where the value abstract domain $\overline{\mathbf{V}}$ approximates properties of values in $\mathbf{V}$. $\overline{\mathbf{R}}$ approximates sets of concrete variable environments. Formally, the concretization is $\gamma_{\overline{\mathbf{R}}} : \overline{\mathbf{R}} \to \mathcal{P}(\mathrm{R_V})$, where $\mathrm{R_V} = \mathtt{X} \to \mathbf{V}$ (cf. Section 4.5.1).

- The expression abstract domain functor $\overline{\mathbf{E}}$ depends on $\overline{\mathbf{X}}$, leading to the expression abstract domain $\overline{\mathbf{E}}(\overline{\mathbf{X}})$. Elements in $\overline{\mathbf{E}}$ (shorthand for $\overline{\mathbf{E}}(\overline{\mathbf{X}})$) are symbolic expressions $\overline{\mathbf{E}} \in \overline{\mathbf{E}}(\overline{\mathbf{X}})$, restricted to a canonical normal form, which depend on variables in $\overline{\mathbf{X}}$ (notice that $\bot_{\overline{\mathbf{E}}}, \top_{\overline{\mathbf{E}}} \in \overline{\mathbf{E}}$). $\overline{\mathbf{E}}$ approximates program expressions. Formally, the concretization is $\gamma_{\overline{\mathbf{E}}} : \overline{\mathbf{E}} \to \overline{\mathbf{R}} \to \mathcal{P}(\mathbf{V})$. Moreover, $\overline{\mathbf{E}}$ is equipped with sum ($\oplus_{\overline{\mathbf{E}}}$), subtraction ($\ominus_{\overline{\mathbf{E}}}$) and comparison ($\leq_{\overline{\mathbf{E}}}$) operations. The comparison result depends on the normal form of the expressions. In general, two expressions are said to be *comparable* if and only if their comparison returns `true`.

  The choice of the expression canonical form is let free.

  Notice that in the following examples we will use the expression normal form $\overline{\mathtt{x}} + \mathrm{k}$ such that $\mathrm{k} \in \mathbb{Z}$ and $\overline{\mathtt{x}} \in \overline{\mathbf{X}} \cup \{\mathtt{v}_0\}$ where $\mathtt{v}_0$ is a special variable whose value is always 0 and we omit it.

- The string element abstract domain $\overline{\mathbf{C}}$ approximates sets of pairs (index, indexed string element), such that $\overline{\mathbf{c}} \in \overline{\mathbf{C}}$. Formally, the abstraction is $\alpha_{\overline{\mathbf{C}}} : \mathcal{P}(\mathbb{Z} \times \Sigma) \to \mathcal{P}(\Sigma) \to \overline{\mathbf{C}}$, i.e., elements in $\mathcal{P}(\mathbb{Z} \times \Sigma)$ may be first abstracted to $\mathcal{P}(\Sigma)$ so to perform a non-relational analysis. The concretization is $\gamma_{\overline{\mathbf{C}}} : \overline{\mathbf{C}} \to \mathcal{P}(\mathbb{Z} \times \Sigma)$.

  The choice of $\overline{\mathbf{C}}$ is let free.

- The segment bound abstract domain functor $\overline{\mathbf{B}}$ is a function of the expression abstract domain $\overline{\mathbf{E}}$, leading to the instantiated segment bound abstract domain $\overline{\mathbf{B}}(\overline{\mathbf{E}})$. Elements in $\overline{\mathbf{B}}$ (shorthand for $\overline{\mathbf{B}}(\overline{\mathbf{E}})$) are symbolic intervals $\overline{\mathbf{b}} \in \overline{\mathbf{B}}$, such that $\overline{\mathbf{b}} = [\overline{\mathbf{e}}, \overline{\mathbf{e}}']$, where $\overline{\mathbf{e}}, \overline{\mathbf{e}}' \in \{\overline{\mathbf{E}} \setminus \{\bot_{\overline{\mathbf{E}}}, \top_{\overline{\mathbf{E}}}\}\}$ and $\overline{\mathbf{e}} \leq_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'$. Formally, the concretization is $\gamma_{\overline{\mathbf{B}}} : \overline{\mathbf{B}} \to \mathcal{P}(\mathrm{R_V})$.

Elements of $\overline{\mathbf{S}}$ belong to the set $\overline{\mathbf{S}} = \left\{\left(\overline{\mathbf{C}} \times \overline{\mathbf{B}}\right)^{\mathrm{k}} \mid \mathrm{k} \geq 1\right\} \cup \{\bot_{\overline{\mathbf{S}}}, \top_{\overline{\mathbf{S}}}\}$, where $\bot_{\overline{\mathbf{S}}}$ and $\top_{\overline{\mathbf{S}}}$ are special elements denoting the bottom/top element of $\overline{\mathbf{S}}$. In particular, elements of $\overline{\mathbf{S}}$ are in the form of $\overline{\mathbf{s}} = \overline{\mathbf{c}}_1\overline{\mathbf{b}}_1 \ldots \overline{\mathbf{c}}_\mathrm{n}\overline{\mathbf{b}}_\mathrm{n}$, where a segment $\overline{\mathbf{c}}_\mathrm{i}\overline{\mathbf{b}}_\mathrm{i}$ (with $\overline{\mathbf{b}}_\mathrm{i} = [\overline{\mathbf{e}}_\mathrm{i}, \overline{\mathbf{e}}_\mathrm{i}']$) abstracts a sequence of equal characters whose length goes from $\overline{\mathbf{e}}_\mathrm{i}$ to $\overline{\mathbf{e}}_\mathrm{i}'$.

Example 4.6. Consider the string variable of the Example 4.5. Its value in the Segmentation abstract domain $\overline{\mathbf{S}}$ is 'b' $[1,1]$ 'u' $[1,1]$ 'n' $[2,2]$ 'y' $[1,1]$, where $\overline{\mathbf{C}}$ is the constant propagation domain for characters.

⬡

Before defining the join and the meet operators of the Segmentation domain we present the following helping procedures: align (cf. Algorithm 3) and fold (cf. Algorithm 4).

Algorithm 3 align procedure

Input: $\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2$

Output: $\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2$ (possibly modified) or `error`

1: $\mathrm{i} \leftarrow 1$
2: $\mathrm{k}_1 \leftarrow 0; \mathrm{k}_2 \leftarrow 0$
3: **if** $\overline{\mathbf{s}}_1 \neq \bot_{\overline{\mathbf{S}}} \wedge \overline{\mathbf{s}}_2 \neq \bot_{\overline{\mathbf{S}}}$ **then**
4: $\quad$ **while** $\mathrm{i} \leq \mathsf{numSeg}(\overline{\mathbf{s}}_1) + \mathrm{k}_1 \wedge \mathrm{i} \leq \mathsf{numSeg}(\overline{\mathbf{s}}_2) + \mathrm{k}_2$ **do**
5: $\quad\quad$ **if** $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[\mathrm{i}] =_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[\mathrm{i}]$ **then**
6: $\quad\quad\quad$ i++
7: $\quad\quad$ **else if** $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[\mathrm{i}] \leq_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[\mathrm{i}]$ **then**
8: $\quad\quad\quad$ $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[\mathrm{i}] \leftarrow \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[\mathrm{i}]$
9: $\quad\quad\quad$ $\overline{\mathbf{s}}_1[\mathrm{i}].\mathrm{append}(\overline{\mathbf{c}}.\overline{\mathbf{s}}_1[\mathrm{i}][0, \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[\mathrm{i}] \ominus_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[\mathrm{i}]])$
10: $\quad\quad\quad$ k_1++, i++
11: $\quad\quad$ **else if** $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[\mathrm{i}] \leq_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[\mathrm{i}]$ **then**
12: $\quad\quad\quad$ $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[\mathrm{i}] \leftarrow \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[\mathrm{i}]$
13: $\quad\quad\quad$ $\overline{\mathbf{s}}_2[\mathrm{j}].\mathrm{append}(\overline{\mathbf{c}}.\overline{\mathbf{s}}_2[\mathrm{i}][0, \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[\mathrm{i}] \ominus_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[\mathrm{i}]])$
14: $\quad\quad\quad$ k_2++, i++
15: $\quad\quad$ **else**
16: $\quad\quad\quad$ **return** `error` ($\overline{\mathbf{s}}_1$ and $\overline{\mathbf{s}}_2$ can not be aligned)
17: **else**
18: $\quad$ **return** `error` ($\overline{\mathbf{s}}_1$ and $\overline{\mathbf{s}}_2$ can not be aligned)
19: **return** $\overline{\mathbf{s}}_1$, $\overline{\mathbf{s}}_2$

Algorithm 3 aligns two segmentations if they are both different from the bottom element of $\overline{\mathbf{S}}$ and comparable. The comparability is actually restricted to the segment bound upper limits of each pair of corresponding segments under consideration during the alignment procedure. Let $\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2 \in \overline{\mathbf{S}}$ different from $\bot_{\overline{\mathbf{S}}}$ (line 3), the alignment procedure starts analysing the two segmentations from their leftmost segments, i.e., $\overline{\mathbf{s}}_1[1]$ and $\overline{\mathbf{s}}_2[1]$, and continues along their number of segments (line 4), i.e., $\mathsf{numSeg}(\overline{\mathbf{s}}_1)$ and $\mathsf{numSeg}(\overline{\mathbf{s}}_2)$, which may change during the procedure, if all the corresponding segment bound upper limits under consideration are comparable. If this is the case and the segment bound upper limits are equal then the procedure moves to the next segments, otherwise if one of the two segment bound upper limits is greater or equal than the other, the first is modified according to the latter segment bound upper limit (line 10 o line 14) and the "remaining part" is appended to the previously modified segment (line 11 or line 15). In the case in which one or both the input segmentations are equals to $\bot_{\overline{\mathbf{S}}}$ or if during the alignment procedure two corresponding segment bound upper limits are not comparable, the algorithm stops and it returns an error message.

Example 4.7. Consider the segmentations:

- $\overline{\mathbf{s}}_1$ = 'a' $[0, 4]$ 'b' $[1, 1]$ 'c' $[0, \mathtt{x}]$
- $\overline{\mathbf{s}}_2$ = 'a' $[2, 5 + \mathtt{x}]$

where the segment predicate abstract domain $\overline{\mathbf{C}}$ is the constant propagation domain for characters and x is an integer variable whose value is greater than or equal to 0. Algorithm 3 on $\overline{\mathbf{s}}_1$ and $\overline{\mathbf{s}}_2$ is applied as follows: starting from i equal to 1, we enter into the loop. The first segment bound upper limit of $\overline{\mathbf{s}}_1$ is strictly smaller than the first segment bound upper limit of $\overline{\mathbf{s}}_2$ (for any value of x), i.e., $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[1] <_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[1]$ (line 11), then the value of $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[1]$ is assigned to $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[1]$ (line 12), and the segment $\overline{\mathbf{c}}.\overline{\mathbf{s}}_2[1][0, \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[1] \ominus_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[1]]$ is appended to $\overline{\mathbf{s}}_2[1]$ (line 13). Thus, after the first iteration of the for loop, $\overline{\mathbf{s}}_1$ is unchanged and $\overline{\mathbf{s}}_2$ is updated. Precisely, we obtain:

- $\overline{\mathbf{s}}_1$ = 'a' $[0, 4]$ 'b' $[1, 1]$ 'c' $[0, \mathtt{x}]$
- $\overline{\mathbf{s}}_2$ = 'a' $[2, 4]$ 'a' $[0, 1 + \mathrm{x}]$

The counter i is increased by 1 and, since $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[2] \leq_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[2]$ (line 11), to the second segment bound upper limit of $\overline{\mathbf{s}}_2$ is assigned the value of $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[2]$ (line 12) and the segment $\overline{\mathbf{c}}.\overline{\mathbf{s}}_2[2][0, \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[2] \ominus_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[2]]$ is appended to $\overline{\mathbf{s}}_2[2]$ (line 13). After the second iteration of the for loop, $\overline{\mathbf{s}}_1$ is still unchanged and $\overline{\mathbf{s}}_2$ is updated as follows:

- $\overline{\mathbf{s}}_1$ = 'a' $[0, 4]$ 'b' $[1, 1]$ 'c' $[0, \mathtt{x}]$
- $\overline{\mathbf{s}}_2$ = 'a' $[2, 4]$ 'a' $[0, 1]$ 'a' $[0, \mathrm{x}]$

Again the counter i is increased by 1. The third segment bound upper limits of $\overline{\mathbf{s}}_1$ and $\overline{\mathbf{s}}_2$ are equal (line 5), i.e., $\overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_1[3] =_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'.\overline{\mathbf{b}}.\overline{\mathbf{s}}_2[3]$, i is increased by 1 but, now, they exceed the number of segments of both $\overline{\mathbf{s}}_1$ and $\overline{\mathbf{s}}_2$, thus the final result is the pair of segmentations obtained after the second iteration of the while loop.

⬡

Algorithm 4 fold procedure

Input: $\overline{\mathrm{s}}$

Output: $\overline{\mathrm{s}}$ (possibly modified)

```
1: i ← 1
2: while i ≤ numSeg(s̄) do
3:     if c̄.s̄[i] =_C̄ c̄.s̄[i+1] ∧ ē.b̄.s̄[i+1] =_Ē 0 then
4:         b̄.s̄[i] ← b̄.s̄[i] + b̄.s̄[i+1]
5:         s̄.remove(s̄[i+1])
6:     else
7:         i++
8: return s̄
```

The alignment procedure (cf. Algorithm 3) is used to facilitate the computation of the join $\sqcup_{\overline{\mathbf{S}}}$ and the meet $\sqcap_{\overline{\mathbf{S}}}$. The latter may lead to segmentations which need to be taken back to their normal form, i.e., where there are no consecutive segments with the same segment abstract predicate $\overline{\mathbf{c}}$. Thus, Algorithm 4 brings a segmentation to its normal form and returns it, merging consecutive segments sharing the same segment abstract predicate (line 4). Join and meet operators, and partial order of $\overline{\mathbf{S}}$ are defined as follows:

- $\sqcap_{\overline{\mathbf{S}}}$ represents the meet operator between two string segmentations. The meet between two segmentations $\overline{\mathbf{s}}_1$ and $\overline{\mathbf{s}}_2$ is computed on their alignment if *i*) $\mathsf{align}(\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2)$ does not raise an error, and *ii*) one of the two aligned segmentations has a number of segments greater than the other, the exceeding segments have to be possibly empty, i.e., their lower limit has to be equal to 0; otherwise $\bot_{\overline{\mathbf{S}}}$ is returned. Formally, let $\overline{\mathbf{s}}_1^{'}$ and $\overline{\mathbf{s}}_2^{'}$ be the results of $\mathsf{align}(\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2)$ then:

$$\overline{\mathbf{s}}_1 \sqcap_{\overline{\mathbf{S}}} \overline{\mathbf{s}}_2 = \begin{cases} \mathsf{fold}(\overline{\mathbf{s}}_1^{'} \sqcap_{\overline{\mathbf{S}}} \overline{\mathbf{s}}_2^{'}) & \text{see (1)} \\ \bot_{\overline{\mathbf{S}}} & \text{otherwise} \end{cases}$$

  (1) $\overline{\mathbf{s}}_1 \sqcap_{\overline{\mathbf{S}}} \overline{\mathbf{s}}_2 = \mathsf{fold}(\overline{\mathbf{s}}_1^{'} \sqcap_{\overline{\mathbf{S}}} \overline{\mathbf{s}}_2^{'})$ if the following conditions hold:

  * $\mathsf{align}(\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2) \neq \mathtt{error}$
  * given $\mathrm{j}, \mathrm{k} \in [1, 2]$ with $\mathrm{j} \neq \mathrm{k}$, if $\mathsf{numSeg}(\overline{\mathbf{s}}_{\mathrm{j}}^{'}) > \mathsf{numSeg}(\overline{\mathbf{s}}_{\mathrm{k}}^{'})$ then,

$$\forall \mathrm{i} \in [\mathsf{numSeg}(\overline{\mathbf{s}}_{\mathrm{k}}^{'}) + 1, \mathsf{numSeg}(\overline{\mathbf{s}}_{\mathrm{j}}^{'})] \text{ of } \overline{\mathbf{s}}_{\mathrm{j}}^{'} : \overline{\mathbf{e}}_{\mathrm{i}} =_{\overline{\mathbf{E}}} 0$$

  The meet between two aligned segmentations is performed such that the string character abstract domain meet ($\sqcap_{\overline{\mathbf{C}}}$) and the bound abstract domain meet ($\sqcap_{\overline{\mathbf{B}}}$) are applied segment-wise. Notice that if the number of segments of $\overline{\mathbf{s}}_1^{'}$ ($\mathsf{numSeg}(\overline{\mathbf{s}}_1^{'})$) is strictly greater than the number of segments of $\overline{\mathbf{s}}_2^{'}$ ($\mathsf{numSeg}(\overline{\mathbf{s}}_2^{'})$) then all the exceeding segments of $\overline{\mathbf{s}}_1^{'}$ are not preserved by $\sqcap_{\overline{\mathbf{S}}}$ (the viceversa is similar). Finally, the fold of the meet between $\overline{\mathbf{s}}_1^{'}$ and $\overline{\mathbf{s}}_2^{'}$ is computed.

- $\sqcup_{\overline{\mathbf{S}}}$ represents the join operator between two string segmentations. The join between two segmentations $\overline{\mathbf{s}}_1$ and $\overline{\mathbf{s}}_2$ is computed on their alignment if $\mathsf{align}(\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2)$ does not raise an error; if only one of the two segmentations is equal to $\bot_{\overline{\mathbf{S}}}$ then their join returns the one which is different from the bottom element; if both $\overline{\mathbf{s}}_1$ and $\overline{\mathbf{s}}_2$ are the bottom element then their join returns $\bot_{\overline{\mathbf{S}}}$; otherwise $\top_{\overline{\mathbf{S}}}$ is returned. Formally, let $\overline{\mathbf{s}}_1^{'}$ and $\overline{\mathbf{s}}_2^{'}$ be the results of $\mathsf{align}(\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2)$ (cf. Algorithm 3) then:

$$\overline{s}_1 \sqcup_{\overline{S}} \overline{s}_2 = \begin{cases} \mathsf{fold}(\overline{s}_1^{'} \sqcup_{\overline{S}} \overline{s}_2^{'}) & \text{if } \mathsf{align}(\overline{s}_1, \overline{s}_2) \neq \texttt{error} \\ \overline{s}_1 & \text{if } \overline{s}_2 = \bot_{\overline{S}} \wedge \overline{s}_1 \neq \bot_{\overline{S}} \\ \overline{s}_2 & \text{if } \overline{s}_1 = \bot_{\overline{S}} \wedge \overline{s}_2 \neq \bot_{\overline{S}} \\ \bot_{\overline{S}} & \text{if } \overline{s}_1 = \bot_{\overline{S}} \wedge \overline{s}_2 = \bot_{\overline{S}} \\ \top_{\overline{S}} & \text{otherwise} \end{cases}$$

The join between two aligned segmentations is performed such that the string character abstract domain join ($\sqcup_{\overline{C}}$) and the bound abstract domain join ($\sqcup_{\overline{B}}$) are applied segment-wise. If the number of segments of $\overline{s}_1^{'}$ is strictly greater than the number of segments of $\overline{s}_2^{'}$ then all the exceeding segments of $\overline{s}_1^{'}$ are preserved by $\sqcup_{\overline{S}}$, but their segment bound lower limit is set to 0 (the viceversa is similar).

- Let $\overline{s}_1$ and $\overline{s}_2$ be two abstract values in the Segmentation domain. The partial order on $\overline{S}$ is defined as follows: $\forall \overline{s} \in \overline{S} : \bot_{\overline{S}} \sqsubseteq_{\overline{S}} \overline{s} \wedge \overline{s} \sqsubseteq_{\overline{S}} \top_{\overline{S}}$. Otherwise, if both $\overline{s}_1$ and $\overline{s}_2$ are different from $\bot_{\overline{S}}$ and $\top_{\overline{S}}$ then, $\overline{s}_1 \sqsubseteq_{\overline{S}} \overline{s}_2 \Leftrightarrow \overline{s}_1 \sqcup_{\overline{S}} \overline{s}_2 = \overline{s}_2$

Concretization

The concretization function on the segmentation abstract domain $\gamma_{\overline{S}} : \overline{S} \to \overline{R} \to \mathcal{P}(S)$ maps an abstract element to a set of strings as follows: $\gamma_{\overline{S}}(\bot_{\overline{S}}) = \emptyset$, $\gamma_{\overline{S}}(\top_{\overline{S}}) = S$, while it is the set of all possible sequences of characters derivable from a segmentation abstract predicate. The formalization below follows the one defined in Section 11.3 of [57]. Let $\gamma_{\overline{R}} : \overline{R} \to \mathcal{P}(R_v)$ be the concretization function for the variable abstract domain and $\gamma_{\overline{C}} : \overline{C} \to \mathcal{P}(\mathbb{Z} \times \Sigma)$ be the concretization function for the string elements abstract domain (cf. Section 4.5.2). $\gamma^{\star}_{\overline{S}}$ denotes the concretization of a generic segment $\overline{c}\overline{b}$ where $\overline{b} = [\overline{e}, \overline{e}']$, formally:

$$\gamma^{\star}_{\overline{S}}(\overline{c}\overline{b})\overline{\rho} = \{(\rho, l, h, A) \mid \rho \in \gamma_{\overline{R}}(\overline{\rho}) \wedge [\![l]\!]\rho = 0 \ \wedge \exists e \in [[\![\overline{e}]\!]\rho, [\![\overline{e}']\!]\rho] : [\![h]\!]\rho = e \wedge \forall i \in [0, e-1) : A(i) \in \gamma_{\overline{C}}(\overline{c})\}$$

where $\overline{\rho} \in \overline{R}$. Then, the concretization function of a string segmentation is as follows:

$$\gamma_{\overline{S}}(\overline{c}_1\overline{b}_1 ... \overline{c}_n\overline{b}_n) = \{(\rho, l, h, A) \in \mathop{+}_{i=1}^{n}{}_{S} \gamma^{*}_{\overline{S}}(\overline{c}_i\overline{b}_i) \mid [\![l]\!]\rho = 0 \ \wedge [\![h]\!]\rho = \sum_{i=1}^{n} e_i\}$$

and $\gamma_{\overline{S}}(\bot_{\overline{S}}) = \emptyset$

Notice that a segmentation abstract predicate is *valid* if the upper bounds of segments that contain the bottom element in an abstract domain $\overline{C}$ are possibly empty, otherwise a string segmentation is *invalid*. The concretization function of an *invalid* segmentation maps the latter abstract value to the empty-set.

Theorem 4.1. Let $\overline{X} \subseteq \overline{\mathbf{S}}$ such that all elements in $\overline{X}$ can be aligned and their meet does not result in an invalid segmentation. The following inference chain holds:

$$\gamma_{\overline{\mathbf{S}}}\Big(\bigsqcap_{\substack{\overline{\mathbf{S}} \\ \overline{s} \in \overline{X}}} \overline{s}\Big) = \gamma_{\overline{\mathbf{S}}}(\overline{s}^{\star}) \qquad \text{by definition of } \sqcap_{\overline{\mathbf{S}}}$$

where $\overline{s}^{\star}$ denotes the result of the meet between the segmentations $\overline{s}$ in $\overline{X}$

$$= \{(\rho, l, h, A) \in \mathop{\mathbin{+\!\!\!-}}_{\substack{\mathbf{S} \\ i=1}}^{n} \gamma^{*}_{\overline{\mathbf{S}}}(\overline{c}_i \overline{b}_i) \mid [\![l]\!]\rho = 0 \ \wedge [\![h]\!]\rho = \sum_{i=1}^{n} e_i\} \qquad \text{by definition of } \gamma_{\overline{\mathbf{S}}}$$

$$= \bigcap_{\overline{s} \in \overline{X}} \{(\rho, l, h, m) \in \mathop{\mathbin{+\!\!\!-}}_{\substack{\mathbf{S} \\ i=1}}^{n} \gamma^{*}_{\overline{\mathbf{S}}}(\overline{c}_i \overline{b}_i) \mid [\![l]\!]\rho = 0 \ \wedge [\![h]\!]\rho = \sum_{i=1}^{n} e_i\}$$

$$= \bigcap_{\overline{s} \in \overline{X}} \gamma_{\overline{\mathbf{S}}}(\overline{s}) \qquad \text{by definition of } \gamma_{\overline{\mathbf{S}}}$$

□

Observe that if the hypotheses of Theorem 4.1 are not satisfied, i.e., if either the abstract elements in $\overline{X}$ can not be aligned or their meet leads to an invalid segmentation, then $\gamma_{\overline{\mathbf{S}}}\Big(\bigsqcap_{\overline{s} \in \overline{X}}^{\overline{\mathbf{S}}} \overline{s}\Big) = \gamma_{\overline{\mathbf{S}}}(\bot_{\overline{\mathbf{S}}}) = \emptyset$, and $\bigcap_{\overline{s} \in \overline{X}} \gamma_{\overline{\mathbf{S}}}(\overline{s}) = \emptyset$

Abstraction

Let $X \in \mathcal{P}(\mathbf{S})$ be a set of concrete string values. The abstraction function on the segmentation abstract domain $\alpha_{\overline{\mathbf{S}}}$ maps X to $\bot_{\overline{\mathbf{S}}}$ in the case in which X is equal to the empty set, otherwise to the segmentation that optimally over-approximates values in X.

To summarize, $\overline{\mathbf{S}}$ abstracts strings by a sequence of segments in an abstract domain $\overline{\mathbf{C}}$ and a segment bound in an abstract domain $\overline{\mathbf{B}}$. The segments are computed by how the string content is manipulated. The value of a string segmentation abstract predicate depends on three parameters: the abstract domain representing symbolic segment bound expressions, the domain abstracting the pairs (index, character), and the abstract domain assigning values to segment bound expressions [57]. Thus, $\overline{\mathbf{S}}$ can be instantiated with different abstract domains, achieving different levels of precision and costs of the analysis. Notice that, the segmentation abstract domain has infinite height as a string may have infinitely symbolic segments and a segment might take successive strictly increasing abstract values. Therefore, to guarantee the convergence of the analysis, a widening operator needs to be defined. The widening operator for the Segmentation domain can be defined similarly to the one presented by Cousot et al. in [57] and, informally, if during the alignment procedure between two segmentations the number of segments exceeds a certain threshold then, the widening transforms those segmentations into $\top_{\overline{\mathbf{S}}}$. Notice that preciser widening operator may be defined and eventually a narrowing operator to improve the accuracy of the analysis.

4.5.3 Abstract Semantics

In Section 4.2 we recalled the syntax of some operations of interest (i.e., `newString`, `concat`, `substring`, and `contains`) presented in [49]. Moreover, in Section 4.3.2 we reported their concrete semantics. Below, we formally define their approximation in the string segmentation abstract domain $\overline{\mathbf{S}}$. The semantics operators $\mathfrak{S}_{\overline{\mathbf{S}}}$ and $\mathfrak{B}_{\overline{\mathbf{S}}}$ are the abstract counterparts of $\mathfrak{S}$ and $\mathfrak{B}$ respectively (cf. Section 4.3.2).

- New string

  Let σ be a string value. The semantics $\mathfrak{S}_{\overline{\mathbf{S}}}$, when applied to `new String`$(\sigma)$, returns the segmentation of the string constant σ. Formally,

  $$\mathfrak{S}_{\overline{\mathbf{S}}}[\![\texttt{new String}(\sigma)]\!]() = \overline{\mathbf{s}}$$

  $\overline{\mathbf{s}}$ abstracts, in an abstract domain $\overline{\mathbf{C}}$, the sequences of equal characters in σ. Notice that each segment bound of $\overline{\mathbf{s}}$ will have equal lower and upper limits, in an abstract domain $\overline{\mathbf{E}}$.

- String concatenation

  Let $\overline{\mathbf{s}}_1$ (i.e., $\overline{\mathbf{c}}_{1,1}\overline{\mathbf{b}}_{1,1}\ldots\overline{\mathbf{c}}_{1,n},\overline{\mathbf{b}}_{1,n}$) and $\overline{\mathbf{s}}_2$ (i.e., $\overline{\mathbf{c}}_{2,1}\overline{\mathbf{b}}_{2,1}\ldots\overline{\mathbf{c}}_{2,n},\overline{\mathbf{b}}_{2,n}$) be two abstract values in $\overline{\mathbf{S}}$. The semantics $\mathfrak{S}_{\overline{\mathbf{S}}}$, when applied to `concat`$(\overline{\mathbf{s}}_1,\overline{\mathbf{s}}_2)$ it returns the concatenation of the two input segmentations. Formally,

  $$\mathfrak{S}_{\overline{\mathbf{S}}}[\![\texttt{concat}]\!](\overline{\mathbf{s}}_1,\overline{\mathbf{s}}_2) = \overline{\mathbf{s}}_1 +_{\overline{\mathbf{S}}} \overline{\mathbf{s}}_2$$

  Note that if the last segment of $\overline{\mathbf{s}}_1$ and the first segment of $\overline{\mathbf{s}}_2$ share the same abstract character then these segments are unified. Precisely, given $\overline{\mathbf{c}}_{1,1}\overline{\mathbf{b}}_{1,1}...\overline{\mathbf{c}}_{1,n},\overline{\mathbf{b}}_{1,n}$ and $\overline{\mathbf{c}}_{2,1}\overline{\mathbf{b}}_{2,1}\ldots\overline{\mathbf{c}}_{2,n},\overline{\mathbf{b}}_{2,n}$, if $\overline{\mathbf{c}}_{1,n} =_{\overline{\mathbf{C}}} \overline{\mathbf{c}}_{2,1}$ then their concatenation is equal to

  $$\overline{\mathbf{c}}_{1,1}\overline{\mathbf{b}}_{1,1}\ldots\overline{\mathbf{c}}[\overline{\mathbf{e}}_{1,n} \oplus_{\overline{\mathbf{E}}} \overline{\mathbf{e}}_{2,1}, \overline{\mathbf{e}}'_{1,n} \oplus_{\overline{\mathbf{E}}} \overline{\mathbf{e}}'_{2,1}]\overline{\mathbf{c}}_{2,2}\overline{\mathbf{b}}_{2,2}\ldots\overline{\mathbf{c}}_{2,n},\overline{\mathbf{b}}_{2,n}$$

  where $\overline{\mathbf{c}}$ represents the character contained in $\overline{\mathbf{c}}_{1,n}$ and $\overline{\mathbf{c}}_{2,1}$.

- Substring

  Let $\overline{\mathbf{s}}$ (i.e., $\overline{\mathbf{c}}_1\overline{\mathbf{b}}_1\ldots\overline{\mathbf{c}}_n\overline{\mathbf{b}}_n$) be an abstract value in $\overline{\mathbf{S}}$. The semantics $\mathfrak{S}_{\overline{\mathbf{S}}}$, when applied to $\texttt{substring}^{\texttt{e}}_{\texttt{b}}(\overline{\mathbf{s}})$ returns the subsegmentation of $\overline{\mathbf{s}}$ from the segment whose associated interval on the indexes that may refer to it contains `b` to the segment whose associated interval on the indexes that may refer to it contains `e`, otherwise it returns $\top_{\overline{\mathbf{S}}}$. Formally,

$$\mathfrak{S}_{\overline{\mathbf{S}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!](\overline{\mathsf{s}}) =$$

$$\begin{cases} \overline{\mathsf{c}}_{\mathrm{k}}\overline{\mathsf{b}}_{\mathrm{k}} \ldots \overline{\mathsf{c}}_{\mathrm{j}}\overline{\mathsf{b}}_{\mathrm{j}} & \text{if } \exists \langle \overline{\mathsf{c}}_{\mathrm{k}}\overline{\mathsf{b}}_{\mathrm{k}}, [\mathrm{i}, \mathrm{i}']_{\mathrm{k}} \rangle, \langle \overline{\mathsf{c}}_{\mathrm{j}}\overline{\mathsf{b}}_{\mathrm{j}}, [\mathrm{i}, \mathrm{i}']_{\mathrm{j}} \rangle \in \mathsf{index}(\overline{\mathsf{s}}) : \\ & \quad \mathrm{k} = \mathsf{min}(\{\mathrm{i} \mid \langle \overline{\mathsf{c}}_{\mathrm{i}}\overline{\mathsf{b}}_{\mathrm{i}}, [\mathrm{i}, \mathrm{i}']_{\mathrm{i}} \rangle \in \mathsf{index}(\overline{\mathsf{s}}) \wedge \mathsf{b} \in [\mathrm{i}, \mathrm{i}']_{\mathrm{i}}\}) \wedge \\ & \quad\quad \mathrm{j} = \mathsf{max}(\{\mathrm{i} \mid \langle \overline{\mathsf{c}}_{\mathrm{i}}\overline{\mathsf{b}}_{\mathrm{i}}, [\mathrm{i}, \mathrm{i}']_{\mathrm{i}} \rangle \in \mathsf{index}(\overline{\mathsf{s}}) \wedge \mathsf{e} \in [\mathrm{i}, \mathrm{i}']_{\mathrm{i}}\}) \\ \top_{\overline{\mathbf{S}}} & \text{otherwise} \end{cases}$$

where $\mathsf{index}(\overline{\mathsf{s}})$ is the function which associates each segment of $\overline{\mathsf{s}}$ to the interval of the index abstract values that may refer to it, i.e., $\mathsf{index}(\overline{\mathsf{s}}) = \{\langle \overline{\mathsf{c}}_{\mathrm{i}}\overline{\mathsf{b}}_{\mathrm{i}}, [\mathrm{i}, \mathrm{i}']_{\mathrm{i}} \rangle \mid \mathrm{i} \in [1, \mathsf{numSeg}(\overline{\mathsf{s}})]\}$. For instance, consider the segmentation $\overline{\mathsf{s}} =$ 'a' $[0, 2]$ 'b' $[4, 4]$ 'c' $[0, 2]$, $\mathsf{index}(\overline{\mathsf{s}}) = \{\langle$'a' $[0, 2], [0, 1]\rangle, \langle$'b' $[4, 4], [0, 5]\rangle, \langle$'c' $[0, 2], [4, 7]\rangle\}$ and the subsegmentation of $\overline{\mathsf{s}}$ from the index 2 to the index 5 is 'b' $[4, 4]$ 'c' $[0, 2]$.

- Is contained

  Let $\overline{\mathsf{s}}$ (i.e., $\overline{\mathsf{c}}_1\overline{\mathsf{b}}_1 ... \overline{\mathsf{c}}_{\mathrm{n}}\overline{\mathsf{b}}_{\mathrm{n}}$) be an abstract value in $\overline{\mathbf{S}}$. The semantics $\mathfrak{B}_{\overline{\mathbf{S}}}$, when applied to $\texttt{contains}_{\mathsf{c}}(\overline{\mathsf{s}})$ it returns `true` if there exists a segment abstract predicate in $\overline{\mathsf{s}}$ which approximates only the character `c` and its segment bound lower limit is different from zero, `false` if does not exist a segment abstract predicate in $\overline{\mathsf{s}}$ which approximates only the character `c` and its segment bound lower limit is different from zero, otherwise it returns $\top_{\mathbf{B}}$. Formally,

$$\mathfrak{B}_{\overline{\mathbf{S}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\overline{\mathsf{s}}) = \begin{cases} \texttt{true} & \text{if } \exists \overline{\mathsf{c}}_{\mathrm{i}}[\overline{\mathsf{e}}_{\mathrm{i}}, \overline{\mathsf{e}}_{\mathrm{i}}^{'}] \in \overline{\mathsf{s}} : \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathsf{c}}_{\mathrm{i}}) = \{\mathsf{c}\} \wedge \overline{\mathsf{e}}_{\mathrm{i}} \neq_{\overline{\mathbf{E}}} 0 \\ \texttt{false} & \text{if } \nexists \overline{\mathsf{c}}_{\mathrm{i}}[\overline{\mathsf{e}}_{\mathrm{i}}, \overline{\mathsf{e}}_{\mathrm{i}}^{'}] \in \overline{\mathsf{s}} : \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathsf{c}}_{\mathrm{i}}) = \{\mathsf{c}\} \wedge \overline{\mathsf{e}}_{\mathrm{i}} \neq_{\overline{\mathbf{E}}} 0 \\ \top_{\mathbf{B}} & \text{otherwise} \end{cases}$$

  where $\mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathsf{c}}) = \{\mathrm{c} : \langle \mathrm{i}, \mathrm{c} \rangle \in \gamma_{\overline{\mathbf{C}}}(\overline{\mathsf{c}})\}$.

4.5.4 Soundness

We prove the soundness of the Segmentation abstract semantics defined above.

Theorem 4.2. $\mathfrak{S}_{\overline{\mathbf{S}}}$ is a sound over-approximations of $\mathfrak{S}$. Formally,

$$\gamma_{\overline{\mathbf{S}}}(\mathfrak{S}_{\overline{\mathbf{S}}}[\![\texttt{Stm}]\!](\overline{\mathsf{s}})) \supseteq \{\mathfrak{S}[\![\texttt{Stm}]\!](\sigma) \mid \sigma \in \gamma_{\overline{\mathbf{S}}}(\overline{\mathsf{s}})\}$$

Proof.
We prove the soundness separately for each operator.

- Consider the `new String` operator and let σ be a sequence of characters. We have to prove that,

$$\gamma_{\overline{\mathbf{S}}}(\mathfrak{S}_{\overline{\mathbf{S}}}[\![\texttt{new String}(\sigma)]\!]()) \supseteq \{\mathfrak{S}[\![\texttt{new String}(\sigma)]\!]()\}$$

This holds by the definitions of $\mathfrak{S}_{\overline{\mathbf{S}}}$ and $\gamma_{\overline{\mathbf{S}}}$.

- Consider the `concat` operation and let $\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2 \in \overline{\mathbf{S}}$. We have to prove that,

$$\gamma_{\overline{\mathbf{S}}}(\mathfrak{S}_{\overline{\mathbf{S}}}[\![\mathtt{concat}]\!](\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2)) \supseteq \{\mathfrak{S}[\![\mathtt{concat}]\!](\sigma_1, \sigma_2) \mid \sigma_1 \in \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}_1) \wedge \sigma_2 \in \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}_2)\}$$

  Let σ be a generic element in $\{\mathfrak{S}[\![\mathtt{concat}]\!](\sigma_1, \sigma_2) \mid \sigma_1 \in \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}_1) \wedge \sigma_2 \in \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}_2)\}$, it represents any possible concatenation between a string taken from $\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}_1)$ and a string taken from $\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}_2)$. Therefore σ belongs to $\gamma_{\overline{\mathbf{S}}}(\mathfrak{S}_{\overline{\mathbf{S}}}[\![\mathtt{concat}]\!](\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2))$ because $\mathfrak{S}_{\overline{\mathbf{S}}}[\![\mathtt{concat}]\!](\overline{\mathbf{s}}_1, \overline{\mathbf{s}}_2) = \overline{\mathbf{s}}_1 +_{\overline{\mathbf{S}}} \overline{\mathbf{s}}_2$, by definition of $\mathfrak{S}_{\overline{\mathbf{S}}}$. Indeed $\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}_1 +_{\overline{\mathbf{S}}} \overline{\mathbf{s}}_2)$ contains all the strings which come from the concatenation of $\overline{\mathbf{s}}_1$ and $\overline{\mathbf{s}}_2$, by definition of $\gamma_{\overline{\mathbf{S}}}$.

- Consider the `substring` operation and let $\overline{\mathbf{s}} \in \overline{\mathbf{S}}$. we have to prove that,

$$\gamma_{\overline{\mathbf{S}}}(\mathfrak{S}_{\overline{\mathbf{S}}}[\![\mathtt{substring}^{\mathtt{e}}_{\mathtt{b}}]\!](\overline{\mathbf{s}})) \supseteq \{\mathfrak{S}[\![\mathtt{substring}^{\mathtt{e}}_{\mathtt{b}}]\!](\sigma) \mid \sigma \in \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})\}$$

  We have two possible cases:

  - If both `b` and `e` belong to an interval of the indexes associated to each segment of a segmentation $\overline{\mathbf{s}}$, then $\mathfrak{S}_{\overline{\mathbf{S}}}[\![\mathtt{substring}^{\mathtt{e}}_{\mathtt{b}}]\!](\overline{\mathbf{s}})$ returns the subsegmentation of $\overline{\mathbf{s}}$ from the leftmost segment whose associated indexes interval contains `b` to the rightmost segment whose associated indexes interval contains `e`, by definition of $\mathfrak{S}_{\overline{\mathbf{S}}}$. This means that any of the concrete strings in $\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})$ contain the substring from the index `b` to the index `e`, by definition of $\mathfrak{S}$ and it follows that any of the substrings in $\{\mathfrak{S}[\![\mathtt{substring}^{\mathtt{e}}_{\mathtt{b}}]\!](\sigma) \mid \sigma \in \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})\}$ is also contained in $\gamma_{\overline{\mathbf{S}}}(\mathfrak{S}_{\overline{\mathbf{S}}}[\![\mathtt{substring}^{\mathtt{e}}_{\mathtt{b}}]\!](\overline{\mathbf{s}}))$.
  - Otherwise $\mathfrak{S}_{\overline{\mathbf{S}}}[\![\mathtt{substring}^{\mathtt{e}}_{\mathtt{b}}]\!](\overline{\mathbf{s}})$ returns $\top_{\overline{\mathbf{S}}}$ that over-approximates any possible results of the concrete semantics.

□

Theorem 4.3. $\mathfrak{B}_{\overline{\mathbf{S}}}$ is a sound over-approximations of $\mathfrak{B}$. Formally,

$$\gamma_{\overline{\mathbf{S}}}(\mathfrak{B}_{\overline{\mathbf{S}}}[\![\mathtt{Stm}]\!](\overline{\mathbf{s}})) \supseteq \{\mathfrak{B}[\![\mathrm{Stm}]\!](\sigma) \mid \sigma \in \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})\}$$

Proof.
Consider the `contains` operator and let $\overline{\mathbf{s}} \in \overline{\mathbf{S}}$. We have to prove that,

$$\gamma_{\overline{\mathbf{S}}}(\mathfrak{B}_{\overline{\mathbf{S}}}[\![\mathtt{contains}_{\mathtt{c}}]\!](\overline{\mathbf{s}})) \supseteq \{\mathfrak{B}[\![\mathtt{contains}_{\mathtt{c}}]\!](\sigma) \mid \sigma \in \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})\}$$

We have three possible cases:

- If any concrete string in $\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})$ contains the character `c` it means that `c` is propagated in a segmentation $\overline{\mathbf{s}}$ by an abstract domain $\overline{\mathbf{C}}$ and it appears in a segment whose segment bound lower limit is different from zero, by definition of $\mathfrak{B}_{\overline{\mathbf{S}}}$. Then both the concrete and the abstract semantics return `true`.

- If any concrete string in $\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})$ does not contain the character `c` it means that `c` is propagated in a segmentation $\overline{\mathbf{s}}$ by an abstract domain $\overline{\mathbf{C}}$ and it appears in a segment whose segment bound lower limit is equal to zero or it is not propagated at all, by definition of $\mathfrak{B}_{\overline{\mathbf{S}}}$. Then both the concrete and the abstract semantics return `false`.
- Otherwise, if not all the concrete strings in $\alpha_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})$ contain the character `c` it means that `c` is over-approximated in a segmentation $\overline{\mathbf{s}}$ by an abstract domain $\overline{\mathbf{C}}$, by definition of $\mathfrak{B}_{\overline{\mathbf{S}}}$. Then both the concrete and the abstract semantics return $\top_{\mathbf{B}}$.

□

4.6 Refined String Abstract Domains

To obtain our refined string abstract domains we exploit the notion of Granger product [87]. As introduced in Section 2.6.4, the Granger product is a binary operator between abstract domains, based on two refinement operators. Let $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$ be two abstract domains. The Granger operators iteratively lift the information of $\overline{\mathbf{D}}_1$ using the information of $\overline{\mathbf{D}}_2$, and vice-versa, until the smallest reduction is obtained. We will show that the reductions we will provide in the next Sections can be achieved without the need of iterating the refinement operators. Formally, our Granger operators are as follows:

- $\mathsf{refine}_{\overline{\mathbf{D}}_2}\colon \overline{\mathbf{D}}_1 \to \overline{\mathbf{D}}_1$
- $\mathsf{refine}_{\overline{\mathbf{D}}_1}\colon \overline{\mathbf{D}}_2 \to \overline{\mathbf{D}}_2$

Given two abstract elements $\overline{\mathbf{d}}_1$ and $\overline{\mathbf{d}}_2$, which belong to $\overline{\mathbf{D}}_1$ and $\overline{\mathbf{D}}_2$ respectively, a precondition to compute the reduced product between $\overline{\mathbf{d}}_1$ and $\overline{\mathbf{d}}_2$ (i.e., $\overline{\mathbf{d}}_1 \otimes \overline{\mathbf{d}}_2$) is that there must not be inconsistency (**Inc**) between them, i.e., $\gamma_{\overline{\mathbf{D}}_1}(\overline{\mathbf{d}}_1) \cap \gamma_{\overline{\mathbf{D}}_2}(\overline{\mathbf{d}}_2) \neq \emptyset$. In case of inconsistency, the reduced product leads to the pair of bottom elements of their respective abstract domains, formally:

$$\overline{\mathbf{d}}_1 \otimes \overline{\mathbf{d}}_2 = \begin{cases} (\bot_{\overline{\mathbf{D}}_1}, \bot_{\overline{\mathbf{D}}_2}) & \text{if } \mathbf{Inc}(\overline{\mathbf{d}}_1, \overline{\mathbf{d}}_2) \\ (\mathsf{refine}_{\overline{\mathbf{d}}_2}(\overline{\mathbf{d}}_1), \mathsf{refine}_{\overline{\mathbf{d}}_1}(\overline{\mathbf{d}}_2)) & \text{otherwise} \end{cases}$$

4.6.1 Refinement Meaning

Before presenting our refined string abstract domains, we remark what refinement means for any of the domains involved in the reductions. An improvement of the precision of an abstract element in (i) String Length domain means decreasing its range, (ii) Character Inclusion domain means either increasing the cardinality of the set of certainly contained characters, and/or decreasing the cardinality of the set of maybe contained characters, (iii) Prefix domain means increasing the length of its sequence of characters, concatenating one

or more characters to its end, and (iv) Segmentation domain means either increasing the precision of its segment abstract predicates, and/or decreasing the range of the segment bounds.

4.6.2 Combining Segmentation and String Length Domains

The first combination involves the abstract elements $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$, which belong to the Segmentation and the String Length abstract domains respectively, such that $\overline{\mathbf{n}}$ denotes the interval $[m, M]$ introduced in Section 4.4.1 (cf. A.1 for more details). As mentioned above, we define the reduced product between $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ in terms of two refinement operators, improving the approximation reached by one domain through the other and vice-versa. Thus, below, we introduce the notion of *inconsistency* between two abstract elements in the Segmentation and in the String Length domains and the definitions of both refinement operators involved in the reduction of the latter abstract elements. Finally, we prove the soundness of the reduction operators we present. We will follow this line-up also for the reductions we will present later in the chapter.

In the following we denote by $\mathsf{minLen}(\overline{\mathbf{s}})$ the minimal length of a segmentation abstract predicate $\overline{\mathbf{s}}$, i.e., the sum of all its segment lower bounds $\overline{\mathbf{e}}_i$, and by $\mathsf{maxLen}(\overline{\mathbf{s}})$ its maximal length, i.e., the sum of all its segment upper bounds $\overline{\mathbf{e}}_i^{'}$. Formally, let $\overline{\mathbf{s}} = (\overline{\mathbf{c}}_i[\overline{\mathbf{e}}_i, \overline{\mathbf{e}}_i^{'}])_{i=1}^{n}$ with $n \geq 1$ (shortcut for $\overline{\mathbf{c}}_1[\overline{\mathbf{e}}_1, \overline{\mathbf{e}}_1^{'}]...\overline{\mathbf{c}}_n[\overline{\mathbf{e}}_n, \overline{\mathbf{e}}_n^{'}]$) then,

- $\mathsf{minLen}(\overline{\mathbf{s}}) = \begin{cases} \sum_{i=1}^{n}{}_{\overline{\mathbf{E}}}\ \overline{\mathbf{e}}_i & \text{if } \forall i \in \overline{\mathbf{s}} : \overline{\mathbf{e}}_i \text{ does not contain variables with constant value} \\ 0 & \text{otherwise} \end{cases}$

- $\mathsf{maxLen}(\overline{\mathbf{s}}) = \begin{cases} \sum_{i=1}^{n}{}_{\overline{\mathbf{E}}}\ \overline{\mathbf{e}}_i^{'} & \text{if } \forall i \in \overline{\mathbf{s}} : \overline{\mathbf{e}}_i^{'} \text{ does not contain variables with constant value} \\ \top_{\overline{\mathbf{E}}} & \text{otherwise} \end{cases}$

Definition 4.1 (Inconsistency between Segmentation and String Length)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ be two abstract elements in the Segmentation and in the String Length domains respectively. Moreover, let $\gamma_{\overline{\mathbf{S}}}$ and $\gamma_{\overline{\mathbf{SL}}}$ be the concretization functions of the domains just mentioned. There is inconsistency between $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ when:

$$\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}) \cap \gamma_{\overline{\mathbf{SL}}}(\overline{\mathbf{n}}) = \emptyset.$$

△

Lemma 4.1 (Inconsistency between Segmentation and String Length)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ be two abstract elements in the Segmentation and in the String Length domains respectively. If at least one of the following conditions holds then $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ are inconsistent.

1. If the intersection between the interval from the minimal to the maximal length of $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ is empty, then $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ are inconsistent. Formally:

$$[\mathsf{minLen}(\overline{\mathbf{s}}), \mathsf{maxLen}(\overline{\mathbf{s}})] \cap [\mathrm{m}, \mathrm{M}] = \emptyset \Rightarrow \mathbf{Inc}(\overline{\mathbf{s}}, \overline{\mathbf{n}})$$

2. If at least one of the two abstract elements corresponds to the bottom element of their lattice, then $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ are inconsistent. Formally:

$$\overline{\mathbf{s}} = \bot_{\overline{\mathbf{S}}} \text{ or } \overline{\mathbf{n}} = \bot_{\overline{\mathbf{SL}}} \Rightarrow \mathbf{Inc}(\overline{\mathbf{s}}, \overline{\mathbf{n}})$$

Proof.

Case 1 The segmentation $\overline{\mathbf{s}}$ approximates strings having length that might varies from $\mathsf{minLen}(\overline{\mathbf{s}})$ to $\mathsf{maxLen}(\overline{\mathbf{s}})$. The interval $\overline{\mathbf{n}}$ approximates all the possible strings having length that ranges from m to M. If the intersection of the two length intervals is empty, this means that $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ are abstracting strings of different length. Thus, the intersection of the concrete string sets represented by $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ will be empty.

Case 2 It is trivial, as the concretization of bottom elements is the empty set.

□

Definition 4.2 (Segmentation refinement through String Length)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ be two abstract elements in the Segmentation and in the String Length domains respectively. We define $\mathsf{refine}_{\overline{\mathbf{n}}}(\overline{\mathbf{s}})$ as the refinement operator describing how the information tracked by $\overline{\mathbf{n}}$ improves the one from $\overline{\mathbf{s}}$. Formally, $\mathsf{refine}_{\overline{\mathbf{n}}}(\overline{\mathbf{s}}) = \overline{\mathbf{s}}^*$ where:

a. if $\overline{\mathbf{s}}$ contains only one segment and its segment bound lower limit might be smaller than or equal to the one of $\overline{\mathbf{n}}$, then the refinement means substituting $\overline{\mathbf{e}}_{\mathrm{i}}$ with m. Formally:

$$\overline{\mathbf{s}} = \overline{\mathbf{c}}_1[\overline{\mathbf{e}}_1, \overline{\mathbf{e}}_1^{'}] \wedge \overline{\mathbf{e}}_1 \leq_{\overline{\mathbf{E}}} \mathrm{m} \Rightarrow \overline{\mathbf{s}}^* = \overline{\mathbf{s}}[\overline{\mathbf{e}}_1/\overline{\mathbf{e}}_1^*]$$

where $\overline{\mathbf{e}}_1^* = \mathrm{m}$.

b. if $\overline{\mathbf{s}}$ contains only one segment and its segment bound upper limit might be greater than or equal to the one of $\overline{\mathbf{n}}$, then the refinement means substituting $\overline{\mathbf{e}}_{\mathrm{i}}^{'}$ with M. Formally:

$$\overline{\mathbf{s}} = \overline{\mathbf{c}}_1[\overline{\mathbf{e}}_1, \overline{\mathbf{e}}_1^{'}] \wedge \mathrm{M} \leq_{\overline{\mathbf{E}}} \overline{\mathbf{e}}_1^{'} \Rightarrow \overline{\mathbf{s}}^* = \overline{\mathbf{s}}[\overline{\mathbf{e}}_1^{'}/\overline{\mathbf{e}}_1^*]$$

where $\overline{\mathbf{e}}_1^* = \mathrm{M}$.

Conditions a. and b. can occur at the same time. In this case both refinements apply. Observe that the order of application is not relevant.

If neither a. nor b. can occur, then $\overline{\mathbf{s}} = \overline{\mathbf{s}}^*$

△

Example 4.8. Consider $\overline{\mathbf{s}}$ = 'a' $[3, 5 + \mathtt{x}]$ where $\overline{\mathbf{C}}$ is the constant propagation domain for characters and $\overline{\mathbf{n}} = [4, 7]$. Moreover, assume that the value of $\mathtt{x}$ is greater than 4. The refinement of $\overline{\mathbf{s}}$ through $\overline{\mathbf{n}}$ is as follows:

$$\begin{aligned} \mathsf{refine}_{\overline{\mathbf{n}}}(\overline{\mathbf{s}}) &= \overline{\mathbf{s}}[\overline{\mathbf{e}}_1^{'}/\overline{\mathbf{e}}_1^{*}] \\ \overline{\mathbf{b}}_1^{*} &= 7 \\ &= \text{'a'}\ [3, 7] \end{aligned}$$

Indeed the segment bound of $\overline{\mathbf{s}}$ has the upper limit equal to $5+\mathtt{x}$ where $\mathtt{x}$ is assumed to be an integer variable whose value is greater than 4, and the maximal length of the strings approximated by $\overline{\mathbf{n}}$ is 7. So we could substitute the segment bound upper limit of $\overline{\mathbf{s}}$ with the upper limit of $\overline{\mathbf{n}}$, as by case b. of Definition 4.2.

⬡

Definition 4.3 (String Length refinement through Segmentation)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ be two abstract elements in the Segmentation and in the String Length domains respectively. We define $\mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{n}})$ as the refinement operator describing how the information tracked by $\overline{\mathbf{s}}$ improves the one from $\overline{\mathbf{n}}$. Formally, $\mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{n}}) = \overline{\mathbf{n}}^*$ where:

a. if the lower limit of $\overline{\mathbf{n}}$ is smaller than the the minimal length of $\overline{\mathbf{s}}$, then the refinement means substituting m with $\mathsf{minLen}(\overline{\mathbf{s}})$. Formally:

$$\mathrm{m} <_{\overline{\mathbf{E}}} \mathsf{minLen}(\overline{\mathbf{s}}) \Rightarrow \overline{\mathbf{n}}^* = \overline{\mathbf{n}}[\mathrm{m}/\mathsf{minLen}(\overline{\mathbf{s}})]$$

b. if the upper limit of $\overline{\mathbf{n}}$ is greater than the maximal length of $\overline{\mathbf{s}}$, then the refinement means substituting M with $\mathsf{maxLen}(\overline{\mathbf{s}})$. Formally:

$$\mathsf{maxLen}(\overline{\mathbf{s}}) <_{\overline{\mathbf{E}}} \mathrm{M} \Rightarrow \overline{\mathbf{n}}^* = \overline{\mathbf{n}}[\mathrm{M}/\mathsf{maxLen}(\overline{\mathbf{s}})]$$

Both the conditions above can occur at the same time. In this case both refinements apply. Observe that the order of application is not relevant.

If both conditions cannot occur, then $\overline{\mathbf{n}} = \overline{\mathbf{n}}^*$.

△

Example 4.9. Consider $\overline{\mathbf{s}}$ = 'a' $[3, 7]$ 'b' $[1, 5]$ 'c' $[1, 1 + \mathtt{x}]$ (with the assumptions of the example above) and $\overline{\mathbf{n}} = [4, 7]$. The refinement of $\overline{\mathbf{n}}$ through $\overline{\mathbf{s}}$ is as follows:

$$\begin{aligned} \mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{n}}) &= \overline{\mathbf{n}}[\mathrm{m}/\mathsf{minLen}(\overline{\mathbf{s}})] \\ \mathsf{minLen}(\overline{\mathbf{s}}) &= 5 \\ &= [5, 7] \end{aligned}$$

Indeed, the lower limit of $\overline{\mathbf{n}}$ is strictly smaller than the minimum length of $\overline{\mathbf{s}}$. So we could substitute the lower limit of $\overline{\mathbf{n}}$ with $\mathsf{minLen}(\overline{\mathbf{s}})$, as by case a. of Definition 4.3.

⬡

Lemma 4.2 (Reduced product between Segmentation and String Length)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ be two abstract elements in the Segmentation and in the String Length abstract domains respectively and, consider the refinement operators presented in the Definitions 4.2 and 4.3. The reduced product of $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ is obtained by applying $\mathsf{refine}_{\overline{\mathbf{n}}}$ and $\mathsf{refine}_{\overline{\mathbf{s}}}$ as follows:

$$\overline{\mathbf{s}} \otimes \overline{\mathbf{n}} = (\mathsf{refine}_{\overline{\mathbf{n}}}(\overline{\mathbf{s}}), \mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{n}}))$$

Lemma 4.3 (Soundness of Segmentation refinement)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ be two abstract elements in the Segmentation and in the String Length abstract domains respectively, and $\mathsf{refine}_{\overline{\mathbf{n}}}(\overline{\mathbf{s}}) = \overline{\mathbf{s}}^*$. The soundness of the Segmentation refinement can be expressed by:

$$\sigma \in [\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}) \backslash \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}^*)] \Rightarrow \sigma \notin \gamma_{\overline{\mathbf{SL}}}(\overline{\mathbf{n}})$$

Proof.
The set $[\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}) \backslash \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}^*)]$ contains representations of strings whose length does not belong to the interval $\overline{\mathbf{n}}$, either in case a. and in case b. of Definition 4.2. Thus, those strings do not belong to $\gamma_{\overline{\mathbf{SL}}}(\overline{\mathbf{n}})$, by definition of $\gamma_{\overline{\mathbf{SL}}}$ itself.

□

Lemma 4.4 (Soundness of String Length refinement)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{n}}$ be two abstract elements in the Segmentation and in the String Length abstract domains respectively, and $\mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{n}}) = \overline{\mathbf{n}}^*$. The soundness of the String Length refinement can be expressed by:

$$\sigma \in [\gamma_{\overline{\mathbf{SL}}}(\overline{\mathbf{n}}) \backslash \gamma_{\overline{\mathbf{SL}}}(\overline{\mathbf{n}}^*)] \Rightarrow \sigma \notin \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})$$

Proof.
The set $[\gamma_{\overline{\mathbf{SL}}}(\overline{\mathbf{n}}) \backslash \gamma_{\overline{\mathbf{SL}}}(\overline{\mathbf{n}}^*)]$ contains representations of strings whose length does not belong to the interval that goes from the minimal to the maximal length of $\overline{\mathbf{s}}$, either in case a. and in case b. of Definition 4.3. Thus, those strings do not belong to $\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})$, by definition of $\gamma_{\overline{\mathbf{S}}}$ itself.

□

4.6.3 Combining Segmentation and Character Inclusion Domains

The second combination involves the abstract elements $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$, which belong to the Segmentation and the Character Inclusion abstract domains respectively, such that $\overline{\mathbf{r}}$ denotes the pair of sets $(\mathrm{C}, \mathrm{MC})$ introduced in Section 4.4.2 (cf. Appendix A.2 for more details).

Definition 4.4 (Inconsistency between Segmentation and Character Inclusion)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ be two abstract elements in the Segmentation and in the Character Inclusion domains respectively. Moreover, let $\gamma_{\overline{\mathbf{S}}}$ and $\gamma_{\overline{\mathbf{CI}}}$ be the concretization functions of the domains just mentioned. There is inconsistency between $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ when:

$$\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}) \cap \gamma_{\overline{\mathbf{CI}}}(\overline{\mathbf{r}}) = \emptyset.$$

△

Lemma 4.5 (Inconsistency between Segmentation and Character Inclusion)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ be two abstract elements in the Segmentation and in the Character Inclusion domains respectively. If at least one of the following conditions holds, then $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ are inconsistent.

1. If for each character in the set of certainly contained characters of $\overline{\mathbf{r}}$ does not exist at least one segment abstract predicate $\overline{\mathbf{c}}$ in $\overline{\mathbf{s}}$ that approximates it, then $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ are inconsistent. Formally:

$$|\mathrm{C}| = \mathrm{v} \wedge \forall \mathrm{c} \in \mathrm{C} : \nexists^{\mathrm{v}} \mathrm{i} \in \overline{\mathbf{s}} : \mathrm{c} \in \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_{\mathrm{i}}) \Rightarrow \mathbf{Inc}(\overline{\mathbf{s}}, \overline{\mathbf{r}})$$

   where $\nexists^{\mathrm{v}}$ is the existential quantifier with cardinality v introduced in Section 2.1.

2. If there exists at least one segment abstract predicate $\overline{\mathbf{c}}$ in $\overline{\mathbf{s}}$ which abstracts only characters that do not occur in the set of maybe contained characters of $\overline{\mathbf{r}}$ and its segment bound lower limit is strictly greater than 0, then $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ are inconsistent. Formally:

$$\exists \mathrm{i} \in \overline{\mathbf{s}} : \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_{\mathrm{i}}) \cap \mathrm{MC} = \emptyset \wedge \overline{\mathbf{e}}_{\mathrm{i}} >_{\overline{\mathbf{E}}} 0 \Rightarrow \mathbf{Inc}(\overline{\mathbf{s}}, \overline{\mathbf{r}})$$

3. If the set of certainly contained and maybe contained characters of $\overline{\mathbf{r}}$ are both equal to the empty-set (i.e., $\overline{\mathbf{r}}$ abstracts an empty string) and the minimum length of $\overline{\mathbf{s}}$ is strictly greater than 0, then $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ are inconsistent. Formally:

$$\mathrm{C} = \mathrm{MC} = \emptyset \wedge \mathsf{minLen}(\overline{\mathbf{s}}) >_{\overline{\mathbf{E}}} 0 \Rightarrow \mathbf{Inc}(\overline{\mathbf{s}}, \overline{\mathbf{r}})$$

4. If at least one of the two abstract elements corresponds to the bottom element of their lattice, then $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ are inconsistent. Formally:

$$\overline{\mathbf{s}} = \bot_{\overline{\mathbf{S}}} \text{ or } \overline{\mathbf{r}} = \bot_{\overline{\mathbf{CI}}} \Rightarrow \mathbf{Inc}(\overline{\mathbf{s}}, \overline{\mathbf{r}})$$

Proof.

Case 1 The pair $(\mathrm{C}, \mathrm{MC})$ approximates strings certainly containing the characters in C and possibly containing the characters in $[\mathrm{MC} \setminus \mathrm{C}]$. If for each character in C does not exist at least one segment abstract predicate $\overline{\mathbf{c}}$ in $\overline{\mathbf{s}}$ approximating it, this means that $\overline{\mathbf{s}}$ is abstracting strings that do not simultaneously contain all the characters belonging to the set of certainly contained characters of $\overline{\mathbf{r}}$. Thus, the intersection of the concrete string sets represented by $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ will be empty.

Case 2 If the segmentation $\overline{\mathbf{s}}$ contains at least one segment abstract predicate $\overline{\mathbf{c}}$ approximating characters that do not belong to the set of maybe contained characters of $\overline{\mathbf{r}}$ and its segment bound lower limit is strictly greater than 0, this means that $\overline{\mathbf{s}}$ is abstracting strings in which a character not present in $\overline{\mathbf{r}}$ surely occurs. Thus, the intersection of the concrete string sets represented by $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ will be empty.

Case 3 If $\overline{\mathbf{r}}$ approximates exactly the empty string and the minimum length of $\overline{\mathbf{s}}$ is strictly greater than 0 means that the concretization of $\overline{\mathbf{s}}$ does not contain the empty string. Thus, the intersection of the concrete string sets represented by $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ will be empty.

Case 4 is trivial, as the concretization of bottom elements is an empty set.

□

Definition 4.5 (Segmentation refinement through Character Inclusion)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ be two abstract elements in the Segmentation and in the Character Inclusion domains respectively. We define $\mathsf{refine}_{\overline{\mathbf{r}}}(\overline{\mathbf{s}})$ as the refinement operator describing how the information tracked by $\overline{\mathbf{r}}$ improves the one from $\overline{\mathbf{s}}$. Formally, $\mathsf{refine}_{\overline{\mathbf{r}}}(\overline{\mathbf{s}}) = \overline{\mathbf{s}}^*$ where:

a. if the number of segment abstract predicates $\overline{\mathbf{c}}$ in $\overline{\mathbf{s}}$ which (also) approximate the characters belonging to the set of certainly contained characters of $\overline{\mathbf{r}}$ is equal to the cardinality of the latter set and, their segment bound lower limit is equal to 0, then the refinement means substituting those lower limits with 1. Formally:

$$|\mathrm{C}| = \mathrm{v} \wedge \exists^{\mathrm{v}} \mathrm{i} \in \overline{\mathbf{s}} : \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_{\mathrm{i}}) \cap \mathrm{C} \neq \emptyset \wedge \overline{\mathbf{e}}_{\mathrm{i}} =_{\overline{\mathbf{E}}} 0 \Rightarrow \overline{\mathbf{s}}^* = \overline{\mathbf{s}}[\overline{\mathbf{e}}_{\mathrm{i}}/1]$$

b. if the number of segment abstract predicates $\overline{\mathbf{c}}$ in $\overline{\mathbf{s}}$ which (also) approximate the characters belonging to the set of certainly contained characters of $\overline{\mathbf{r}}$ is equal to the cardinality of the latter set, then the refinement means substituting those predicates, followed by a segment upper bound equal to 1, with the abstract value, in an abstract domain $\overline{\mathbf{C}}$, of the set resulting from the intersection between the set of characters represented by $\overline{\mathbf{c}}$ and the set of certainly contained characters of $\overline{\mathbf{r}}$ (if it is preciser than $\overline{\mathbf{c}}$). Formally:

$$|\mathrm{C}| = \mathrm{v} \wedge \exists^{\mathrm{v}} \mathrm{i} \in \overline{\mathbf{s}} : \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_\mathrm{i}) \cap \mathrm{C} \neq \emptyset \wedge \overline{\mathbf{e}}_\mathrm{i}^{'} =_{\overline{\mathbf{E}}} 1$$
$$\alpha_{\overline{\mathbf{C}}}(\mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_\mathrm{i}) \cap \mathrm{C}) \sqsubseteq_{\overline{\mathbf{C}}} \overline{\mathbf{c}}_\mathrm{i} \Rightarrow \overline{\mathbf{s}}^* = \overline{\mathbf{s}}[\overline{\mathbf{c}}_\mathrm{i}/\alpha_{\overline{\mathbf{C}}}(\mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_\mathrm{i}) \cap \mathrm{C})]$$

c. if there are segment abstract predicates $\overline{\mathbf{c}}$ in $\overline{\mathbf{s}}$ which (also) approximate characters that do not belong to the set of maybe contained characters of $\overline{\mathbf{r}}$, then the refinement means substituting those predicates, followed by a segment upper bound strictly greater than 1, with the abstract value, in an abstract domain $\overline{\mathbf{C}}$, of the set resulting from the intersection between the set of characters represented by $\overline{\mathbf{c}}$ and the set of maybe contained characters of $\overline{\mathbf{r}}$ (if it is preciser than $\overline{\mathbf{c}}$). Formally:

$$\exists \mathrm{i} \in \overline{\mathbf{s}} : \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_\mathrm{i}) \backslash \mathrm{MC} \neq \emptyset \wedge \overline{\mathbf{e}}_\mathrm{i}^{'} >_{\overline{\mathbf{E}}} 1$$
$$\alpha_{\overline{\mathbf{C}}}(\mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_\mathrm{i}) \cap \mathrm{MC}) \sqsubseteq_{\overline{\mathbf{C}}} \overline{\mathbf{c}}_\mathrm{i} \Rightarrow \overline{\mathbf{s}}^* = \overline{\mathbf{s}}[\overline{\mathbf{c}}_\mathrm{i}/\alpha_{\overline{\mathbf{C}}}(\mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_\mathrm{i}) \cap \mathrm{MC})]$$

The conditions above can occur at the same time. In this case all refinements apply. Observe that the order of application is not relevant.

If none of the conditions above apply, then $\overline{\mathbf{s}} = \overline{\mathbf{s}}^*$.

△

Example 4.10. Consider $\overline{\mathbf{s}}$ = 'a' $[0,2]$ $\top_{\overline{\mathsf{CP}}}[0,1]$ 'b' $[4,8]$, where $\top_{\overline{\mathsf{CP}}}$ is the top element in the constant propagation domains for characters, and $\overline{\mathbf{r}} = (\{\mathrm{a,b,c}\}, \{\mathrm{a,b,c,d}\})$. The refinement of $\overline{\mathbf{s}}$ through $\overline{\mathbf{r}}$ is as follows:

$$\begin{aligned}\mathsf{refine}_{\overline{\mathbf{r}}}(\overline{\mathbf{s}}) &= \overline{\mathbf{s}}[\overline{\mathbf{e}}_1/1, \overline{\mathbf{e}}_2/1, \overline{\mathbf{c}}_2/\alpha_{\overline{\mathsf{CP}}}(\{\mathrm{c}\})] \\ &= \text{'a'}\ [1,2]\ \text{'c'}\ [1,1]\ \text{'b'}\ [4,8]\end{aligned}$$

Indeed, the number of segment abstract predicates in $\overline{\mathbf{s}}$ approximating characters contained in the set of certainly contained characters of $\overline{\mathbf{r}}$ is equal to the cardinality of the latter set. So we substitute their segment bound lower limit with 1 when it was originally equal to zero, as by case a of Definition 4.5. Moreover, we substitute the second abstract predicate of $\overline{\mathbf{s}}$ with the value (in the constant propagation abstract domain) of the only certainly contained character of $\overline{\mathbf{r}}$ that is not directly present in $\overline{\mathbf{s}}$, as by case b. of Definition 4.5, thus refining it.

⬡

Definition 4.6 (Character Inclusion refinement through Segmentation)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ be two abstract elements in the Segmentation and in the Character Inclusion domains respectively. We define $\mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{r}})$ as the refinement operator describing how the information tracked by $\overline{\mathbf{s}}$ improves the one from $\overline{\mathbf{r}}$. Formally, $\mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{r}}) = \overline{\mathbf{r}}^*$ where:

a. if there is a segment abstract predicate $\overline{\mathbf{c}}$ in $\overline{\mathbf{s}}$ which (also) approximates exactly one character belonging to the set of maybe contained characters of $\overline{\mathbf{r}}$, but that does not

belong to the set of certainly contained characters of $\overline{\mathbf{r}}$ and, its segment bound lower limit is strictly greater than 0, then the refinement means adding that character to the set of certainly contained characters of $\overline{\mathbf{r}}$. Formally:

$$\exists \mathrm{i} \in \overline{\mathbf{s}} : \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_\mathrm{i}) \cap (\mathrm{MC} \backslash \mathrm{C}) = \{\mathrm{c}\} \wedge \overline{\mathbf{e}}_\mathrm{i} >_{\overline{\mathbf{E}}} 0 \Rightarrow \overline{\mathbf{r}}^* = \overline{\mathbf{r}}[\mathrm{C}/\mathrm{C} \cup \{\mathrm{c}\}]$$

b. if the the set of characters represented by $\overline{\mathbf{s}}$ is a strict subset of the maybe contained characters of $\overline{\mathbf{r}}$, then the set of maybe contained characters of $\overline{\mathbf{r}}$ can be refined considering just the characters occurring in that intersection. Formally:

$$\mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{s}}) \subset \mathrm{MC} \neq \mathrm{MC} \Rightarrow \overline{\mathbf{r}}^* = \overline{\mathbf{r}}[\mathrm{MC}/(\mathsf{char}_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}) \cap \mathrm{MC})]$$

where $\mathsf{char}_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}) = \bigcup_{\overline{\mathbf{c}} \in \overline{\mathbf{s}}} \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}})$

△

Example 4.11. Consider $\overline{\mathbf{s}}$ = 'a' $[2,2]$ 'b' $[3,5]$ 'c' $[4,4]$ 'd' $[0,2]$ and $\overline{\mathbf{r}} = (\{\mathrm{a},\mathrm{b}\}, \{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d},\mathrm{e},\mathrm{f}\})$. The refinement of $\overline{\mathbf{r}}$ through $\overline{\mathbf{s}}$ is as follows:

$$\begin{aligned}
\mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{r}}) &= \overline{\mathbf{r}}[\mathrm{C}/(\mathrm{C} \cup (\mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_3) \cap (\mathrm{MC} \backslash \mathrm{C})), \mathrm{MC}/\mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{s}})] \\
&= (\{\mathrm{a},\mathrm{b}\} \cup (\{\mathrm{c}\} \cap \{\mathrm{c},\mathrm{d},\mathrm{e},\mathrm{f}\}), \{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d}\}) \\
&= (\{\mathrm{a},\mathrm{b}\} \cup \{\mathrm{c}\}, \{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d}\}) \\
&= (\{\mathrm{a},\mathrm{b},\mathrm{c}\}, \{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d}\})
\end{aligned}$$

Indeed, the third segment abstract predicate of $\overline{\mathbf{s}}$ has a segment bound lower limit strictly greater than 0 and it approximates exactly the character 'c' which belongs to the set of maybe contained character of $\overline{\mathbf{r}}$, but that does not belong to the set of certainly contained characters of $\overline{\mathbf{r}}$. So we add the character 'c' to the set of certainly contained characters of $\overline{\mathbf{r}}$, as by case a. of Definition 4.6. Moreover, we modify the set of maybe contained character of $\overline{\mathbf{r}}$ considering just the characters occurring in the set of characters approximated by $\overline{\mathbf{s}}$ ($\mathsf{char}_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}) \cap \mathrm{MC} = \{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d}\} \cap \{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d},\mathrm{e},\mathrm{f}\} = \{\mathrm{a},\mathrm{b},\mathrm{c},\mathrm{d}\}$), as by case b. of Definition 4.6, thus refining it.

⬡

Lemma 4.6 (Reduced product between Segmentation and Character Inclusion)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ be two abstract elements in the Segmentation and in the Character Inclusion abstract domains respectively and, consider the refinement operators presented in the Definitions 4.5 and 4.6. The reduced product of $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ is obtained by applying $\mathsf{refine}_{\overline{\mathbf{r}}}$ and $\mathsf{refine}_{\overline{\mathbf{s}}}$ as follows:

$$\overline{\mathbf{s}} \otimes \overline{\mathbf{r}} = (\mathsf{refine}_{\overline{\mathbf{r}}}(\overline{\mathbf{s}}), \mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{r}}))$$

Lemma 4.7 (Soundness of Segmentation refinement)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ be two abstract elements in the Segmentation and in the Character Inclusion abstract domains respectively, and let $\gamma_{\overline{\mathbf{S}}}$ and $\gamma_{\overline{\mathbf{CI}}}$ be the concretization functions of the domains just mentioned. Moreover, $\mathsf{refine}_{\overline{\mathbf{r}}}(\overline{\mathbf{s}}) = \overline{\mathbf{s}}^*$. The soundness of the Segmentation refinement can be expressed by:

$$\sigma \in [\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}) \setminus \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}^*)] \Rightarrow \sigma \notin \gamma_{\overline{\mathbf{CI}}}(\overline{\mathbf{r}})$$

Proof.
The set $[\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}) \setminus \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}}^*)]$ contains representations of strings in which do not occur at least one character that appears in the set of certainly contained characters of $\overline{\mathbf{r}}$, either in case a. and in case b. of Definition 4.5, or at least one character that does not appear in the set of maybe contained characters of $\overline{\mathbf{r}}$, in case c of Definition 4.5. Thus, those strings do not belong to $\gamma_{\overline{\mathbf{CI}}}(\overline{\mathbf{r}})$, by definition of $\gamma_{\overline{\mathbf{CI}}}$ itself.

□

Lemma 4.8 (Soundness of Character Inclusion refinement)**.** Let $\overline{\mathbf{s}}$ and $\overline{\mathbf{r}}$ be two abstract elements in the Segmentation and in the Character Inclusion abstract domains respectively, and let $\gamma_{\overline{\mathbf{S}}}$ and $\gamma_{\overline{\mathbf{CI}}}$ be the concretization functions of the domains just mentioned. Moreover, $\mathsf{refine}_{\overline{\mathbf{s}}}(\overline{\mathbf{r}}) = \overline{\mathbf{r}}^*$. The soundness of the Character Inclusion refinement can be expressed by:

$$\sigma \in [\gamma_{\overline{\mathbf{CI}}}(\overline{\mathbf{r}}) \setminus \gamma_{\overline{\mathbf{CI}}}(\overline{\mathbf{r}}^*)] \Rightarrow \sigma \notin \gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})$$

Proof.
The set $[\gamma_{\overline{\mathbf{CI}}}(\overline{\mathbf{r}}) \setminus \gamma_{\overline{\mathbf{CI}}}(\overline{\mathbf{r}}^*)]$ contains representations of strings in which occurs at least one character that does not appear in the set of characters represented by $\overline{\mathbf{s}}$, either in case a. and in case b. of Definition 4.6. Thus, those strings do not belong to $\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})$, by definition of $\gamma_{\overline{\mathbf{S}}}$ itself.

□

4.6.4 Combining Segmentation and Prefix Domains

The last combination we present is the one involving the abstract elements $\overline{\mathbf{s}}$ and $\overline{\mathbf{p}}$, which belong to the Segmentation and the Prefix abstract domains respectively. The reductions between the Segmentation and both the Suffix and the Prefix-Suffix abstract domains can be naturally induced by the Granger product among $\overline{\mathbf{s}}$ and $\overline{\mathbf{p}}$. Notice that the refinement of $\overline{\mathbf{s}}$ is defined according to the length of the sequence of characters of $\overline{\mathbf{p}}$.

Algorithm 5 isPrefix procedure

Input: $\overline{\mathsf{p}}$ and $\overline{\mathsf{s}}$

Output: res $\in \{\texttt{true}, \texttt{false}, \top_{\mathbf{B}}\}$

1: $\mathrm{k} \leftarrow 0$, $\mathrm{i} \leftarrow 1$, res $\leftarrow$ `true`
2: $\overline{\mathsf{s}} \leftarrow \mathsf{unFold}(\overline{\mathsf{s}})$
3: **while** $\mathrm{k} \in [0, \mathsf{len}_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}) - 1]$ **do**
4: **while** $\mathrm{i} \in [1, \mathsf{numSeg}((\overline{\mathsf{s}}))]$ **do**
5: **if** $\overline{\mathsf{p}}[\mathrm{k}] \in \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}.\overline{\mathsf{s}}[\mathrm{i}]) \wedge \overline{\mathbf{b}}.\overline{\mathsf{s}}[\mathrm{i}] \in \{[0,1],[1,1]\}$ **then**
6: $\mathrm{k} \leftarrow \mathrm{k}+1$, $\mathrm{i} \leftarrow \mathrm{i}+1$
7: **else if** $(\overline{\mathsf{p}}[\mathrm{k}] \in \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}.\overline{\mathsf{s}}[\mathrm{i}]) \wedge \overline{\mathbf{b}}.\overline{\mathsf{s}}[\mathrm{i}] \in \{[0,\mathtt{x}],[\mathtt{x},\mathtt{x}]\}) \vee (\overline{\mathsf{p}}[\mathrm{k}] \notin \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}.\overline{\mathsf{s}}[\mathrm{i}]) \wedge \overline{\mathbf{b}}.\overline{\mathsf{s}}[\mathrm{i}] \in \{[\mathtt{x},\mathtt{x}]\})$ **then**
8: **return** res $\leftarrow \top_{\mathbf{B}}$
9: **else if** $(\overline{\mathsf{p}}[\mathrm{k}] \notin \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}.\overline{\mathsf{s}}[\mathrm{i}]) \wedge \overline{\mathbf{b}}.\overline{\mathsf{s}}[\mathrm{i}] \in \{[0,0],[0,\mathtt{x}],[0,1]\}) \vee (\overline{\mathsf{p}}[\mathrm{k}] \in \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}.\overline{\mathsf{s}}[\mathrm{i}]) \wedge \overline{\mathbf{b}}.\overline{\mathsf{s}}[\mathrm{i}] \in \{[0,0]\})$ **then**
10: $\mathrm{i} \leftarrow \mathrm{i}+1$
11: **else**
12: **return** res $\leftarrow$ `false`
13: **return** res

Algorithm 5 determines if an abstract element in $\overline{\mathbf{PR}}$ might be a prefix of (some of) the strings approximated by a segmentation in $\overline{\mathbf{S}}$. Note that $\mathsf{len}_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}})$ denotes the length of the prefix $\overline{\mathsf{p}}$ and it is equal to the length of its sequence of characters, asterisk $*$ included. Once the segmentation has been unfolded[1] (line 2), we compare the characters in the prefix element $\overline{\mathsf{p}}$ and the segments in the unfolded segmentation $\overline{\mathsf{s}}$. An element $\overline{\mathsf{p}}$ is a prefix of (some of) the strings approximated by $\overline{\mathsf{s}}$ if for each character of $\overline{\mathsf{p}}$ there exists a *corresponding* segment whose segment abstract predicate approximates (also) the character of $\overline{\mathsf{p}}$ under consideration, and its segment bound is of the type [0,1] or [1,1] (lines 5-6). The algorithm return $\top_{\mathbf{B}}$ if the character of $\overline{\mathsf{p}}$ under consideration is approximated by a segment abstract predicate whose bound is of the type [0,x] or [x,x] (lines 7-8). It returns $\top_{\mathbf{B}}$ also in the case in which a character of $\overline{\mathsf{p}}$ is not approximated by the segment abstract predicate under consideration if its segment bound is of the type [x,x]. The algorithm returns `false` if the corresponding segment of the character of $\overline{\mathsf{p}}$ under consideration is not empty and does not approximate it. In all the other cases, i.e., where possibly empty segments that do not approximate the character of $\overline{\mathsf{p}}$ under consideration or where possibly empty segments approximate the character of $\overline{\mathsf{p}}$ under consideration, the procedure moves the comparison to the next segment (lines 9-10).

Definition 4.7 (Inconsistency between Segmentation and Prefix)**.** Let $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ be two abstract elements in the Segmentation and in the Prefix domains respectively. Moreover,

[1] The unFold procedure can not be defined in general as it depends on the normal form of the expressions in $\overline{\mathbf{E}}$, but it is reasonable to assume that an unfolded segmentation can have a limited set of segment bounds types, that are: $[0,0],[0,1],[1,1],[0,\mathtt{x}],[\mathtt{x},\mathtt{x}]$, with $\mathtt{x} \in \overline{\mathbf{X}}$ (cf. Section 4.5) if expressions contain variables.

let $\gamma_{\overline{\mathbf{S}}}$ and $\gamma_{\overline{\mathbf{PR}}}$ be the concretization functions of the domains just mentioned. There is inconsistency between $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ when:

$$\gamma_{\overline{\mathbf{S}}}(\overline{\mathsf{s}}) \cap \gamma_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}) = \emptyset$$

△

Lemma 4.9 (Inconsistency between Segmentation and Prefix)**.** Let $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ be two abstract elements in the Segmentation and in the Prefix domains respectively. If at least one of the following condition holds, then $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ are inconsistent.

1. If isPrefix (cf. Algorithm 5) of $\overline{\mathsf{p}}$ and $\overline{\mathsf{s}}$ returns `false`, then $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ are inconsistent. Formally:

$$\mathsf{isPrefix}(\overline{\mathsf{p}}, \overline{\mathsf{s}}) = \texttt{false} \Rightarrow \mathbf{Inc}(\overline{\mathsf{s}}, \overline{\mathsf{p}})$$

2. If at least one of the two abstract elements corresponds to the bottom element of their lattice, then $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ are inconsistent. Formally:

$$\overline{\mathsf{s}} = \bot_{\overline{\mathbf{S}}} \text{ or } \overline{\mathsf{p}} = \bot_{\overline{\mathbf{PR}}} \Rightarrow \mathbf{Inc}(\overline{\mathsf{s}}, \overline{\mathsf{p}})$$

Proof.

Case 1 The prefix $\overline{\mathsf{p}}$ approximates all the possible strings sharing the same starting sequence of characters. The segmentation $\overline{\mathsf{s}}$ approximates strings highlighting sequences of equal characters. If the isPrefix procedure on $\overline{\mathsf{p}}$ and $\overline{\mathsf{s}}$ returns `false` means that $\overline{\mathsf{p}}$ and $\overline{\mathsf{s}}$ are abstracting strings with different prefixes. Thus, the intersection of the concrete string sets represented by $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ will be empty.

Case 2 is trivial, as the concretization of bottom elements is the empty set.

□

Definition 4.8 (Segmentation refinement through Prefix)**.** Let $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ be two abstract elements in the Segmentation and in the Prefix domains respectively. We define $\mathsf{refine}_{\overline{\mathsf{p}}}(\overline{\mathsf{s}})$ as the refinement operator describing how the information tracked by $\overline{\mathsf{p}}$ improves the one from $\overline{\mathsf{s}}$. Let $\mathrm{unfold}(\overline{\mathsf{s}}) = \overline{\mathsf{s}}'$. Formally, $\mathsf{refine}_{\overline{\mathsf{p}}}(\overline{\mathsf{s}}) = \overline{\mathsf{s}}^*$ where:

a. if the abstraction of a character in $\overline{\mathsf{p}}$, in an abstract domain $\overline{\mathbf{C}}$, precedes the segment abstract predicate of its matching segment in $\overline{\mathsf{s}}'$, then the refinement means substituting that segment abstract predicate with the abstract value of its corresponding character in $\overline{\mathsf{p}}$. Formally:

$$\forall \mathrm{i} \in \overline{\mathsf{s}}'\ \forall \mathrm{k} \in [0, \mathsf{len}_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}) - 1]\colon\ \overline{\mathbf{c}}_\mathrm{i}\overline{\mathbf{b}}_\mathrm{i} \text{ is the corresponding segment of } \overline{\mathsf{p}}[\mathrm{k}] \wedge$$

$$\alpha_{\overline{\mathbf{C}}}(\overline{\mathsf{p}}[\mathrm{k}]) \sqsubseteq_{\overline{\mathbf{C}}} \overline{\mathbf{c}}_\mathrm{i} \Rightarrow \mathsf{fold}(\overline{\mathsf{s}}^* = \overline{\mathsf{s}}'[\overline{\mathbf{c}}_\mathrm{i}/\alpha_{\overline{\mathbf{C}}}(\overline{\mathsf{p}}[\mathrm{k}])])$$

b. if the matching segment in $\overline{\mathsf{s}}'$ of a character in $\overline{\mathsf{p}}$ is preceded by one or more segments possibly empty and none of those is the matching segment of the character preceding the first mentioned in $\overline{\mathsf{p}}$, then the refinement means removing those segments from $\overline{\mathsf{s}}'$. Formally:

$$\forall i \in \overline{\mathsf{s}}'\ \forall k \in [0, \mathsf{len}_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}) - 1]\text{: } \overline{\mathbf{c}}_i\overline{\mathbf{b}}_i \text{ is the corresponding segment of a } \overline{\mathsf{p}}[k] \wedge$$
$$\exists(\overline{\mathbf{c}}_j\overline{\mathbf{b}}_j)_{j=i-1}^{n\in[i-1,1]} \in \overline{\mathsf{s}}' : \overline{\mathbf{c}}_j\overline{\mathbf{b}}_j \text{ is not the corresponding segment of } \overline{\mathsf{p}}[k-1] \wedge$$
$$\overline{\mathsf{p}}[k] \notin \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_j) \wedge \overline{\mathbf{b}}_j \in \{[0,0],[0,1],[0,\mathtt{x}]\} \Rightarrow \mathsf{fold}(\overline{\mathsf{s}}^* = \overline{\mathsf{s}}' -_{\overline{\mathsf{s}}} (\overline{\mathbf{c}}_j\overline{\mathbf{b}}_j)_{j=i-1}^{n\in[i-1,1]})$$

The conditions above can occur at the same time. In this case both refinements apply. Observe that the order of application is not relevant.

If none of the conditions above apply, then $\overline{\mathsf{s}}^* = \overline{\mathsf{s}}$.

△

Example 4.12. Consider $\overline{\mathsf{s}}$ = 'a' $[2,4]$ 'b' $[1,1]$ 'c' $[0,2]$ and $\overline{\mathsf{p}}$ = aab$*$. Let $\overline{\mathsf{s}}'$ denote $\mathsf{unFold}(\overline{\mathsf{s}})$ = 'a' $[1,1]$ 'a' $[1,1]$ 'a' $[0,1]$ 'a' $[0,1]$ 'b' $[1,1]$ 'c' $[0,1]$ 'c' $[0,1]$. The refinement of $\overline{\mathsf{s}}$ through $\overline{\mathsf{p}}$ is as follows:

$$\begin{aligned}\mathsf{refine}_{\overline{\mathsf{p}}}(\overline{\mathsf{s}}) &= \mathsf{fold}(\overline{\mathsf{s}}' -_{\overline{\mathsf{s}}} (\overline{\mathbf{c}}_k\overline{\mathbf{b}}_k)_{k=4}^{3}) \\ &= \mathsf{fold}(\text{'a' } [1,1] \text{ 'a' } [1,1] \text{ 'b' } [1,1] \text{ 'c' } [0,1] \text{ 'c' } [0,1]) \\ &= \overline{\mathsf{s}} = \text{'a' } [2,2] \text{ 'b' } [1,1] \text{ 'c' } [0,2]\end{aligned}$$

Indeed, the fifth segment of $\overline{\mathsf{s}}'$ is the matching segment for $\overline{\mathsf{p}}[2]$ and it is preceded by two possibly empty segments and none of them is the matching segment of $\overline{\mathsf{p}}[1]$. So we could remove them form $\overline{\mathsf{s}}'$, as by case b. of Definition 4.8, thus refining it.

⬡

Definition 4.9 (Prefix refinement through Segmentation)**.** Let $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ be two abstract elements in the Segmentation and in the Prefix domains respectively. We define $\mathsf{refine}_{\overline{\mathsf{s}}}(\overline{\mathsf{p}})$ as the refinement operator describing how the information tracked by $\overline{\mathsf{s}}$ improves the one from $\overline{\mathsf{p}}$. Let $\mathsf{unFold}(\overline{\mathsf{s}}) = \overline{\mathsf{s}}'$. Formally, $\mathsf{refine}_{\overline{\mathsf{s}}}(\overline{\mathsf{p}}) = \overline{\mathsf{p}}^*$ where: if there is one or more segments in $\overline{\mathsf{s}}'$ after the one matching the last character of $\overline{\mathsf{p}}$, such that their segment abstract element approximates exactly one character and their bound is equal to the interval $[1,1]$. Then the refinement means concatenating that character(s) to $\overline{\mathsf{p}}$. Formally:

$$\exists i \in \overline{\mathsf{s}}'\text{: } \overline{\mathbf{c}}_i\overline{\mathbf{b}}_i \text{ is the corresponding segment of } \overline{\mathsf{p}}[\mathsf{len}_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}) - 1] \wedge$$
$$\exists(\overline{\mathbf{c}}_j\overline{\mathbf{b}}_j)_{j=i+1}^{n\in[i+1,\mathsf{numSeg}(\overline{\mathsf{s}}')]} \in \overline{\mathsf{s}}' : \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_j) = \{c\} \wedge$$
$$\overline{\mathbf{b}}_j = [1,1] \Rightarrow \overline{\mathsf{p}}^* = \overline{\mathsf{p}} + \langle c : \mathsf{char}_{\overline{\mathbf{C}}}(\overline{\mathbf{c}}_j) = \{c\} \wedge j \in [i+1, n]\rangle$$

△

Example 4.13. Consider $\overline{\mathsf{s}}$ = 'a' $[2,3]$ 'b' $[2,2]$ and $\overline{\mathsf{p}}$ = aab$*$. Let $\overline{\mathsf{s}}$' denote unfold($\overline{\mathsf{s}}$) = 'a' $[1,1]$ 'a' $[1,1]$'a' $[0,1]$ 'b' $[1,1]$ 'b' $[1,1]$. The refinement of $\overline{\mathsf{p}}$ through $\overline{\mathsf{s}}$ is as follows:

$$\begin{aligned}\mathsf{refine}_{\overline{\mathsf{s}}}(\overline{\mathsf{p}}) &= \mathrm{aaa} + \langle \mathrm{b} \rangle \\ &= \mathrm{aab} + \mathrm{b} \\ &= \mathrm{aabb}*\end{aligned}$$

Indeed, the segment of $\overline{\mathsf{s}}$' which corresponds to the last character in $\overline{\mathsf{p}}$ is followed by a segment whose character abstract element approximates exactly the character 'b' and its segment bound is the interval [1,1]. So we could concatenate to the sequence of character of $\overline{\mathsf{p}}$, the character approximated by $\overline{\mathsf{c}}_5$, as by Definition 4.9, thus refining it.

⬡

Lemma 4.10 (Reduced product between Segmentation and Prefix)**.** Let $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ be two abstract elements in the Segmentation and in the Prefix abstract domains respectively and, consider the refinement operators presented in the Definitions 4.8 and 4.9. The reduced product of $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ is obtained by applying $\mathsf{refine}_{\overline{\mathsf{p}}}$ and $\mathsf{refine}_{\overline{\mathsf{s}}}$ as follows:

$$\overline{\mathsf{s}} \otimes \overline{\mathsf{p}} = (\mathsf{refine}_{\overline{\mathsf{p}}}(\overline{\mathsf{s}}), \mathsf{refine}_{\overline{\mathsf{s}}}(\overline{\mathsf{p}}))$$

Lemma 4.11 (Soundness of Segmentation refinement)**.** Let $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ be two abstract elements in the Segmentation and in the Prefix abstract domains respectively, and let $\gamma_{\overline{\mathbf{S}}}$, and let $\gamma_{\overline{\mathbf{PR}}}$ be the concretization functions of the domains just mentioned. Moreover, $\mathsf{refine}_{\overline{\mathsf{p}}}(\overline{\mathsf{s}}) = \overline{\mathsf{s}}^*$. The soundness of the Segmentation refinement can be expressed by:

$$\sigma \in [\gamma_{\overline{\mathbf{S}}}(\overline{\mathsf{s}}) \setminus \gamma_{\overline{\mathbf{S}}}(\overline{\mathsf{s}}^*)] \Rightarrow \sigma \notin \gamma_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}})$$

Proof.
The set $[\gamma_{\overline{\mathbf{S}}}(\overline{\mathsf{s}}) \setminus \gamma_{\overline{\mathbf{S}}}(\overline{\mathsf{s}}^*)]$ contains representations of strings which do not share the same longest common prefix with the strings approximated by $\overline{\mathsf{p}}$, either in case a. and in case b. of Definition 4.8. Thus, those strings do not belong to $\gamma_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}})$, by definition of $\gamma_{\overline{\mathbf{PR}}}$ itself.

□

Lemma 4.12 (Soundness of Prefix refinement)**.** Let $\overline{\mathsf{s}}$ and $\overline{\mathsf{p}}$ be two abstract elements in the Segmentation and in the Prefix abstract domains respectively, and let $\gamma_{\overline{\mathbf{S}}}$ and $\gamma_{\overline{\mathbf{PR}}}$ be the concretization functions of the domains just mentioned. Moreover, $\mathsf{refine}_{\overline{\mathsf{s}}}(\overline{\mathsf{p}}) = \overline{\mathsf{p}}^*$. The soundness of the Prefix refinement can be expressed by:

$$\sigma \in [\gamma_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}) \setminus \gamma_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}^*)] \Rightarrow \sigma \notin \gamma_{\overline{\mathbf{S}}}(\overline{\mathsf{s}})$$

Proof.
The set $[\gamma_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}) \setminus \gamma_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}^*)]$ contains representations of strings that do not share the same longest common prefix with the strings approximated by $\overline{\mathsf{s}}$, as by Definition 4.9. Thus,

those strings do not belong to $\gamma_{\overline{\mathbf{S}}}(\overline{\mathbf{s}})$, by definition of $\gamma_{\overline{\mathbf{S}}}$ itself.

□

4.7 Discussion

Strings are characterized by their content and shape. Existing string abstract domains capture information about the content or the shape of strings separately or they are able to track both but partially, leading to a consistent loss of precision during the analysis. The Segmentation domain specialised for string analysis tracks information about the shape and the content of strings in their entirety. Its precision depends on the abstract domains for characters, segment bound and variables it has been instantiated. Preciser domains lead the segmentation analysis efficiency to decrease inevitably, on the other hand, less precise domains increase the efficiency of the segmentation analysis at the expense of lost of important information, e.g., information useful to prevent undesired behaviours of the considered program.

By lifting basic domains for strings by segmentation abstraction, we generate more sophisticated domains where structural information and character-based information are combined together yielding to a more accurate representation.

Chapter 5

COMPLETENESS OF STRING DOMAINS

In this chapter we study the problem of completeness of string abstractions, and considering two well known string abstract domains for `JavaScript`, we show how to complete them. Moreover, we provide a procedure to measure the precision gained by complete domain with respect to its original version.

The contribution of this chapter was published in [18].

Chapter Structure

Section 5.2 recalls relevant concepts related to the completeness property in Abstract Interpretation. Moreover, a motivating example is given to show the importance of guaranteeing completeness in an Abstract Interpretation-based analysis with respect to strings. Section 5.3 introduces the syntax and the concrete semantics of the dynamic imperative core language used in this chapter. Section 5.4 presents the completion of two well-known string abstract domains with respect to two operations of interest. Section 5.5 highlights the strengths and usefulness of the completeness approach to static analysis of string manipulating programs. Section 5.6 defines the measurement procedure of the analysis precision increment. Section 5.7 concludes.

5.1 Introduction

Important features of Abstract Interpretation are *soundness* and *completeness* [56] (cf. Section 2.6.2). If soundness (or correctness), as a basic requirement, actually is always guaranteed by static analysis tools, in order to avoid the presence of false negatives, completeness is frequently not met.

If completeness is satisfied, it means that the abstract computations do not lose information, during the abstraction process, with respect to a property of interest, and so the analysis can be considered optimal. Guaranteeing completeness of string analysis is of particular interest, especially for dynamic languages, as poorly managed string manipulation code may easily lead to significant security flaws. For instance, in the context of

web applications, the common programming languages used for the web-based software development (e.g., `JavaScript`) offer a wide range of dynamic features that make string manipulation dangerous.

In [80], Giacobazzi et al. highlighted the fact that completeness is an abstract domain property, they presented a methodology to obtain complete abstract domains with respect to operations, by minimally extending or restricting the underlying domains.

Contribution

We provide a way-to-proceed in the context of imprecise string abstractions. In particular, we exploit the theoretical framework of the complete shells, constructively showing how to improve precision of incomplete abstraction, without designing new string abstract domains.

We consider two `JavaScript` string abstract domains defined as part of TAJS [105] and SAFE [122] static analysers, focusing on their completeness with respect to the main string-manipulating operations. We compute their complete versions, and we discuss the benefits of guaranteeing completeness in the context of Abstract Interpretation-based string analysis of dynamic languages. Finally, we present an effective procedure to measure the precision increment when analysing a program with a complete abstract domain.

5.2 Making Abstract Interpretation Complete

In this section, we introduce the notions and methodologies that we will use through the whole chapter (and proposed in [80]), in order to constructively build, from an initial abstract domain, a new abstract domain that is complete w.r.t. an operation of interest. Finally, a motivating example showing the usefulness of completion of abstract domains for string analysis is given.

It is worth noting that completeness is a property related to the underlying abstract domain. Starting from this fact, Giacobazzi et al. proposed a constructive method to manipulate the underlying incomplete abstract domain in order to get a complete abstract domain w.r.t. a certain operation. In particular, given two abstract domains $\overline{\mathbf{D}}$ and $\overline{\mathbf{D}}'$ and an operator $f : \overline{\mathbf{D}}^n \to \overline{\mathbf{D}}'$ with $n \in \mathbb{N}$, the authors gave two different notions of completion of abstract domains w.r.t. f: the one that adds the minimal number of abstract points to the input abstract domain $\overline{\mathbf{D}}$ and the other that removes the minimal number of abstract points from the output abstract domain $\overline{\mathbf{D}}'$. The first approach captures the notion of complete shell of $\overline{\mathbf{D}}$, while the latter defines the complete core of $\overline{\mathbf{D}}'$, both w.r.t. an operator f.

5.2.1 Complete Shell vs Complete Core

We will focus on the construction of complete shells of string abstract domains, rather than complete cores. This choice is guided by the fact that a complete core for an operation f removes abstract points from a starting abstract domain, and so, even if it is complete for f, the complete core could worsen the precision of other operations.

Conversely, complete shells augment the starting abstract domains (adding abstract points), and consequently it cannot compromise the precision of other operations.

Below, we recall two important theorems proved in [80] that provide a constructive method to compute abstract domain complete shells, defined in terms of an upper closure operator ρ (cf. Section 2.2). Precisely, the latter theorems present two notions of complete shells:

- complete shells of ρ relative to η (where η is an upper closure operator), meaning that they are complete shells of operations defined on ρ that return results in η, and
- absolute complete shells of ρ, meaning that they are complete shells of operations that are defined on ρ and return results in ρ.

Definition 5.1 (Complete shell of ρ relative to η [80])**.** Let $\langle A, \sqsubseteq_A, \sqcup_A\rangle$ and $\langle B, \sqsubseteq_B, \sqcup_B\rangle$ be two posets and $f : A^n \to B$ be a continuous function. Given $\rho \in uco(A)$ and $\eta \in uco(B)$, then let $\mathcal{S}_f^\eta : uco(A) \to uco(A)$ be the domain transformer:

$$\mathcal{S}_f^\eta(\rho) = \bigsqcup_{uco(A)} \{\delta \in uco(A) \mid \delta \sqsubseteq \rho, \langle \delta, \eta\rangle \in \Gamma(A, B, f)\}.$$

If $\langle \mathcal{S}_f^\eta(\rho), \eta\rangle \in \Gamma(A, B, f)$, then $\mathcal{S}_f^\eta(\rho)$ is called complete shell of ρ relative to η with respect to an operation f.

△

As discussed in [80], Definition 5.1 does not offer a constructive methodology to compute $\mathcal{S}_f^\eta(\rho)$. Theorem 5.1 reports a constructive characterization of the complete shell of ρ relative to η w.r.t. f, making use of the Moore closure operator defined in Section 2.2.

Theorem 5.1 ([80])**.** Let $\langle A, \sqsubseteq_A, \sqcup_A\rangle$ and $\langle B, \sqsubseteq_B, \sqcup_B\rangle$ be two posets and $f : A^n \to B$ be a continuous function. Given $\rho \in uco(A)$, $\eta \in uco(B)$ and $\mathcal{S}_f^\eta(\rho)$ as in Definition 5.1, the following equality holds:

$$\mathcal{S}_f^\eta(\rho) = \mathcal{M}(\rho \cup (\bigsqcup_{\substack{A \\ i\in[1,n), \\ x\in A^n,\ y\in\eta}} \mathsf{max}^{\sqsubseteq_A}(\{z \in A \mid (f_x^i)(z) \sqsubseteq_B y\}))).$$

□

As already mentioned above, the idea behind the complete shell of ρ (input abstraction) relative to η (output abstraction) is to refine ρ adding the minimum number of abstract points to make ρ complete w.r.t. an operation f. By Theorem 5.1, this is obtained adding to ρ the maximal elements in A, whose f image is dominated by elements in η, at least in one dimension i. Clearly, the so-obtained abstraction may not be an upper closure operator for A. Hence, the Moore closure operator is applied. On the other hand, absolute complete shells are involved in the case in which the operator f of interest has same input and output abstract domain, i.e., $\mathrm{f} : \mathrm{A}^{\mathrm{n}} \to \mathrm{A}$. In this case, given $\rho \in uco(\mathrm{A})$, absolute complete shells of ρ can be obtained as the greatest fix-point (gfp) of the domain transformer presented in Definition 5.1, as stated below.

Definition 5.2 (Absolute complete shell of ρ [80])**.** Let $\langle \mathrm{A}, \sqsubseteq_{\mathrm{A}}, \sqcup_{\mathrm{A}} \rangle$ be a poset and $\mathrm{f} : \mathrm{A}^{\mathrm{n}} \to \mathrm{A}$ be a continuous function. Given $\rho \in uco(\mathrm{A})$, then let $\overline{\mathcal{S}}_{\mathrm{f}} : uco(\mathrm{A}) \to uco(\mathrm{A})$ be the domain transformer:

$$\overline{\mathcal{S}}_{\mathrm{f}}(\rho) = \bigsqcup_{uco(\mathrm{A})} \{\delta \in uco(\mathrm{A}) \mid \delta \sqsubseteq \rho, \langle \delta, \delta \rangle \in \Gamma(\mathrm{A}, \mathrm{A}, \mathrm{f})\}.$$

If $\langle \overline{\mathcal{S}}_{\mathrm{f}}(\rho), \overline{\mathcal{S}}_{\mathrm{f}}(\rho) \rangle \in \Gamma(\mathrm{A}, \mathrm{A}, \mathrm{f})$, then $\overline{\mathcal{S}}_{\mathrm{f}}(\rho)$ is called absolute complete shell of ρ with respect to an operation f.

$\triangle$

Theorem 5.2 ([80])**.** Let A be a poset and $\mathrm{f} : \mathrm{A}^{\mathrm{n}} \to \mathrm{A}$ be a continuous function. For $\rho \in uco(\mathrm{A})$, if $\mathcal{S}_{\mathrm{f}}^{\rho}(\rho)$ is the complete shell of ρ relative to ρ w.r.t. f, and if $\overline{\mathcal{S}}_{\mathrm{f}}(\rho)$ is the absolute complete shell of ρ w.r.t. f, then the following equality holds:

$$\overline{\mathcal{S}}_{\mathrm{f}}(\rho) = \mathsf{gfp}(\lambda\rho.\mathcal{S}_{\mathrm{f}}^{\rho}(\rho)).$$

$\square$

In [80], Giacobazzi et al. have discussed the completeness and incompleteness of the Sign numerical abstract domain. In particular, $\mathsf{Sign} = \{\mathsf{NotPos}, \mathsf{NotNeg}\} \cup \{\bot_{\mathsf{Sign}}, \top_{\mathsf{Sign}}\}$, where: negative numbers are approximated by the abstract value NotPos, positive numbers by NotNeg, and $\bot_{\mathsf{Sign}}$ and $\top_{\mathsf{Sign}}$ are the bottom and the top elements of Sign respectively. Sign is complete for the product operation. Let $* : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ be the concrete product operation and $\circledast_{\mathsf{Sign}} : \mathsf{Sign} \times \mathsf{Sign} \to \mathsf{Sign}$ be the corresponding abstract product operation. Given the expression $\mathsf{e}_1 * \mathsf{e}_2$, the equality $\alpha_{\mathsf{Sign}}(\mathsf{e}_1 * \mathsf{e}_2) = \alpha_{\mathsf{Sign}}(\mathsf{e}_1) \circledast_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}(\mathsf{e}_2)$ holds, with α_{Sign} being the abstraction function of Sign. As an example, consider the concrete

expression $\{2,5\} * \{-1,-3\}$, then

$$\begin{aligned}
\alpha_{\mathsf{Sign}}(\{2,5\} * \{-1,-3\}) &= \alpha_{\mathsf{Sign}}(\{-2,-5,-6,-15\}) \\
&= \mathsf{NotPos} \\
&= \alpha_{\mathsf{Sign}}(\{2,5\}) \circledast_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}(\{-1,-3\}) \\
&= \mathsf{NotNeg} \circledast_{\mathsf{Sign}} \mathsf{NotPos} \\
&= \mathsf{NotPos}
\end{aligned}$$

On the other hand, Sign is not complete for the sum operation. Let $+ : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ be the concrete sum operation and $\oplus_{\mathsf{Sign}} : \mathsf{Sign} \times \mathsf{Sign} \to \mathsf{Sign}$ be the corresponding abstract sum operation. Consider the concrete expression $\{2\} + \{-1,-2\}$, then

$$\begin{aligned}
\alpha_{\mathsf{Sign}}(\{2\} + \{-1,-2\}) &= \alpha_{\mathsf{Sign}}(\{0,1\}) \\
&= \mathsf{NotNeg} \\
&\neq \alpha_{\mathsf{Sign}}(\{2\}) \oplus_{\mathsf{Sign}} \alpha_{\mathsf{Sign}}(\{-1,-2\}) \\
&= \mathsf{NotNeg} \oplus_{\mathsf{Sign}} \mathsf{NotPos} \\
&= \mathbb{Z}
\end{aligned}$$

In [80], the absolute complete shell of Sign w.r.t. the sum operation has been computed and it corresponds to the interval abstract domain [51].

5.2.2 Domain Completion Procedure

To improve the understanding about how to obtain a complete domain w.r.t. an operation of interest we provide a step-by-step reading of the formula in Theorem 5.1 (i.e., relative complete shell) by means of the procedure reported in Algorithm 6. The algorithm takes as input two posets $\langle \mathrm{A}, \sqsubseteq_{\mathrm{A}} \rangle$, $\langle \mathrm{B}, \sqsubseteq_{\mathrm{B}} \rangle$, two closures $\rho \in \mathit{uco}(\mathrm{A})$, $\eta \in \mathit{uco}(\mathrm{B})$, which respectively correspond to the input and output abstraction, and a continuous function $\mathrm{f} : \mathrm{A}^{\mathrm{n}} \to \mathrm{B}$. The procedure returns the complete shell of ρ relative to η w.r.t. f. Algorithm 6 follows Theorem 5.1 and collects in X, for each dimension i and element of ρ, any element $\mathrm{z} \in \mathrm{A}$ whose f image is dominated by elements in η (lines 2-4). Then, the starting input abstraction ρ is joined with the new elements collected in X and since the so obtained result may not be a closure, the Moore closure operator is applied (line 5). Finally, the complete shell of ρ relative to η is returned at line 6.

We reported the procedure only to improve the understanding of the complete shell since unfortunately, this is not an effective (decidable) procedure. Indeed, Algorithm 6 diverges (at lines 2-4) when A is an infinite set or η is not a finite closure.

Algorithm 6 Relative complete shell procedure pseudo-code.

Input:

- $\langle A, \sqsubseteq_A \rangle$, $\langle B, \sqsubseteq_B \rangle$ (posets)
- $\rho \in uco(A)$, $\eta \in uco(B)$ (abstractions)
- $f : A^n \to B$

Output: $\mathcal{S}_f^\eta(\rho)$

1: $X \leftarrow \emptyset$

2: **for** $i \in [1, n]$ and $x \in A$ and $y \in \eta$ **do**

3: **let** $z \in A$ be the maximum element such that $f_x^i(z) \sqsubseteq_B y$

4: **add** z to X

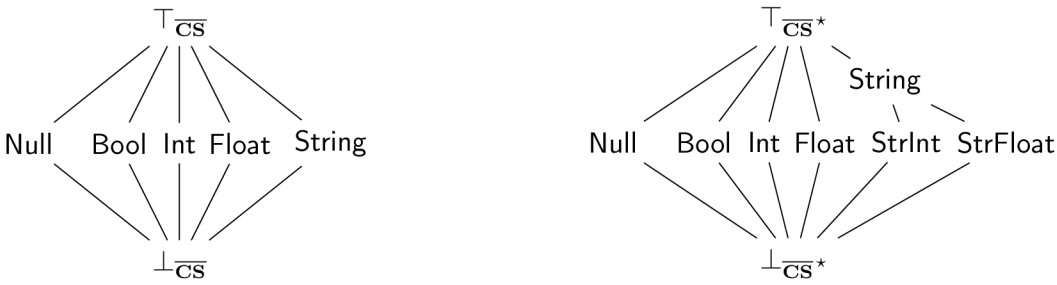
5: **let** $\mathcal{S}_f^\eta(\rho)$ be the Moore closure of $\rho \cup X$

6: **return** $\mathcal{S}_f^\eta(\rho)$

5.2.3 Motivating Example

A common feature of dynamic languages, such as `PHP` or `JavaScript`, is to be weakly typed. Hence, in those languages, it is allowed to change the variable type through the program execution. For example, in `PHP`, it is completely legal to write fragments such as `x = 1;x= true;`, where the type of the variable `x` changes from integer to boolean. The first attempt for statically reasoning about variable types was adopting the so-called Coalesced Sum abstract domain ($\overline{\mathbf{CS}}$) [13, 118], in order to detect whether a certain variable has constant type through the whole program execution. In Figure 5.1a, we report the Coalesced Sum abstract domain for an intra-procedural version of `PHP` [13], that tracks null, boolean, integer, float and string types[1]. Consider the formal semantics of the sum operation in `PHP` [67]. When one of the operand is a string, since the sum operation is feasible only between numbers, *implicit type conversion* occurs and converts the operand string to a number. In particular, if the prefix of the string is a number, it is converted to the maximum prefix of the string corresponding to a number, otherwise it is converted to 0. For example, the expression `e = "2.4hello" + "4"` returns `6.4`. Let $\alpha_{\overline{\mathbf{CS}}}$ and $\oplus_{\overline{\mathbf{CS}}}$ be the abstraction function and the abstract sum operation on the Coalesced Sum abstract domain respectively. The type of the expression `e` is given by: $\alpha_{\overline{\mathbf{CS}}}(\{\texttt{"2.4hello"}\}) \oplus_{\overline{\mathbf{CS}}} \alpha_{\overline{\mathbf{CS}}}(\{\texttt{"4"}\}) = \mathsf{String} \oplus_{\overline{\mathbf{CS}}} \mathsf{String} = \top_{\overline{\mathbf{CS}}}$. The static type analysis based on the Coalesced

[1] By closing the Coalesced Sum abstract domain with the powerset operation we get a more precise domain called Union Type abstract domain [118], that tracks the set of types of a certain variable during program execution.



(a) (b)

FIGURE 5.1: (a) Coalesced Sum abstract domain for PHP. (b) Complete shell of Coalesced Sum abstract domain w.r.t. the sum operation.

Sum abstract domain returns $\top_{\overline{\mathbf{CS}}}$ (i.e., any possible value), since the sum between two strings may return either an integer or a float value. Precisely, the Coalesced Sum abstract domain is not complete w.r.t. the PHP sum operation, since for any string σ and σ', it does not meet the completeness condition, i.e., $\alpha_{\overline{\mathbf{CS}}}(\sigma \texttt{ + } \sigma') = \alpha_{\overline{\mathbf{CS}}}(\sigma) \oplus_{\overline{\mathbf{CS}}} \alpha_{\overline{\mathbf{CS}}}(\sigma')$. Indeed, $\alpha_{\overline{\mathbf{CS}}}(\{\texttt{"2.4hello" + "4"}\}) = \mathsf{Float}$ that is different from $\alpha_{\overline{\mathbf{CS}}}(\{\texttt{"2.4hello"}\}) \oplus_{\overline{\mathbf{CS}}} \alpha_{\overline{\mathbf{CS}}}(\{\texttt{"4"}\}) = \top_{\overline{\mathbf{CS}}}$.

Intuitively, the Coalesced Sum abstract domain is not complete w.r.t. the sum operation due to the loss of precision that occurs during the abstraction process of the inputs, since the domain is not precise enough to distinguish between strings that may be implicitly converted to integers or floats.

Figure 5.1b shows the complete shell of the Coalesced Sum abstract domain w.r.t. the sum ($\overline{\mathbf{CS}}^\star$). The latter adds two abstract values to the original domain, namely StrFloat and StrInt, that correspond to the abstractions of the strings that may be implicitly converted to floats and to integers, respectively. Notice that the type analysis on the new abstract domain is now complete w.r.t. the sum operation, e.g., $\alpha_{\overline{\mathbf{CS}}^\star}(\{\texttt{"2.4hello" + "4"}\}) = \mathsf{Float}$ that is equal to $\alpha_{\overline{\mathbf{CS}}^\star}(\{\texttt{"2.4hello"}\}) \oplus_{\overline{\mathbf{CS}}^\star} \alpha_{\overline{\mathbf{CS}}^\star}(\{\texttt{"4"}\}) = \mathsf{StrFloat} \oplus_{\overline{\mathbf{CS}}^\star} \mathsf{StrInt} = \mathsf{Float}$.

As pointed out above, guaranteeing completeness in Abstract Interpretation is a desirable property that an abstract domain should aim to, since it ensures that no loss of precision occurs during the input abstraction process of the operation of interest. It is worth noting that *guessing* a complete abstract domain for a certain operation becomes particularly hard when the operation has a tricky semantics, such as in our example or, more in general, in dynamic languages operations. For this reason, complete shells become important since they are able to mathematically guarantee completeness for a certain operation, starting from an abstract domain of interest.

```
a ::= n | x | a + a | a - a | a * a | a \ a | toNum(s) | length(s)
b ::= x | true | false | b && b | b || b | ! b
s ::= x | "σ" | concat(s1,s2)
e ::= a | b | s
st ::= x = e; | if (b) { st } else { st } | while (b) { st }
   | st1 st2
P ∈ μDyn ::= st

where x ∈ Id, σ ∈ Σ*, n ∈ Int ∪ Float
```

FIGURE 5.2: μDyn syntax

5.3 Core Language

We define μDyn, an imperative toy language expressive enough to handle some interesting behaviors related to strings in dynamic languages, e.g., implicit type conversion, and inspired by the `JavaScript` programming language [128].

Note that, in this chapter, the semantics notation, both concrete and abstract, is different from the one presented in the previous chapters, in order to be consistent with the notation used in [18].

5.3.1 Syntax

μDyn syntax is reported in Figure 5.2. The μDyn basic values are represented by the set $\textsc{Val} = \textsc{Int} \cup \textsc{Float} \cup \textsc{Bool} \cup \textsc{Str}$, such that $\textsc{Int}$ denotes the set of signed integers $\mathbb{Z}$, $\textsc{Float}$ denotes the set of signed decimal numbers[2], $\textsc{Bool}$ denotes the set of booleans $\{\texttt{true}, \texttt{false}\}$, and $\textsc{Str}$ denotes the set of strings Σ^* over an alphabet Σ. Then, in this chapter, we consider $\Sigma^* = \Sigma^*_{\mathsf{Num}} \cup \Sigma^*_{\mathsf{NotNum}}$, where:

Σ^*_{Num} is the set of numeric strings (e.g., `"42"`, `"-7.2"`), and

$\Sigma^*_{\mathsf{NotNum}}$ is the set of non numeric strings (e.g., `"foo"`, `"-2a"`).

Moreover, we consider Σ^*_{Num} additionally composed of four sets:

$$\Sigma^*_{\mathsf{Num}} = \Sigma^*_{\mathsf{UInt}} \cup \Sigma^*_{\mathsf{UFloat}} \cup \Sigma^*_{\mathsf{SInt}} \cup \Sigma^*_{\mathsf{SFloat}},$$

which correspond, from left to right, to the set of unsigned integer strings, unsigned float strings, signed integer strings and signed float strings, respectively.

[2]In general, floats are represented in programming languages in the IEEE 754 double precision format. For the sake of simplicity, we use instead decimal numbers.

$$
\begin{aligned}
[\![\mathtt{x} = \mathtt{e};]\!]\xi &= \xi[\mathtt{x} \leftarrow [\![\mathtt{e}]\!]\xi] \\
[\![\mathtt{if}\ (\mathtt{b})\ \mathtt{bl}_1\ \mathtt{else}\ \mathtt{bl}_2]\!]\xi &= \begin{cases} [\![\mathtt{bl}_1]\!]\xi & [\![\mathtt{b}]\!]\xi = \mathtt{true} \\ [\![\mathtt{bl}_2]\!]\xi & [\![\mathtt{b}]\!]\xi = \mathtt{false} \end{cases} \\
[\![\mathtt{while}\ (\mathtt{b})\ \mathtt{bl}]\!]\xi &= [\![\mathtt{if}\ (\mathtt{b})\ \{\mathtt{bl}\ \mathtt{while}\ (\mathtt{b})\ \mathtt{bl}\}\ \mathtt{else}\ \{\}]\!]\xi \\
[\![\{\}]\!]\xi &= \xi \qquad [\![\{\mathtt{st}\}]\!]\xi = [\![\mathtt{st}]\!]\xi \\
[\![\mathtt{st}_1\mathtt{st}_2]\!]\xi &= [\![\mathtt{st}_2]\!]([\![\mathtt{st}_1]\!]\xi)
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathtt{concat}(\mathtt{s},\mathtt{s'})]\!]\xi &= [\![\mathtt{s}]\!]\xi \cdot [\![\mathtt{s'}]\!]\xi \qquad [\![\mathtt{length}(\mathtt{s})]\!]\xi = |[\![\mathtt{s}]\!]\xi| \\
[\![\mathtt{toNum}(\mathtt{s})]\!]\xi &= \begin{cases} \mathcal{N}([\![\mathtt{s}]\!]\xi) & [\![\mathtt{s}]\!]\xi \in \Sigma^*_{\mathsf{Num}} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

FIGURE 5.3: μDyn semantics

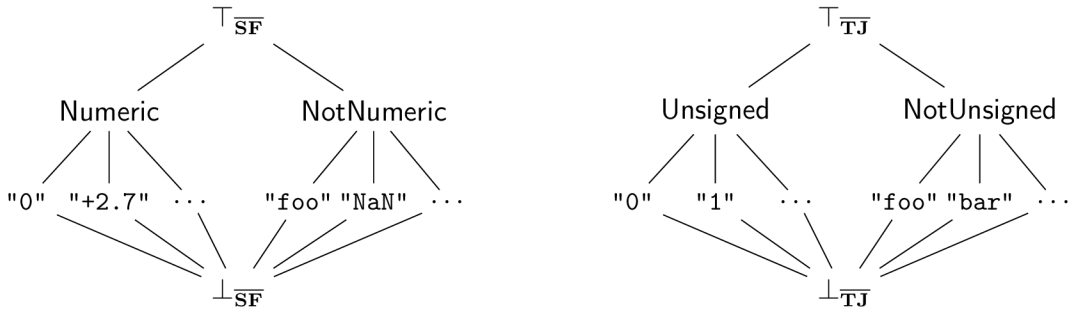
5.3.2 Concrete Semantics

μDyn programs are elements generated by `st` syntax rules. Program states $\textsc{State} : \textsc{Id} \rightarrow \textsc{Val}$, ranging over ξ, are maps from identifiers to values. The concrete semantics of μDyn statements follows[3] [13], and it is given by the function $[\![\cdot]\!]\cdot : \textsc{Stmt} \times \textsc{State} \rightarrow \textsc{State}$, inductively defined on the structure of the statements, as reported in Figure 5.3. We abuse notation in defining the concrete semantics of expressions: $[\![\cdot]\!]\cdot : \textsc{Exp} \times \textsc{State} \rightarrow \textsc{Val}$. Figure 5.3 shows the formal semantics of two relevant expressions involving strings we focus on: `concat`, that concatenates two strings, and string-to-number operation, namely `toNum`, that takes a string as input and returns the number that it represents if the input string corresponds to a numerical strings, 0 otherwise. Given $\sigma \in \textsc{Str}$, we denote by $\mathcal{N}(\sigma) \in \textsc{Int} \cup \textsc{Float}$ the numeric value of a given string. For example, `toNum("4.2")` $= 4.2$ and `toNum("asd")` $= 0$.

5.4 Making JavaScript String Abstract Domains Complete

In this section, we study the completeness of two string abstract domains integrated into two state-of-the-art `JavaScript` static analysers based on Abstract Interpretation, namely TAJS [105] and SAFE [122]. Both the abstract domains track important information on `JavaScript` strings, e.g., TAJS is able to infer when a string corresponds to an unsigned integer, that may be used as array index, and SAFE tracks numeric strings, such as `"2.5"` or `"+5"`.

[3]Note that we give the semantics of the operations that are of interest for this chapter. The complete concrete semantics of μDyn is defined in [13].



(a) (b)

FIGURE 5.4: (a) SAFE, (b) TAJS recasted for μDyn.

For the sake of readability, we recast the original string abstract domains for μDyn, following the notation adopted in [10]. Figure 5.4 depicts them. Notice that the original abstract domain part of SAFE analyser treats the string `"NaN"` as a numeric string. Since our core language does not provide the primitive value `NaN`, the corresponding string, i.e., `"NaN"`, has no particular meaning here, and it is treated as a non-numerical string.

5.4.1 Completing TAJS String Abstract Domain

Figure 5.4b depicts the string abstract domain $\overline{\mathbf{TJ}}$, the recasted version of the domain integrated into the TAJS static analyser [105]. $\overline{\mathbf{TJ}}$ splits the strings into Unsigned, that denotes the strings corresponding to unsigned numbers, and NotUnsigned, any other string. Hence, for example, $\alpha_{\overline{\mathbf{TJ}}}(\{\texttt{"9"}, \texttt{"+9"}\}) = \top_{\overline{\mathbf{TJ}}}$ and $\alpha_{\overline{\mathbf{TJ}}}(\{\texttt{"9.2"}, \texttt{"foo"}\}) = \mathsf{NotUnsigned}$. Before reaching these abstract values, $\overline{\mathbf{TJ}}$ precisely tracks each string.

Here we focus on the `toNum` (i.e., string-to-number) operation. Since this operation clearly involves numbers, we introduce the TAJS numerical abstract domain, denoted by $\overline{\mathbf{TJ}}_{\mathsf{N}}$. The latter domain behaves similarly to $\overline{\mathbf{TJ}}$, distinguishing between unsigned (UnsignedInt) and not unsigned integers (NotUnsignedInt).

Below we define the abstract semantics of the string-to-number operation for $\overline{\mathbf{TJ}}$. In particular, we define the function: $[\![\texttt{toNum}(\bullet)]\!]^{\overline{\mathbf{TJ}}} : \overline{\mathbf{TJ}} \to \overline{\mathbf{TJ}}_{\mathsf{N}}$ that takes as input a string abstract value in $\overline{\mathbf{TJ}}$, and returns an integer abstract value in $\overline{\mathbf{TJ}}_{\mathsf{N}}$. Formally,

$$[\![\texttt{toNum}(\texttt{s})]\!]^{\overline{\mathbf{TJ}}} = \begin{cases} \bot_{\overline{\mathbf{TJ}}_{\mathsf{N}}} & [\![\texttt{s}]\!]^{\overline{\mathbf{TJ}}} = \bot_{\overline{\mathbf{TJ}}} \\ [\![\texttt{toNum}(\sigma)]\!] & [\![\texttt{s}]\!]^{\overline{\mathbf{TJ}}} = \sigma \\ \mathsf{UnsignedInt} & [\![\texttt{s}]\!]^{\overline{\mathbf{TJ}}} = \mathsf{Unsigned} \\ \top_{\overline{\mathbf{TJ}}_{\mathsf{N}}} & [\![\texttt{s}]\!]^{\overline{\mathbf{TJ}}} = \mathsf{NotUnsigned} \vee [\![\texttt{s}]\!]^{\overline{\mathbf{TJ}}} = \top_{\overline{\mathbf{TJ}}} \end{cases}$$

When the input evaluates to $\bot_{\overline{\mathbf{TJ}}}$, bottom is propagated and $\bot_{\overline{\mathbf{TJ}}_{\mathsf{N}}}$ is returned (first row). If the input evaluates to a single string value, the abstract semantics relies on its concrete one (second row), as single strings are precisely captured by $\overline{\mathbf{TJ}}$. When the input

evaluates to the string abstract value Unsigned, the integer abstract value UnsignedInt is returned (third row). Indeed, in the concrete scenario, an unsigned and not float numeric string, which exactly represents the strings approximated by Unsigned in $\overline{\mathbf{TJ}}$, is converted into the correspondent numeric value by the string-to-number operation. Therefore, the abstraction in $\overline{\mathbf{TJ}}_{\mathsf{N}}$ of the numeric value of all the strings approximated by Unsigned is UnsignedInt. Finally, when the input evaluates to NotUnsigned or $\top_{\overline{\mathbf{TJ}}}$, the top abstract value $\top_{\overline{\mathbf{TJ}}_{\mathsf{N}}}$ is returned (fourth row). In the second case, i.e., when the input is evaluated to $\top_{\overline{\mathbf{TJ}}}$, it is trivial to notice that the result of the abstract string-to-number operation represents the best correct approximation. On the other hand, in the first case, it is not straightforward. Note that NotUnsigned approximates not numeric strings and signed and/or float numeric strings, thus the only safe abstraction of the numeric value of strings approximated by NotUnsigned is $\top_{\overline{\mathbf{TJ}}_{\mathsf{N}}}$.

Lemma 5.1. The $\overline{\mathbf{TJ}}$ abstract domain is not complete with respect to the `toNum` operation. In particular[4], $\exists \mathrm{S} \in \mathcal{P}(\Sigma^*)$ such that:

$$\alpha_{\overline{\mathbf{TJ}}_{\mathsf{N}}}([\![\mathtt{toNum}(\mathrm{S})]\!]) \subsetneq [\![\mathtt{toNum}(\alpha_{\overline{\mathbf{TJ}}}(\mathrm{S}))]\!]^{\overline{\mathbf{TJ}}}$$

Proof.
As a counterexample to completeness, consider $\mathrm{S} = \{\mathtt{"2.3"}, \mathtt{"3.4"}\}$. We can show that $\alpha_{\overline{\mathbf{TJ}}_{\mathsf{N}}}([\![\mathtt{toNum}(\mathrm{S})]\!]) \neq [\![\mathtt{toNum}(\alpha_{\overline{\mathbf{TJ}}}(\mathrm{S}))]\!]^{\overline{\mathbf{TJ}}}$. Indeed,

$$\begin{aligned}
\alpha_{\overline{\mathbf{TJ}}_{\mathsf{N}}}([\![\mathtt{toNum}(\mathrm{S})]\!]) &= \mathsf{NotUnsignedInt} \\
&\neq [\![\mathtt{toNum}(\alpha_{\overline{\mathbf{TJ}}}(\mathrm{S}))]\!]^{\overline{\mathbf{TJ}}} \\
&= [\![\mathtt{toNum}(\mathsf{NotUnsigned})]\!]^{\overline{\mathbf{TJ}}} \\
&= \top_{\overline{\mathbf{TJ}}_{\mathsf{N}}}
\end{aligned}$$

□

The completeness condition does not hold because the $\overline{\mathbf{TJ}}$ string abstract domain loses too much information during the abstraction process, and the latter information cannot be retrieved during the abstract `toNum` operation. In particular, when non-numeric strings and unsigned integer strings are converted to numbers by `toNum`, they are mapped to the same value, namely 0. Indeed, $\overline{\mathbf{TJ}}$ does not differentiate between non-numeric and unsigned integer string values, and this is the principal cause of the $\overline{\mathbf{TJ}}$ incompleteness w.r.t. `toNum`. Additionally, more precision can be obtained if we could differentiate numeric strings holding float numbers from those holding integer numbers. Thus, in order to make

[4]We abuse notation denoting with $[\![\cdot]\!]$ the additive lift to set of basic values of the concrete semantics, i.e., the collecting semantics.

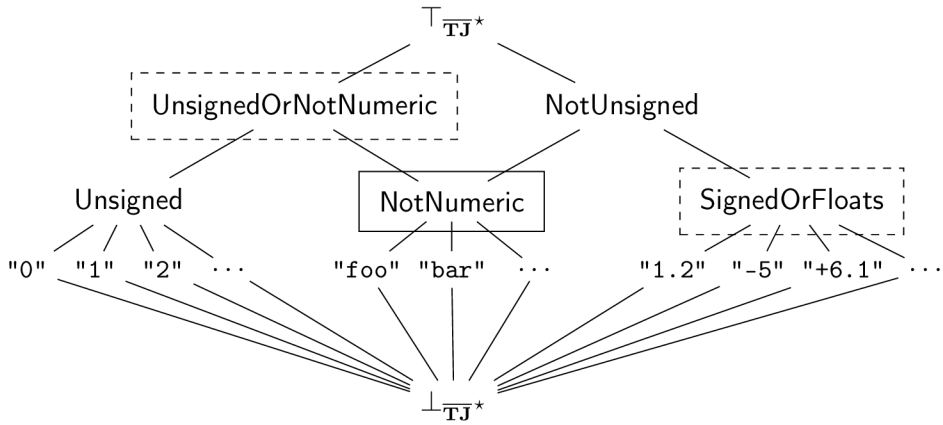


FIGURE 5.5: Complete shell of $\rho_{\overline{\mathbf{TJ}}}$ relative to $\rho_{\overline{\mathbf{TJ}}_\mathsf{N}}$ w.r.t. `toNum`.

$\overline{\mathbf{TJ}}$ complete w.r.t. `toNum`, we have to derive the complete shell of the $\overline{\mathbf{TJ}}$ string abstract domain relative to the $\overline{\mathbf{TJ}}_\mathsf{N}$ numerical abstract domain, applying Theorem 5.1.

Definition 5.3 (Complete shell of $\overline{\mathbf{TJ}}$)**.** Let $\rho_{\overline{\mathbf{TJ}}} \in uco(\mathcal{P}(\Sigma^*))$, $\rho_{\overline{\mathbf{TJ}}_\mathsf{N}} \in uco(\mathcal{P}(\textsc{Int} \cup \textsc{Float}))$ be the upper closure operators related to $\overline{\mathbf{TJ}}$ and $\overline{\mathbf{TJ}}_\mathsf{N}$ abstract domains respectively. Given $[\![\mathtt{toNum}(\bullet)]\!]^{\overline{\mathbf{TJ}}} : \overline{\mathbf{TJ}} \to \overline{\mathbf{TJ}}_\mathsf{N}$, we define by $\overline{\mathbf{TJ}}^\star$ the transformer $\mathcal{S}^{\rho_{\overline{\mathbf{TJ}}_\mathsf{N}}}_{\mathtt{toNum}}(\rho_{\overline{\mathbf{TJ}}})$.

△

By Definition 5.1, $\overline{\mathbf{TJ}}^\star$ is the complete shell of $\rho_{\overline{\mathbf{TJ}}}$ relative to $\rho_{\overline{\mathbf{TJ}}_\mathsf{N}}$ with respect to the `toNum` operation. As already argued in Section 5.4, to compute $\overline{\mathbf{TJ}}^\star$ we use the constructive characterization given by Theorem 5.1.

Figure 5.5 depicts $\overline{\mathbf{TJ}}^\star$. In particular, the abstract points inside dashed boxes are the abstract values added during the computation of $\overline{\mathbf{TJ}}^\star$, the point inside the solid box is instead obtained by the Moore closure of the other points of the domain, while the remaining abstract values were already in $\overline{\mathbf{TJ}}$. A non-intuitive point added in $\overline{\mathbf{TJ}}^\star$ is SignedOrFloats, namely the abstract value s.t. its concretization contains any float string and the signed integers. This abstract point is added during the computation of $\overline{\mathbf{TJ}}^\star$. In particular, following Algorithm 6, instantiated with $\mathrm{A} = \mathcal{P}(\Sigma^*)$ and $\eta = \rho_{\overline{\mathbf{TJ}}_\mathsf{N}}$, this abstract point is computed at lines 2-4 at the iteration when y is $\gamma_{\overline{\mathbf{TJ}}_\mathsf{N}}(\mathsf{NotUnsignedInt})$, namely

$$\mathsf{SignedOrFloats} \in \mathsf{max}^{\subseteq}(\{\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid [\![\mathtt{toNum}(\mathrm{Z})]\!] \subseteq \gamma_{\overline{\mathbf{TJ}}_\mathsf{N}}(\mathsf{NotUnsignedInt})\})$$

Informally: which is the maximal set of strings Z such that `toNum`(Z) is dominated by NotUnsignedInt? To obtain from `toNum`(Z) only values dominated by NotUnsignedInt, the maximal set doing so is exactly the set of the float strings and the signed strings. Other strings, such that: unsigned integer strings or not numerical strings are excluded, since they are both converted to unsigned integers, and they would violate the dominance relation. Similarly, the abstract point UnsignedOrNotNumeric is added to the complete shell $\overline{\mathbf{TJ}}^\star$. Following again Algorithm 6, instantiated with $\mathrm{A} = \mathcal{P}(\Sigma^*)$ and $\eta = \rho_{\overline{\mathbf{TJ}}_\mathsf{N}}$, the above

	$[\![\texttt{concat}(\texttt{s}_1,\texttt{s}_2)]\!]^{\overline{\mathbf{SF}}}$				
$[\![\texttt{s}_1]\!]^{\overline{\mathbf{SF}}}$ \ $[\![\texttt{s}_2]\!]^{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$	$\sigma_2 \in \Sigma^*$	Numeric	NotNumeric	$\top_{\overline{\mathbf{SF}}}$
$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$
$\sigma_1 \in \Sigma^*$	$\bot_{\overline{\mathbf{SF}}}$	$\sigma_1 \cdot \sigma_2$	$\begin{cases} \text{Numeric} & \sigma_1 = \texttt{""} \text{ or } \sigma_1 \in \Sigma^*_{\mathsf{UInt}} \\ \text{NotNumeric} & \text{otherwise} \end{cases}$	NotNumeric	$\top_{\overline{\mathbf{SF}}}$
Numeric	$\bot_{\overline{\mathbf{SF}}}$	$\begin{cases} \text{Numeric} & \sigma_2 = \texttt{""} \text{ or } \sigma_2 \in \Sigma^*_{\mathsf{UInt}} \\ \text{NotNumeric} & \text{otherwise} \end{cases}$	$\top_{\overline{\mathbf{SF}}}$	NotNumeric	$\top_{\overline{\mathbf{SF}}}$
NotNumeric	$\bot_{\overline{\mathbf{SF}}}$	NotNumeric	$\top_{\overline{\mathbf{SF}}}$	NotNumeric	$\top_{\overline{\mathbf{SF}}}$
$\top_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$	$\top_{\overline{\mathbf{SF}}}$	$\top_{\overline{\mathbf{SF}}}$	$\top_{\overline{\mathbf{SF}}}$	$\top_{\overline{\mathbf{SF}}}$

FIGURE 5.6: $\overline{\mathbf{SF}}$ `concat` abstract semantics

.

abstract element is computed at lines 2-4 at the iteration when y is $\gamma_{\overline{\mathbf{TJ}}_{\mathsf{N}}}(\mathsf{UnsignedInt})$, namely

$$\mathsf{UnsignedOrNotNumeric} \in \max^{\subseteq}(\{\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid \texttt{toNum}(\mathrm{Z}) \subseteq \gamma_{\overline{\mathbf{TJ}}_{\mathsf{N}}}(\mathsf{UnsignedInt})\})$$

To obtain from $\texttt{toNum}(\mathrm{Z})$ only values dominated by UnsignedInt, the maximal set doing so is exactly the set of the unsigned integer strings and the non-numerical strings, since the latter are converted to 0.

Example 5.1. Consider again the string set $\mathrm{S} = \{\texttt{"2.3"}, \texttt{"3.4"}\}$ of Example 5.4.1. We can show that in $\overline{\mathbf{TJ}}^{\star}$, $\alpha_{\overline{\mathbf{TJ}}_{\mathsf{N}}}([\![\texttt{toNum}(\mathrm{S})]\!]) = [\![\texttt{toNum}(\alpha_{\overline{\mathbf{TJ}}^{\star}}(\mathrm{S}))]\!]^{\overline{\mathbf{TJ}}^{\star}}$. Indeed,

$$\begin{aligned} \alpha_{\overline{\mathbf{TJ}}_{\mathsf{N}}}([\![\texttt{toNum}(\mathrm{S})]\!]) &= \mathsf{NotUnsignedInt} \\ &= [\![\texttt{toNum}(\alpha_{\overline{\mathbf{TJ}}^{\star}}(\mathrm{S}))]\!]^{\overline{\mathbf{TJ}}^{\star}} \\ &= [\![\texttt{toNum}(\mathsf{SignedOrFloats})]\!]^{\overline{\mathbf{TJ}}^{\star}} \\ &= \mathsf{NotUnsignedInt} \end{aligned}$$

⬡

Note that, the new abstract semantics $[\![\texttt{toNum}(\bullet)]\!]^{\overline{\mathbf{TJ}}^{\star}}$, handling the abstract points added by the complete shell, corresponds to the best correction approximation, namely $\alpha_{\overline{\mathbf{TJ}}^{\star}} \circ [\![\texttt{toNum}(\bullet)]\!] \circ \gamma_{\overline{\mathbf{TJ}}^{\star}} : \overline{\mathbf{TJ}}^{\star} \to \overline{\mathbf{TJ}}_{\mathsf{N}}$ (cf. Definition 2.4).

5.4.2 Completing SAFE String Abstract Domain

Figure 5.4a depicts the string abstract domain $\overline{\mathbf{SF}}$, i.e., the recasted version of the domain involved into SAFE [122] static analyser. It splits strings into the abstract values: Numeric (i.e., numerical strings) and NotNumeric (i.e., all the other strings). As for $\overline{\mathbf{TJ}}$, before

reaching these abstract values, $\overline{\mathbf{SF}}$ precisely tracks single string values. For instance, $\alpha_{\overline{\mathbf{SF}}}(\{\texttt{"+9.6"}, \texttt{"7"}\}) = \mathsf{Numeric}$, and $\alpha_{\overline{\mathbf{SF}}}(\{\texttt{"+9.6"}, \texttt{"bar"}\}) = \top_{\overline{\mathbf{SF}}}$.

We study the completeness of $\overline{\mathbf{SF}}$ w.r.t. `concat` operation. Figure 5.6 presents the abstract semantics of the concatenation operation for $\overline{\mathbf{SF}}$, that is:

$$[\![\texttt{concat}(\bullet, \bullet)]\!]^{\overline{\mathbf{SF}}} : \overline{\mathbf{SF}} \times \overline{\mathbf{SF}} \to \overline{\mathbf{SF}}$$

In particular, when both abstract values correspond to single strings, the standard string concatenation is applied (second row, second column). In the case in which one abstract value, involved in the concatenation, is a string and the other is Numeric (third row, second column and second row, third column) we distinguish two cases: if the string is empty or corresponds to an unsigned integer we can safely return Numeric, otherwise NotNumeric is returned. This happens because, when two float strings (hence numerical strings) are concatenated, a non-numerical string is returned (e.g., $\texttt{concat}(\texttt{"1.1"}, \texttt{"2.2"}) = \texttt{"1.12.2"}$). For the same reason, when both input abstract values are Numeric, the result is not guaranteed to be numerical. Indeed, $[\![\texttt{concat}(\mathsf{Numeric}, \mathsf{Nu\text{-}meric})]\!]^{\overline{\mathbf{SF}}} = \top_{\overline{\mathbf{SF}}}$.

Lemma 5.2. The $\overline{\mathbf{SF}}$ abstract domain is not complete with respect to the `concat` operation. In particular, $\exists \mathrm{S}_1, \mathrm{S}_2 \in \mathcal{P}(\Sigma^*)$ such that:

$$\alpha_{\overline{\mathbf{SF}}}([\![\texttt{concat}(\mathrm{S}_1, \mathrm{S}_2)]\!]) \subsetneq [\![\texttt{concat}(\alpha_{\overline{\mathbf{SF}}}(\mathrm{S}_1), \alpha_{\overline{\mathbf{SF}}}(\mathrm{S}_2))]\!]^{\overline{\mathbf{SF}}}$$

Proof.
As a counterexample to completeness, consider the sets $\mathrm{S}_1 = \{\texttt{"2.2"}, \texttt{"2.3"}\}$ and $\mathrm{S}_2 = \{\texttt{"2"}, \texttt{"3"}\}$. Then, in $\overline{\mathbf{SF}}$, $\alpha_{\overline{\mathbf{SF}}}([\![\texttt{concat}(\mathrm{S}_1, \mathrm{S}_2)]\!]) \neq [\![\texttt{concat}(\alpha_{\overline{\mathbf{SF}}}(\mathrm{S}_1), \alpha_{\overline{\mathbf{SF}}}(\mathrm{S}_2))]\!]^{\overline{\mathbf{SF}}}$. Indeed,

$$\begin{aligned}
\alpha_{\overline{\mathbf{SF}}}([\![\texttt{concat}(\mathrm{S}_1, \mathrm{S}_2)]\!]) &= \mathsf{Numeric} \\
&\neq [\![\texttt{concat}(\alpha_{\overline{\mathbf{SF}}}(\mathrm{S}_1), \alpha_{\overline{\mathbf{SF}}}(\mathrm{S}_2))]\!]^{\overline{\mathbf{SF}}} \\
&= [\![\texttt{concat}(\mathsf{Numeric}, \mathsf{Numeric})]\!]^{\overline{\mathbf{SF}}} \\
&= \top_{\overline{\mathbf{SF}}}
\end{aligned}$$

□

The $\overline{\mathbf{SF}}$ abstract domain loses too much information during the abstraction process; information that can not be retrieved during the abstract concatenation. Intuitively, to gain completeness w.r.t. `concat` operation, $\overline{\mathbf{SF}}$ should improve the precision of the numerical strings abstraction, e.g., discriminating between float and integer strings.

Definition 5.4 (Complete shell of $\overline{\mathbf{SF}}$)**.** Let $\rho_{\overline{\mathbf{SF}}} \in uco(\mathcal{P}(\Sigma^*))$ be the upper closure operator related to $\overline{\mathbf{SF}}$ abstract domain. Given $[\![\texttt{concat}(\bullet, \bullet)]\!]^{\overline{\mathbf{SF}}} : \overline{\mathbf{SF}} \times \overline{\mathbf{SF}} \to \overline{\mathbf{SF}}$, we define by $\overline{\mathbf{SF}}^{\star}$ the transformer $\overline{\mathcal{S}}_{\texttt{concat}}(\rho_{\overline{\mathbf{SF}}})$.

△

X \ Y	Numeric	NotNumeric	"n" $\in \mathbb{Z}$	"f" $\in \mathbb{F}$	"s" $\in$ NotNum
Numeric	{""} $\cup$ UInt	[NotNum $\setminus$ {""}] $\cup$ NotUInt $\cup$ NotUFloat	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$
NotNumeric	$\bot_{\overline{\mathbf{SF}}}$	NotNumeric	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$
"n" $\in \mathbb{Z}$	{""} $\cup$ UInt $\cup$ UFloat	[NotNum $\setminus$ {""}] $\cup$ NotUInt $\cup$ NotUFloat	$\bot_{\overline{\mathbf{SF}}} \vee$ str	$\bot_{\overline{\mathbf{SF}}} \vee$ str	$\bot_{\overline{\mathbf{SF}}} \vee$ str
"f" $\in \mathbb{F}$	{""} $\cup$ UInt	[NotNum $\setminus$ {""}] $\cup$ Float $\cup$ NotUInt	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}} \vee$ str	$\bot_{\overline{\mathbf{SF}}} \vee$ str
"s" $\in$ NotNum	$\bot_{\overline{\mathbf{SF}}}$	NotNumeric	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}}$	$\bot_{\overline{\mathbf{SF}}} \vee$ str

TABLE 5.1: First iteration of $\mathsf{max}^{\subseteq}(\{Z \in \mathcal{P}(\Sigma^*) \mid [\![\mathtt{concat}(X,Z)]\!] \subseteq \gamma_{\overline{\mathbf{SF}}}(Y)\})$

By Definition 5.2, $\overline{\mathbf{SF}}^{\star}$ is the absolute complete shell of $\rho_{\overline{\mathbf{SF}}}$ with respect to the `concat` operation. By Theorem 5.2, the transformer $\overline{\mathcal{S}}_{\mathtt{concat}}(\rho_{\overline{\mathbf{SF}}})$ is equal to the Moore closure of the union between $\overline{\mathbf{SF}}$ and the binary operator defined in Table 5.1. Table 5.1 depicts the first iteration of the fix-point computation of Theorem 5.2, where $\bot_{\overline{\mathbf{SF}}}$ and $\top_{\overline{\mathbf{SF}}}$ rows and columns are omitted for space limitations. In particular, $\forall X \in \mathcal{P}(\Sigma^*)$, when $X \neq \bot_{\overline{\mathbf{SF}}}$, we have that:

- $\mathsf{max}(\{Z \in \mathcal{P}(\Sigma^*) \mid [\![\mathtt{concat}(Z,X)]\!] \subseteq \gamma_{\overline{\mathbf{SF}}}(\bot_{\overline{\mathbf{SF}}})\}) = \bot_{\overline{\mathbf{SF}}}$
- $\mathsf{max}(\{Z \in \mathcal{P}(\Sigma^*) \mid [\![\mathtt{concat}(X,Z)]\!] \subseteq \gamma_{\overline{\mathbf{SF}}}(\bot_{\overline{\mathbf{SF}}})\}) = \bot_{\overline{\mathbf{SF}}}$
- $\mathsf{max}(\{Z \in \mathcal{P}(\Sigma^*) \mid [\![\mathtt{concat}(X,Z)]\!] \subseteq \gamma_{\overline{\mathbf{SF}}}(\bot_{\overline{\mathbf{SF}}})\}) = \top_{\overline{\mathbf{SF}}}$

On the other hand, $\forall X \in \mathcal{P}(\Sigma^*)$, when $X \neq \top_{\overline{\mathbf{SF}}}$, we have that:

$$\mathsf{max}(\{Z \in \mathcal{P}(\Sigma^*) \mid [\![\mathtt{concat}(Z,X)]\!] \subseteq \gamma_{\overline{\mathbf{SF}}}(\bot_{\overline{\mathbf{SF}}})\}) = \bot_{\overline{\mathbf{SF}}}$$

The complete shell $\overline{\mathbf{SF}}^{\star}$ is depicted in Figure 5.7.[5] In particular, the points inside dashed boxes are the abstract values added during the iterative computations of $\overline{\mathbf{SF}}^{\star}$, the points inside solid boxes are instead obtained by the Moore closure of the other points of the domain, while the remaining abstract values were already in $\overline{\mathbf{SF}}$. The meaning of abstract values in $\overline{\mathbf{SF}}^{\star}$ is intuitive. In order to satisfy the completeness property, $\overline{\mathbf{SF}}^{\star}$ splits the Numeric abstract value, already taken into account in $\overline{\mathbf{SF}}$, into all the strings corresponding to unsigned integer (UInt), unsigned floats (UFloat), and signed numbers

[5]The empty strings (`""`) at the most-left and most-right sides of Figure 5.7 are not distinct elements but they are only duplicated in order to declutter the figure.

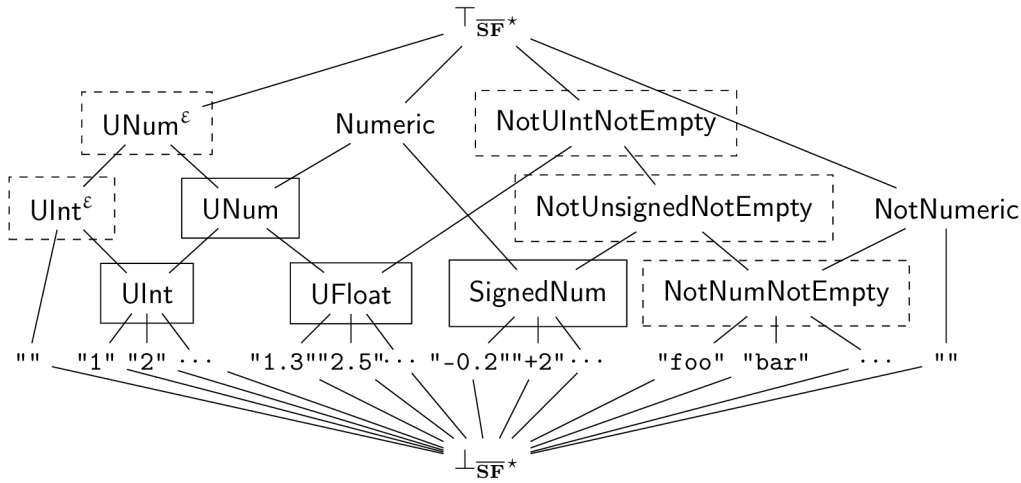


FIGURE 5.7: Absolute complete shell of $\rho_{\overline{\mathbf{SF}}}$ w.r.t. `concat`.

(SignedNum). Moreover, particular importance is given to the empty string, since the new abstract domain specifies whether each abstract value contains `""`. Indeed, the UInt$^\varepsilon$ abstract value represents the strings corresponding to unsigned integer or to the empty string, and the UNum$^\varepsilon$ abstract value represents the strings corresponding to unsigned numbers or to the empty string. An unexpected abstract value considered in $\overline{\mathbf{SF}}^\star$ is NotUnsignedNotEmpty, such that:

$$\gamma_{\overline{\mathbf{SF}}^\star}(\mathsf{NotUnsignedNotEmpty}) = \{\sigma \in \Sigma^* \mid \sigma \in \Sigma^*_{\mathsf{SInt}} \cup \Sigma^*_{\mathsf{SFloat}} \cup (\Sigma^*_{\mathsf{NotNum}} \setminus \{\texttt{""}\})\}$$

Namely, the abstract point whose concretization corresponds to the set of any non-numerical string, except the empty string, and any string corresponding to a signed number. This abstract point has been added to $\overline{\mathbf{SF}}^\star$, following Theorem 5.2. Since the absolute complete shell is the greatest fix-point of the relative one (that in our case is reached after one iteration), the corresponding procedure is the greatest fix-point of Algorithm 6, instantiated with $\mathrm{A} = \mathcal{P}(\Sigma^*)$ and $\eta = \rho_{\overline{\mathbf{SF}}}$. In particular, the abstract element is computed at lines 2-4 at the iteration when y is $\gamma_{\overline{\mathbf{SF}}}(\mathsf{NotNumeric})$, x is $\gamma_{\overline{\mathbf{SF}}}(\mathsf{Numeric})$ and $\mathrm{i} = 1$, namely NotUnsignedNotEmpty belongs to

$$\mathsf{max}^{\subseteq}(\mathrm{Z} \in \mathcal{P}(\Sigma^*) \mid [\![\texttt{concat}(\gamma_{\overline{\mathbf{SF}}}(\mathsf{Numeric}), \mathrm{Z})]\!] \subseteq \gamma_{\overline{\mathbf{SF}}}(\mathsf{NotNumeric}))$$

Informally: which is the maximal set of strings s.t. concatenated to any possible numerical string will produce any possible non-numerical string? Indeed, in order to be sure to obtain non-numerical strings, the maximal set doing so is exactly the set of any non-numerical non-empty string, and any string corresponding to a signed number, that is NotUnsignedNotEmpty.

Example 5.2. Let $\mathrm{S}_1 = \{\texttt{"2.2"}, \texttt{"2.3"}\}$ and $\mathrm{S}_2 = \{\texttt{"2"}, \texttt{"3"}\}$ be the string sets of Example 5.4.2. We can show that in $\overline{\mathbf{SF}}^\star$, $\alpha_{\overline{\mathbf{SF}}^\star}([\![\texttt{concat}(\mathrm{S}_1, \mathrm{S}_2)]\!]) = [\![\texttt{concat}(\alpha_{\overline{\mathbf{SF}}^\star}(\mathrm{S}_1), \alpha_{\overline{\mathbf{SF}}^\star}(\mathrm{S}_2))]\!]^{\overline{\mathbf{SF}}^\star}$.

Indeed,

$$\begin{aligned}\alpha_{\overline{\mathbf{SF}}^\star}([\![\texttt{concat}(\mathrm{S}_1,\mathrm{S}_2)]\!]) &= \mathsf{UFloat}\\ &= [\![\texttt{concat}(\alpha_{\overline{\mathbf{SF}}^\star}(\mathrm{S}_1),\alpha_{\overline{\mathbf{SF}}^\star}(\mathrm{S}_2))]\!]^{\overline{\mathbf{SF}}^\star}\\ &= [\![\texttt{concat}(\mathsf{UFloat},\mathsf{UInt})]\!]^{\overline{\mathbf{SF}}^\star}\\ &= \mathsf{UFloat}\end{aligned}$$

⬡

As in the $\overline{\mathbf{TJ}}^\star$ case, the new abstract semantics $[\![\texttt{concat}(\bullet,\bullet)]\!]^{\overline{\mathbf{SF}}^\star}$, handling the abstract points added by the complete shell, corresponds the best correction approximation, namely $\alpha_{\overline{\mathbf{SF}}^\star} \circ [\![\texttt{concat}(\bullet,\bullet)]\!] \circ \gamma_{\overline{\mathbf{SF}}^\star} : \overline{\mathbf{SF}}^\star \times \overline{\mathbf{SF}}^\star \to \overline{\mathbf{SF}}^\star$ (see Definition 2.4).

5.5 Benefits of Adopting Complete String Abstractions

Now, we discuss and evaluate the benefits gained from using the complete shells reported in Section 5.4 and, more in general, complete domains, w.r.t. a certain operation. In particular, we compare the μDyn versions of the string abstract domains adopted by SAFE and TAJS with their corresponding complete shells, we discuss the complexity of the complete shells, and finally we argue how adopting complete abstract domains can be useful into static analysers.

5.5.1 Precision

In the previous section, we focused on the completeness of the string abstract domains integrated into SAFE and TAJS, for μDyn, w.r.t. two string operations, namely `concat` and `toNum`, respectively. While string concatenation is common in any programming language, `toNum` assumes critical importance for dynamic languages, where implicit type conversion is provided. Since type conversion is often hidden from the developer, aim to completeness of the analysis increases the precision of such operations. For instance, let `x` be a variable, at a certain program execution point. `x` may have concrete value in the set $\mathrm{S} = \{\texttt{"foo"},\texttt{"bar"}\}$. If S is abstracted into the original TAJS string abstract domain, its abstraction will corresponds to NotUnsigned, losing the information about the fact that the concrete value of `x` surely does not contain numerical values. Hence, when the abstract value of S is used as input of `toNum`, the result will return $\top_{\overline{\mathbf{TJ}}_\mathsf{N}}$, i.e., any possible concrete integer value. Conversely, abstracting S in $\overline{\mathbf{TJ}}^\star$ (the absolute complete shell of $\overline{\mathbf{TJ}}$ relative to `toNum` discussed in Section 5.4.1) leads to a more precise abstraction, since $\overline{\mathbf{TJ}}^\star$ is able to differentiate between non-numerical and numerical strings. In particular, the abstract value of S in $\overline{\mathbf{TJ}}^\star$ is NotNumeric, and $[\![\texttt{toNum}(\mathsf{NotNumeric})]\!]^{\overline{\mathbf{TJ}}^\star}$ will precisely return 0.

Adopting a complete shell w.r.t. a certain operation does not compromise the precision of the others. For example, consider again the domain of TAJS and the following `JavaScript` fragment.

```
1 var obj = {
2     "foo" : 1,
3     "bar" : 2,
4     "1.2" : 3,
5     "2.2" : "hello"
6 }
7
8 y = obj[idx];
```

Suppose that the value of `idx` is the abstraction, in the starting TAJS string abstract domain, of the string set $S = \{$`"foo"`, `"bar"`$\}$, namely the abstract value NotUnsigned. The variable `idx` is used to access the property of the object `obj` at line 8 and, to guarantee soundness, it accesses *all* the properties of `obj`, including the fields `"1.2"` and `"2.2"`, introducing noise in the abstract computation, since `"1.2"` and `"2.2"` are false positives values introduced by the abstraction of the values of `idx`. If we analyse the same `JavaScript` fragment with the absolute complete shell (w.r.t. `toNum` operation) of the TAJS string abstract domain defined in Section 5.4.1, we obtain more precise results. Indeed, in this case, the value of `idx` corresponds to the the abstract value NotNumeric, and when it is used to access the object `obj` at line 8, only `"foo"` and `"bar"` are accessed, since they are the only non-numerical string properties of `obj`.

5.5.2 Qualitative Evaluation of Complete Shells

We evaluate, from a qualitative point of view, the complete shells we have provided in the previous section. As usual in static analysis by Abstract Interpretation, there exists a trade-off between precision and efficiency: choose a more precise abstract domain may compromise the efficiency of the abstract computations. A representative example is reported in [80]: the complete shell of the sign abstract domain w.r.t. addition is the interval abstract domain. Hence, starting from a finite height abstract domain (signs) we obtain an infinite height abstract domain (intervals). In particular, fix-point computations on signs converge, while on intervals may diverge. Indeed, after the completion, the interval abstract domain should be equipped also with a widening [51] in order to still guarantee termination. A worst-case scenario is when the complete shells w.r.t. a certain operation exactly corresponds to the collecting abstract domain, i.e., the concrete domain. Clearly, we cannot use the concrete domain due to undecidability reasons, but this suggest us to change the starting abstract domain, since it is not able to track any information related to the operation of interest. An example is the suffix abstract domain [49] with `substring` operation: since this abstract domain tracks only the common suffix of a strings set, it can

not track the information about the indexes of the common suffix, and the complete shell of the suffix abstract domain w.r.t. `substring` would lead to the concrete domain. Hence, if the focus of the abstract interpreter is to improve the precision of the `substring` operation, we should change the abstract domain with a more precise one for `substring`, such as the finite state automata [16] abstract domain.

Consider now the complete shells reported in Section 5.4. The obtained complete shells still have finite height, hence termination is still guaranteed without the need to equip the complete shells with widening operators. Moreover, the complexity of the string operations of interest is preserved after completion. Indeed, in both TAJS and SAFE starting abstract domains, `concat` and `toNum` operations have constant complexity, respectively, and the same complexity is preserved in the corresponding complete shells. It is worth noting that also the complexity of the abstract domain-related operations, such as least upper bound, greatest lower bound and the ordering operator, is preserved in the complete shells. Hence, to conclude, as far as the complete shells we have reported for TAJS and SAFE are concerned, there is no worsening when we substitute the original string abstract domains with the corresponding complete shells, and this leads, as we have already mentioned before, to completeness during the input abstraction process w.r.t. the relative operations, namely `concat` for SAFE and `toNum` for TAJS.

5.5.3 False Positives Reduction

In static analysis, a certain degree of abstraction must be added in order to obtain decidable procedures to infer invariants on a generic program. Clearly, using less precise abstract domains lead to an increase of *false positive* values of the computed invariants. In particular, after a program is analysed, this burdens the phase of false positive detection: when a program is analysed, the following phase consists in detecting which values of the invariants derived by the static analyser are spurious, namely those values that certainly are not computed by the concrete execution of the program of interest.

In particular, using imprecise (i.e., not complete) abstract domains clearly increase the number of false positives in the abstract computation of the static analyser, burdening the next phase of detection of the spurious values. Conversely, adopting (backward) complete abstract domains w.r.t. a certain operation reduce the numbers of false positives introduced during the abstract computations, at least in the input abstraction process. Clearly, in this way, the next phase of detection of false positives will be lighter since less noise has been introduced during the abstract computation of the invariants. Consider again the `JavaScript` fragment reported in the previous paragraph. As we already discussed before, using the starting TAJS abstract domain to abstract the variable `idx` leads to a loss of precision, since the spurious value `"foo"` and `"bar"` are taken into account in its abstract value, namely Unsigned. Using the complete shell of TAJS w.r.t. `toNum` instead does not add noise when `idx` is used to access `obj`.

5.6 Relative Precision

In this section, we recall the notion of pseudo-distance on abstract domains, firstly defined in [125], which takes into account the order relation between abstract elements (together with the fact that different abstract elements might approximate the same set of concrete values) and their possible incomparability. Then we formalise the increment of the precision obtained when analysing a program with a complete abstract domain with respect to original version of the domain.

5.6.1 Abstract Domains Precision: an Overview

It is well known that abstract domains precision can be qualitatively compared by exploiting the information they are able to capture [44, 45]. On the other hand, quantitatively evaluating the precision of abstract domains has been proven to be quite challenging and frequently resulted in ad hoc measures.

Di Pierro and Wiklicky [144], introduced the notion of probabilistic Abstract Interpretation, used to numerically estimate the incompleteness of numerical abstract domains.

Sotin [160] presented the notion of precision of a numerical abstract value, measuring the volume it describes, with the purpose of quantitatively comparing the precision of numerical abstract domains. Finally, Casso et al. [32] wanted to compare the precision of different analyses on logic programs. Thus, they proposed distances in two abstract domains used in constraint logic programming and they extended them to distances between the results of different analyses of a given program.

Logozzo et al. [125] gave a more general definition of pseudo-distance between abstract domain elements that allows to quantify the error of approximating a concrete element in an abstract domain.

Definition 5.5 (Pseudo-distance [125])**.** Let $\langle \overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}}, \bot_{\overline{\mathbf{D}}}, \top_{\overline{\mathbf{D}}}, \sqcap_{\overline{\mathbf{D}}}, \sqcup_{\overline{\mathbf{D}}} \rangle$ be an abstract domain. A function $\delta : \overline{\mathbf{D}} \times \overline{\mathbf{D}} \to \mathbb{R} \cup \{+\infty\}$ is a *pseudo-distance* over the abstract domain $\overline{\mathbf{D}}$ if and only if, for any $\overline{\mathbf{d}}, \overline{\mathbf{d}}', \overline{\mathbf{d}}'' \in \overline{\mathbf{D}}$, it satisfies the following axioms:

- *non-negativity*: $\delta(\overline{\mathbf{d}}, \overline{\mathbf{d}}') \geq 0$
- *if-identity*: $\overline{\mathbf{d}} =_{\overline{\mathbf{D}}} \overline{\mathbf{d}}' \Rightarrow \delta(\overline{\mathbf{d}}, \overline{\mathbf{d}}') = 0$
- *symmetry*: $\delta(\overline{\mathbf{d}}, \overline{\mathbf{d}}') = \delta(\overline{\mathbf{d}}', \overline{\mathbf{d}})$
- *weak triangle inequality*: $\overline{\mathbf{d}} \sqsubseteq_{\overline{\mathbf{D}}} \overline{\mathbf{d}}'' \sqsubseteq_{\overline{\mathbf{D}}} \overline{\mathbf{d}}' \Rightarrow \delta(\overline{\mathbf{d}}, \overline{\mathbf{d}}'') \leq \delta(\overline{\mathbf{d}}, \overline{\mathbf{d}}') + \delta(\overline{\mathbf{d}}', \overline{\mathbf{d}}'')$

$\triangle$

We recall the path length distance definition.

```
1 nums = "";
2 notnums = "";
3 i = 0;
4 while (i < length(s)) {
    5 if (toNum(charAt(str, i)) == 0) {
        6 notnums = concat(notnums, charAt(s, i));7
    } else {
        8 nums = concat(nums, charAt(s, i));9
    }
10 i = i + 1;11
}12
```

FIGURE 5.8: Example of μDyn annotated program

Definition 5.6 (Path length distance [125])**.** Let $\langle \overline{\mathbf{D}}, \sqsubseteq_{\overline{\mathbf{D}}}, \bot_{\overline{\mathbf{D}}}, \top_{\overline{\mathbf{D}}}, \sqcap_{\overline{\mathbf{D}}}, \sqcup_{\overline{\mathbf{D}}} \rangle$ be an abstract domain. The path length distance $\delta_{\mathsf{plen}} : \overline{\mathbf{D}} \times \overline{\mathbf{D}} \to \mathbb{R} \cup \{+\infty\}$ is defined as

$$\delta_{\mathsf{plen}}(\overline{\mathbf{d}}, \overline{\mathbf{d}}') = \begin{cases} \mathsf{plen}(\overline{\mathbf{d}}, \overline{\mathbf{d}}') & \overline{\mathbf{d}} \sqsubseteq_{\overline{\mathbf{D}}} \overline{\mathbf{d}}' \vee \overline{\mathbf{d}}' \sqsubseteq_{\overline{\mathbf{D}}} \overline{\mathbf{d}} \\ +\infty & \text{otherwise} \end{cases}$$

where $\mathsf{plen}(\overline{\mathbf{d}}, \overline{\mathbf{d}}')$ computes the distance between two abstract values $\overline{\mathbf{d}}, \overline{\mathbf{d}}' \in \overline{\mathbf{D}}$ and it is defined as follows.

$$\mathsf{plen}(\overline{\mathbf{d}}, \overline{\mathbf{d}}') = \min\{n \mid \{\overline{\mathbf{d}}_0, \overline{\mathbf{d}}_1, \ldots, \overline{\mathbf{d}}_n\} \in \mathcal{P}(\overline{\mathbf{D}}), \overline{\mathbf{d}}_0 = \overline{\mathbf{d}}, \overline{\mathbf{d}}_n = \overline{\mathbf{d}}', \\ \forall i \in [0, n). \overline{\mathbf{d}}_i \sqsubseteq_{\overline{\mathbf{D}}} \overline{\mathbf{d}}_{i+1}\}$$

△

5.6.2 Measuring Precision Gained by Complete Shells

We aim at computing the distance, and in turn the increment of precision, between abstract points of an abstract domain $\overline{\mathbf{D}}$ and its complete shell, w.r.t. a certain operation of interest f, noting that the abstract values of $\overline{\mathbf{D}}$ are all contained in its complete shell. In order not to clutter the notation, given an abstract domain $\overline{\mathbf{D}}$, we denote by $\mathsf{Shell}_f(\overline{\mathbf{D}})$ its complete shell w.r.t the function f.

Let P be a μDyn program and let denote by $\mathsf{Lab}(\mathsf{P})$ the program points of P. We denote by $\ell_i \in \mathsf{Lab}(\mathsf{P})$ its i-th program point and by $\mathsf{Vars}(\mathsf{P})$ the program variables. An example of μDyn program annotated with its program points is reported in Figure 5.8. Given such a program, as usual in static program analysis, the goal is to compute the abstract values associated to each variable $x \in \mathsf{Vars}(\mathsf{P})$ at each program point $\ell \in \mathsf{Lab}(\mathsf{P})$.

We denote by $\xi^{\overline{\mathbf{D}}} : \mathsf{Vars}(\mathsf{P}) \to \overline{\mathbf{D}}$ the abstract state associating each variable to an abstract value of $\overline{\mathbf{D}}$. When it is clear from the context, we denote the abstract state by ξ. Hence, given an abstract domain $\overline{\mathbf{D}}$ and a program P, the result of the analysis of P using

the abstract domain $\overline{\mathbf{D}}$ (and the corresponding abstract semantics $[\![\cdot]\!]^{\overline{\mathbf{D}}}$) is defined by

$$\mathsf{Analysis}(\mathsf{P}, \overline{\mathbf{D}}, \xi_\emptyset) = \{(\ell_i, \xi_i) \mid \ell_i \in \mathsf{Lab}(\mathsf{P}), \xi_i = [\![\mathsf{P}]\!]^{\overline{\mathbf{D}}}\xi_\emptyset \text{ at program point } \ell_i\}$$

where $\xi_\emptyset$ is the (initial) empty abstract state. For example, let us analyse the μDyn program in Figure 5.8 with the TAJS abstract domain reported in Figure 5.4b. The abstract state holding at program point 12 (namely the exit program point) is $\xi_{12} = \{\mathtt{i} \mapsto \mathsf{UnsignedInt}, \mathtt{nums} \mapsto \top_{\overline{\mathbf{TJ}}}, \mathtt{notnums} \mapsto \top_{\overline{\mathbf{TJ}}}\}$, indeed the pair $(12, \xi_{12}) \in \mathsf{Analysis}(\mathsf{P}, \overline{\mathbf{TJ}}, \xi_\emptyset)$.

We can exploit the definition of path length distance reported in Definition 5.6 to define, for each program point $\ell \in \mathsf{Lab}(\mathsf{P})$, the distance from $\bot_{\overline{\mathbf{D}}}$ and each abstract value associated with each $\mathtt{x} \in \mathsf{Vars}(\mathsf{P})$.

Definition 5.7 (Set of ⊥-distances)**.** Let P be a μDyn program, $\overline{\mathbf{D}}$ an abstract domain and δ_{plen} the path length distance defined in Definition 5.6. The set of $\bot_{\overline{\mathbf{D}}}$-distances for a given $\ell \in \mathsf{Lab}(\mathsf{P})$ is defined as follows.

$$\Phi_\ell(\mathsf{P}, \overline{\mathbf{D}}) = \{(\mathtt{x}, \delta(\bot_{\overline{\mathbf{D}}}, \xi(\mathtt{x}))) \mid (\ell, \xi) \in \mathsf{Analysis}(\mathsf{P}, \overline{\mathbf{D}}, \xi_\emptyset), \mathtt{x} \in \mathsf{Vars}(\mathsf{P})\}$$

△

For example, let us consider again ξ_{12}, previously defined, that is the abstract state holding at the program point 12 of the program reported in Figure 5.8 analyzing it with $\overline{\mathbf{TJ}}$ abstract domain. Hence, the set of the $\bot_{\overline{\mathbf{TJ}}}$-distances at program point 12 is $\Phi_{12}(\mathsf{P}, \overline{\mathbf{TJ}}) = \{(\mathtt{i}, 2), (\mathtt{nums}, 3), (\mathtt{notnums}, 3)\}$.

Note that, by the weak triangle inequality, given two abstract elements $\overline{\mathbf{d}}, \overline{\mathbf{d}}' \in \overline{\mathbf{D}}$, if $\overline{\mathbf{d}} \sqsubseteq_{\overline{\mathbf{D}}} \overline{\mathbf{d}}' \Rightarrow \delta(\bot_{\overline{\mathbf{D}}}, \overline{\mathbf{d}}) \leq \delta(\bot_{\overline{\mathbf{D}}}, \overline{\mathbf{d}}')$. Moreover, since $\bot_{\overline{\mathbf{D}}}$ is comparable with any abstract value $\overline{\mathbf{d}} \in \overline{\mathbf{D}}$, $\delta(\bot_{\overline{\mathbf{D}}}, \overline{\mathbf{d}}) \neq \infty$.

Definition 5.8 (Precision entropy at a program point)**.** Let P be a μDyn program and $\overline{\mathbf{D}}$ an abstract domain. Given a program point $\ell \in \mathsf{Lab}(\mathsf{P})$, the precision entropy of $\overline{\mathbf{D}}$ for the program P at ℓ is defined as

$$\mathbb{P}_\ell(\mathsf{P}, \overline{\mathbf{D}}) = \sum_{(\mathtt{x}_i, \mathrm{d}_i) \in \Phi_\ell(\mathsf{P}, \overline{\mathbf{D}})} \mathrm{d}_i$$

where d_i is the path length distance from $\bot_{\overline{\mathbf{D}}}$ to the abstract value of $\mathtt{x}_i$ at the program point ℓ.

△

We can use Definition 5.8 to define the precision entropy of an analysis for a given program.

Definition 5.9 (Precision entropy)**.** Let P be a μDyn program and $\overline{\mathbf{D}}$ be an abstract domain. The precision entropy $\mathbb{P}(\mathsf{P}, \overline{\mathbf{D}})$ of the abstract domain A for the program P is

defined as

$$\mathbb{P}(\mathsf{P}, \overline{\mathbf{D}}) = \sum_{\ell \in \mathsf{Lab}(\mathsf{P})} \mathbb{P}_\ell(\mathsf{P}, \overline{\mathbf{D}})$$

$\triangle$

The precision entropy $\mathbb{P}(\mathsf{P}, \overline{\mathbf{D}})$ says how much information the analysis based on the abstract domain $\overline{\mathbf{D}}$ is able to express: the more $\mathbb{P}(\mathsf{P}, \overline{\mathbf{D}})$ is low, the more the analysis based on $\overline{\mathbf{D}}$ is precise for the program P. Using the so far defined metric, we are able to compare the analysis results based on a certain abstract domain and its corresponding complete shell. As we have already mentioned at the beginning of Section 5.6, any abstract point of $\overline{\mathbf{D}}$ is contained in its complete shell. Moreover, it is worth noting that, given a variable $\mathtt{x} \in \mathsf{Vars}(\mathsf{P})$ and a certain program point $\ell \in \mathsf{Lab}(\mathsf{P})$, the abstract value computed by the analysis on $\overline{\mathbf{D}}$ associated to $\mathtt{x}$ is always comparable with the abstract value computed by the analysis on $\mathsf{Shell}_\mathrm{f}(\overline{\mathbf{D}})$ associated to $\mathtt{x}$. Formally, let consider $(\ell, \xi) \in \mathsf{Analysis}(\mathsf{P}, \overline{\mathbf{D}}, \xi_\emptyset)$ and $(\ell, \xi') \in \mathsf{Analysis}(\mathsf{P}, \mathsf{Shell}_\mathrm{f}(\overline{\mathbf{D}}), \xi_\emptyset)$, for some program point $\ell \in \mathsf{Lab}(\mathsf{P})$, we have that $\forall \mathtt{x} \in \mathsf{Vars}(\mathsf{P}).\ \xi'(\mathtt{x}) \sqsubseteq_{\mathsf{Shell}_\mathrm{f}(\overline{\mathbf{D}})} \xi(\mathtt{x})$. This can be expressed by the following predicate

$$\mathbb{P}(\mathsf{P}, \mathsf{Shell}_\mathrm{f}(\overline{\mathbf{D}})) \leq \mathbb{P}(\mathsf{P}, \overline{\mathbf{D}}).$$

Informally speaking, for a program point ℓ, the analysis result of $\mathtt{x}$ on $\mathsf{Shell}_\mathrm{f}(\overline{\mathbf{D}})$ is always dominated (i.e., less or equals) by the analysis result of $\mathtt{x}$ on $\overline{\mathbf{D}}$ (contained, as we have mentioned before, to $\mathsf{Shell}_\mathrm{f}(\overline{\mathbf{D}})$). This fact directly comes from the dominance relation involved in the definition of complete shells given in Theorems 5.1 and 5.2. For this reason, we can always compare the analysis results produced by $\overline{\mathbf{D}}$ and $\mathsf{Shell}_\mathrm{f}(\overline{\mathbf{D}})$ for any variable and any program point.

This leads us to an automatic procedure to compute how much the analysis performed by $\mathsf{Shell}_\mathrm{f}(\overline{\mathbf{D}})$ is better than the one performed by $\overline{\mathbf{D}}$.

5.6.3 Experimental Evaluation

The Clam static analyzer for μDyn programs[6] implements the TAJS and SAFE string abstract domains and their corresponding complete shells discussed in Section 5.4. The abstract interpreter is parametric, as it can be tuned by selecting the string abstract domain to be used to analyse a given program. Other string abstract domains can be easily plugged into our static analyser, without re-implementing the underlying abstract interpreter. Moreover, it is possible to check the precision entropy, discussed in Section 5.6, of an abstract domain and its complete shell. In this way, it is possible to check at which program point and for which variables the complete shell-based analysis gains precision w.r.t. the analysis on the original abstract domain.

[6] Available at `https://github.com/VincenzoArceri/clam`

Variable	$\overline{\mathbf{TJ}}$	$\overline{\mathbf{TJ}}^{\star}$	$\mathbb{P}_{12}(\mathsf{P}, \overline{\mathbf{TJ}}) - \mathbb{P}_{12}(\mathsf{P}, \overline{\mathbf{TJ}}^{\star})$
`s`	UnsignedStr	UnsignedStr	0
`i`	UnsignedInt	UnsignedInt	0
`nums`	$\top_{\overline{\mathbf{TJ}}}$	UnsignedOrNotNumeric	1
`notnums`	$\top_{\overline{\mathbf{TJ}}}$	UnsignedOrNotNumeric	1

TABLE 5.2: Analysis output results with $\overline{\mathbf{TJ}}$ and $\overline{\mathbf{TJ}}^{\star}$.

For example, let us consider the μDyn program reported in Figure 5.8, where the value of `s` is statically unknown. The program takes the strings `s` and puts its non-zero numerical characters into `nums` and the others into `notnums`. If the variable `s` is initialised as "`24kobe8`", at the program point 12 the value of `nums` is "`248`" and the value of of `notnumbs` is "`kobe`". Let consider TAJS and its complete shell and let analyse the program with both the abstract domains. When the variable `s` is initialised with the abstract value UnsignedStr, the analysis output, for the exit point is reported in Table 5.2, where the second and third columns correspond to the result analysis of $\overline{\mathbf{TJ}}$ and $\overline{\mathbf{TJ}}^{\star}$ for the corresponding variable, respectively, and the last column is the increment precision gained by performing the analysis on $\overline{\mathbf{TJ}}^{\star}$. Observe that, for the variables `s` and `i`, we have no precision improvement, as stated by the last column of Table 5.2. As far as the variables `nums` and `notnumbs`, the analysis on $\overline{\mathbf{TJ}}$ returns the top values, whereas the analysis on $\overline{\mathbf{TJ}}^{\star}$ leads to an precision increment, since it returns the abstract value UnsignedOrNotNumeric, for both variables. The precision increment is stated by the last column of Table 5.2 for the variables `nums` and `notnumbs`, since their abstract values (UnsignedOrNotNumeric) on the analysis on $\overline{\mathbf{TJ}}^{\star}$ is distant 1 from the abstract values of the analysis on $\overline{\mathbf{TJ}}$ ($\top_{\overline{\mathbf{TJ}}}$). The total precision entropy of the analysis on $\overline{\mathbf{TJ}}$ and the analysis on $\overline{\mathbf{TJ}}^{\star}$ is 54 and 44, respectively, so the overall precision increment when considering the complete shell, in this case, is equal to 10.

5.7 Discussion

In this chapter we focused on backward completeness in `JavaScript`-specific string abstract domains, and provided in particular the complete shells of TAJS and SAFE string abstract domains with respect to `toNum` and `concat` operations, together with an effective procedure to measure the precision improvement of the analysis when moving to the complete shell. Our results can be easily applied also to JSAI string abstract domain [112], as it can be seen as an extension of the SAFE domain.

Chapter 6

RELATIONAL STRING ABSTRACT DOMAINS

In this chapter we define an Abstract Interpretation framework for relational string analysis. The framework is based on string orders and allows to detect relations between the values of distinct string variables, in contrast to typical string abstract domains which describe the content of each string variable independently. We instantiate the framework to some string order relations of interest. Note that the abstract domains we introduce in this chapter are not language-specific and consequently they may be suitable for analyzing programs written in different programming languages. Moreover, we are able to track relations between string variables and expressions and, based on the string order onto which the framework has been instantiated, we are also able to infer content and shape information of string values. Finally, our framework can be easily combined with basics string abstract domains and lead to preciser analyses.

The contribution of this chapter is still submitted for publication.

Chapter Structure

Section 6.1 highlights the importance of having relational string analysis and explains our contribution. Section 6.2 presents our core language. Section 6.3 formalizes the framework from which relational string abstract domains based on a given strings order can be built and defines some instances of it. Section 6.4 presents the experimental evaluation. Section 6.5 concludes.

6.1 Introduction

A great effort has been spent by the static analysis community in proposing new abstractions, to better approximate and analyse string values. Unfortunately, almost all the existing string abstract domains are in position to track information of single variables used in a program (e.g., if a string contains some characters, or it starts with a given sequence), without inspecting their relationship with other values (e.g., if a string is a substring of another one, despite the fact we do not know anything else about the two string values).

```
String secName(String javaName, String pr1, String pr2) {
  if (javaName.startsWith(pr1)) {
    return pr2 + javaName.substring(4);
  } else if (javaName.startsWith(pr2)) {
    return pr1 + javaName.substring(4);
  } else {
    return javaName;
  }
}
```

FIGURE 6.1: SECNAME sample

To relate with integer abstractions, advanced and sophisticated relational abstractions have been studied and improved over the years in order to track relations between integer variables: a representative example, among others, is the Polyhedra abstract domain [58] proposed in the late 70s, but constantly and heavily improved over the years, as reported by the more recent important works on its optimization, e.g., [21, 20]. Unfortunately, we cannot say the same for string abstract domains, where the majority of them infer only *non-relational* information.

Let us show the problem by considering the Java function SECNAME[1] reported in Figure 6.1. SECNAME takes as input three arguments, `javaName`, `pr1` and `pr2`. Then, if `javaName` starts with `pr1`, the function returns `pr2` concatenated to the substring, from the index 4, of `javaName`. SECNAME behaves analogously when `javaName` starts with `pr2` and it concatenates `pr1` to `javaName.substring(4)`, otherwise `javaName` is returned. The relational information we want to capture here is the one relating `pr1` and `pr2` with `javaName` and the returned value. In particular, we want to infer that `javaName.substring(4)` is always part of the returned value and `pr1` is returned as part of the return value if `javaName` starts with `pr2`, and `pr2` is returned as part of the returned value if `javaName` starts with `pr1`. Unfortunately, in order to catch such information, non-relational abstractions, even the more advanced and sophisticated ones, are not enough and new relational domains for strings are a necessity.

State of the Art

Several relational abstract domains have been proposed for numerical values, such as Polyhedra [58], Octagons [135], Logahedra [98], Pentagons [126], Stripes [66], and Weighted Hexagons [72]. We have been inspired from some of these domains to build the string relational domains that will be presented in Section 6.3. Indeed, consider the Octagons and the Pentagons abstract domains. Octagons track relations of the form $\pm x \pm y \leq k$,

[1] SECNAME is the result of a slight modification made to the function available at `https://www.codota.com/code/java/classes/java.lang.String`

where k is a constant. On the other hand, Pentagons, a less precise domain than Octagons, combines the numerical properties tracked by the Interval domain (i.e., $x \in [n, m]$) and the symbolic ones captured by the Strict Upper Bound domain (i.e., $x < y$). Similarly to the Strict Upper Bound domain, our framework instantiates domains which track information of the form $x \preceq y$ where $\preceq$ is a general order over string variables. Moreover, the framework extension we define in order to track relations between string expressions and variables, like $x + y \preceq z$, has been modelled similarly to Octagons.

Other abstractions have been proposed to infer information about the relations between heap-allocated data structures a program manipulates [179]. In [88], an abstract domain which approximates must and may equalities among pointer expressions has been defined and a relational abstract domain for shape analysis has been presented in [102], built on the top of a set of logical connectives, it represents relations among memory states.

Also for string values a big effort has been made to improve precision of string abstractions, but contrary to the numerical world, most of them only focus on the abstraction of a single variable (cf. Chapter 1).

Contribution

In this chapter we define a general framework upon which relational strings abstract domains can be instantiated, starting from a string order of interest and we introduce a suite of relational abstract domains fitting on the proposed framework, based on length inequality, char inclusion, substring relations. Precisely our framework will be firstly formalized in order to track relations between single string variables, and later extended to infer relations between string expressions and variables, so to appreciate the improvement of the captured information precision.

We conclude presenting the experimental analysis results of the relational string abstract domain based on the substring relation we propose, compared with state-of-the-art general purpose string abstract domains.

6.2 Core Language

As core language we use Imp, a modified version of μDyn (cf. Section 5.3).

6.2.1 Syntax

Imp syntax is given in Figure 6.2. Let P be an Imp program. Each Imp statement is annotated with a label $\ell \in \mathsf{Lab}(\mathsf{P})$ (not belonging to the syntax), where $\mathsf{Lab}(\mathsf{P})$ denotes the set of the P labels, i.e., its program points. We denote by $\ell_{\vdash}$ the initial program point and by $\ell_{\dashv}$ the exit program point.

As it usually happens in static analysis, a program can be analysed by looking at its control-flow graph (CFG for short), i.e., a directed graph that embeds the control structure

```
a ∈ AE ::= x | n | a + a | a - a | a * a | a / a
     | length(s) | indexOf(s,s)
b ∈ BE ::= x | true | false | b && b | b || b | ! b
     | e < e | e == e | contains(s1,s2)
s ∈ SE ::= x | "σ" | substr(s,a,a) | s1 + s2
e ∈ E ::= a | b | s
st ∈ ST ::= ℓ1st ℓ2st ℓ3 | ℓ1skip;ℓ2 | ℓ1x = e;ℓ2
     | ℓ1if (b) ℓ2st ℓ3 else 4st ℓ5 6 | ℓ1while (b) 2st 3 ℓ4
P ∈ IMP ::= ℓ1st ℓ2
where x ∈ X (finite set of variables), n ∈ Z and σ ∈ Σ*
```

FIGURE 6.2: IMP syntax.

```
1x = "a";
2y = "c";
3while (B) {
4x = x + "a"; 5
}6
```

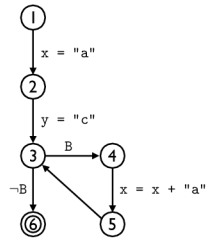


FIGURE 6.3: Example of CFG generation.

of a program, where nodes are the program points and edges express the flow paths from the entry to the exit block. Formally[2], given a program $\mathsf{P} \in$ IMP, we define the corresponding CFG $\mathsf{G_P} = \langle \mathrm{Nodes_P}, \mathrm{Edges_P}, \mathrm{In_P}, \mathrm{Out_P} \rangle$ as the CFG whose nodes are the program points, i.e., $\mathrm{Nodes_P} = \mathsf{Lab}(\mathsf{P})$, the input node is the entry program point, i.e., $\mathrm{In_P} = \ell_{\vdash}$, and the output node is the last program point, i.e., $\mathrm{Out_P} = \ell_{\dashv}$. The algorithm computing the CFG of a program P is standard and can be found in [99, 14]. An example of CFG is depicted in Figure 6.3, where empty edges correspond to edges labeled with `true`.

As observed in [99], the language of the edges label is different from IMP, since the CFG embeds the control structure of the program. In particular, the grammar of the edges label is $\mathrm{IMP}^{\mathsf{CFG}} ::= \texttt{skip} \mid \mathrm{x} = \mathrm{exp} \mid \texttt{b}$.

6.2.2 Concrete Semantics

Since our aim is to analyze CFGs in order to answer questions on the corresponding programs, we need to define the semantics of the edges label, elements of $\mathrm{IMP}^{\mathsf{CFG}}$, which express the effect that each edge has from its entry node to its exit node. Let $\mathrm{VAL} = \mathrm{INT} \cup \mathrm{BOOL} \cup \mathrm{STR}$ be the set of the possible values associated with a variable, such that INT denotes the set of signed integers $\mathbb{Z}$, BOOL denotes the set of booleans $\{\texttt{true}, \texttt{false}\}$, and STR denotes the set of strings Σ^* over an alphabet Σ. We define by $\mathtt{m} \in \mathbb{M} = \mathrm{X} \to \mathrm{VAL}$

[2]Both static analysis process and notation follow the one presented in [99].

the set of memories, where $\mathfrak{m}_\varnothing$ denotes the empty memory. The semantics of expressions is captured by the function $(\!|\ \mathsf{e}\ |\!) : \mathbb{M} \to \textsc{Val}$. Since the semantics of integer and Boolean expressions is standard (and not of interest of this chapter), in the following we only give the concrete semantics of string expressions. Since the semantics of integer and Boolean expressions is standard (and not of interest of this chapter), in the following we only give the concrete semantics of string expressions.

$$
\begin{aligned}
(\!|\ \mathtt{s}_1 + \mathtt{s}_2\ |\!)\mathfrak{m} &= (\!|\ \mathtt{s}_1\ |\!)\mathfrak{m} \cdot (\!|\ \mathtt{s}_2\ |\!)\mathfrak{m} \qquad (\!|\ \mathrm{x}\ |\!)\mathfrak{m} = \mathfrak{m}(\mathrm{x}) \qquad (\!|\ \sigma\ |\!)\mathfrak{m} = \sigma \\
(\!|\ \mathtt{substr}(\mathtt{s}, \mathtt{a}_1, \mathtt{a}_2)\ |\!)\mathfrak{m} &= \sigma_\mathrm{i} \ldots \sigma_{\mathrm{j}-1} \\
&\quad \text{where } \sigma = (\!|\ \mathtt{s}\ |\!)\mathfrak{m}, \mathrm{i} = (\!|\ \mathtt{a}_1\ |\!)\mathfrak{m}, \mathrm{j} = (\!|\ \mathtt{a}_2\ |\!)\mathfrak{m}, 0 \leq \mathrm{i} \leq \mathrm{j} < |\sigma| \\
(\!|\ \mathtt{indexOf}(\mathtt{s}_1, \mathtt{s}_2)\ |\!)\mathfrak{m} &= \begin{cases} \mathsf{idx}(\sigma_1, \sigma_2) & \text{if } \sigma_2 \curvearrowright \sigma_1 \\ -1 & \text{otherwise} \end{cases} \\
&\quad \text{where } \sigma_1 = (\!|\ \mathtt{s}_1\ |\!)\mathfrak{m}, \sigma_2 = (\!|\ \mathtt{s}_2\ |\!)\mathfrak{m} \\
(\!|\ \mathtt{length}(\mathtt{s})\ |\!)\mathfrak{m} &= |(\!|\ \mathtt{s}\ |\!)\mathfrak{m}| \\
(\!|\ \mathtt{contains}(\mathtt{s}_1, \mathtt{s}_2)\ |\!)\mathfrak{m} &= (\!|\ \mathtt{s}_2\ |\!)\mathfrak{m} \curvearrowright (\!|\ \mathtt{s}_1\ |\!)\mathfrak{m}
\end{aligned}
$$

Once we have defined the expression concrete semantics, we are able to formalize the edges label semantics. Abusing notation, we define the function $(\!|\ \mathtt{st}\ |\!) : \mathbb{M} \to \mathbb{M}$ that captures the semantics of elements of $\textsc{Imp}^{\mathsf{CFG}}$.

$$
\begin{aligned}
(\!|\ \mathtt{skip}\ |\!)\mathfrak{m} &= \mathfrak{m} \qquad (\!|\ \mathrm{x} = \mathsf{e}\ |\!)\mathfrak{m} = \mathfrak{m}[\mathrm{x} \leftarrow (\!|\ \mathsf{e}\ |\!)\mathfrak{m}] \\
(\!|\ \mathsf{b}\ |\!)\mathfrak{m} &= \begin{cases} \mathfrak{m} & \text{if } (\!|\ \mathsf{b}\ |\!)\mathfrak{m} = \mathtt{true} \\ \mathfrak{m}_\varnothing & \text{if } (\!|\ \mathsf{b}\ |\!)\mathfrak{m} = \mathtt{false} \end{cases}
\end{aligned}
$$

To conclude, we introduce the notion of store [14], which is the collection of memories for each program point. The store is defined as $\mathfrak{s} \in \mathbb{S} = \mathsf{Lab}(\mathsf{P}) \to \mathbb{M}$ and it associates a memory to each program point.

Static analysis computes (abstract) invariants for each program point; thus, we first define a collecting semantics which associates to each program point, namely to each node of a CFG, the set of its possible memories holding at that program point. This boils down to lifting the concrete semantics $(\!|\ \mathtt{st}\ |\!) : \mathbb{M} \to \mathbb{M}$ (working on single memories), to the collecting semantics $[\![\mathtt{st}]\!] : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$ working on sets of memories. We need also to slightly change the notion of store, since it has to map each program point to sets of memories, namely $\overline{\mathfrak{s}} \in \overline{\mathbb{S}} = \mathsf{Lab}(\mathsf{P}) \to \mathcal{P}(\mathbb{M})$.

Finally, we could use standard fix-point analysis algorithms (i.e., the ones presented in [14, 99, 139]) which returns a store $\overline{\mathfrak{s}}$ such that, for each $\ell \in \mathsf{Lab}(\mathsf{P})$, $\overline{\mathfrak{s}}(\ell)$ is the fix-point collecting semantics (i.e., a set of memories) for the program point ℓ. In this way, unfortunately, from Rice's Theorem, the invariants for each node of a CFG are not computable. It follows that we need abstraction to make static analysis decidable.

6.3 A Suite of String Relational Abstract Domains

In this section, we provide a suite of relational string abstract domains, based on several well known orders over strings. We start proposing a general framework to build string relational abstract domains starting from a given string order. Within this framework, we present three different string relational abstract domains, namely length inequality, char inclusion, and substring domains, with the corresponding abstract semantics of IMP.

6.3.1 General Relational Framework

We aim at capturing relations between string variables in the form $\mathrm{x} \preceq \mathrm{y}$, w.r.t. some (partial or pre-order) relation $\preceq$ over strings, such as, for example, "y is substring of x" or "y is the prefix of x". As introduced in Section 6.1, in the numerical world, such relation is captured by the (strict) upper bound abstract domain [135, 126], which is able to express relations in the form $\mathrm{y} \leq \mathrm{x}$, for some numerical variables x and y. In this section we generalize the upper bound abstract domain to string variables, making it parametric w.r.t. a given string order.

The framework starting point is an order $\sim \subseteq \Sigma^* \times \Sigma^*$ between strings, from which we can build, given the finite set of string variables $\mathrm{X_{str}}$, a new order $\preceq \subseteq \mathrm{X_{str}} \times \mathrm{X_{str}}$ between single string variables. Upon the order $\preceq$ we design a generalized relational string abstract domain.

Definition 6.1 (General string relational abstract domain)**.** Let $\preceq \subseteq \mathrm{X_{str}} \times \mathrm{X_{str}}$ be an order over string variables. The general string relational abstract domain A is defined as

$$\mathrm{A} = \mathcal{P}(\{\mathrm{y} \preceq \mathrm{x} \mid \mathrm{x}, \mathrm{y} \in \mathrm{X_{str}}\}) \cup \{\bot_{\mathrm{A}}\}$$

which ranges over the meta-variable $\mathcal{A}$, such that the top element $\top_{\mathrm{A}}$ corresponds to $\varnothing$ and the bottom element is $\bot_{\mathrm{A}}$. Least upper bound, greatest lower bound and partial order of A are defined as follows.

$$\begin{aligned}
\mathcal{A}_1 \sqcap_{\mathrm{A}} \mathcal{A}_2 &= \{\mathrm{y} \preceq \mathrm{x} \mid \mathrm{y} \preceq \mathrm{x} \in \mathcal{A}_1 \vee \mathrm{y} \preceq \mathrm{x} \in \mathcal{A}_2\} \\
\mathcal{A}_1 \sqcup_{\mathrm{A}} \mathcal{A}_2 &= \mathsf{Clos}(\{\mathrm{y} \preceq \mathrm{x} \mid \mathrm{y} \preceq \mathrm{x} \in \mathcal{A}_1 \wedge \mathrm{y} \preceq \mathrm{x} \in \mathcal{A}_2\}) \\
\mathcal{A}_1 \sqsubseteq_{\mathrm{A}} \mathcal{A}_2 &\iff \mathrm{A}_1 \supseteq \mathcal{A}_2
\end{aligned}$$

where $\mathsf{Clos} : \mathrm{A} \to \mathrm{A}$ is the function performing the transitive closure of the abstract element input.

$\triangle$

The abstract domain A is intended to collect $\preceq$ *must* relations, i.e., informally speaking, if a relation $\mathrm{y} \preceq \mathrm{x}$ is captured in the abstract, it means that it holds in the concrete.

Note that elements of A (cf. Definition 6.1) are sets of relations $y \preceq x$ between single string variables. Moreover, the general abstract domain A is finite, given that the set of string variables is finite and, in turn, also the number of possible relations. Thus, it is trivial to see that the domain $(A, \sqsubseteq_A, \bot_A, \top_A, \sqcap_A, \sqcup_A)$ is a complete lattice, such that its greatest lower bound $\sqcap_A$ and least upper bound $\sqcup_A$ are defined as the union and the intersection between abstract elements, respectively. Concretization and abstraction functions $\gamma_A : A \to \mathcal{P}(\mathbb{M})$ and $\alpha_A : \mathcal{P}(\mathbb{M}) \to A$ are generally defined as

$$\gamma_A(\mathcal{A}) = \bigcap_{y \preceq x \in \mathcal{A}} \{\mathfrak{m} \mid \mathfrak{m}(x), \mathfrak{m}(y) \in \Sigma^*, \mathfrak{m}(y) \sim \mathfrak{m}(x)\} \tag{6.1}$$

$$\alpha_A(\mathsf{M}) = \{y \preceq x \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}(y) \sim \mathfrak{m}(x),\ x, y \in X_{str}\} \tag{6.2}$$

Note that $(\mathcal{P}(\mathbb{M}), \alpha_A, \gamma_A A)$ is a Galois Connection, since $\mathcal{P}(\mathbb{M})$ and A are complete lattices and α_A is a join-morphism.

At this point, we need to define a general and parametric abstract semantics of Imp. In particular, given an abstract domain A, built upon the order $\preceq$, we need to define the function $[\![\ \mathtt{st}\ ]\!]^A : A \to A$ capturing the $\preceq$-relations between variables generated by the statement `st`. We start by defining the parametric abstract semantics of the assignment $x = \mathtt{s}$. Here, the crucial point is the definition of the auxiliary function $\mathsf{extr} : \textsc{se} \to \mathcal{P}(X_{str})$ that extracts from a string expression `s`, all the variables appearing in `s` that are related w.r.t. $\preceq$ with x. In particular, extr should approximate the set of variables appearing in `s` that are $\preceq$-related with x.

Once the extraction function extr is defined, we are able to provide the general abstract semantics of the assignment, which is defined by the following steps:

- **[remove]**

$$\mathcal{A}_r = \begin{cases} \mathcal{A} \setminus \{w \preceq z \mid w = x\} & \text{if x appears at the top-level of } \mathtt{s} \\ \mathcal{A} \setminus \{w \preceq z \mid w = x \vee z = x\} & \text{otherwise} \end{cases}$$

- **[add]**

$$\mathcal{A}_a = \mathcal{A}_r \cup \{y \preceq x \mid y \in \mathsf{extr}(\mathtt{s})\}$$

- **[closure]**

$$[\![\ x = \mathtt{s}\ ]\!]^A \mathcal{A} = \mathsf{Clos}(\mathcal{A}_a)$$

The first phase is **[remove]**, that has to remove, from the input memory $\mathcal{A}$, the relations that surely do not hold anymore after the assignment execution. In particular, we always remove the relations of the form $x \preceq z$, for some $z \in X_{str}$, since x is going to be overwritten, but we remove any relations of the form $w \preceq x$, for some $w \in X_{str}$, iff x does not appear at the top-level of the expression `s`.

Then, the **[add]** phase adds the $\preceq$-relations $y \preceq x$, for each variable y collected in $\mathsf{extr}(\mathtt{s})$. Finally, **[closure]** performs the transitive closure of the abstract memory obtained from **[add]**, i.e., $\mathcal{A}_a$, by means of the function Clos, in order to derive the implicit $\preceq$-relations which are not directly present in $\mathcal{A}_a$.

As far as Boolean expressions are concerned, the only IMP Boolean expressions that generate $\preceq$-relations (for the string domains presented in this chapter) are $\mathtt{contains}(\mathtt{s}_1, \mathtt{s}_2)$, $\mathtt{s}_1 == \mathtt{s}_2$, conjunctive and disjunctive expressions. Note that, given the expression $\mathtt{contains}(\mathtt{s}_1, \mathtt{s}_2)$, we are able to infer $\preceq$-relations only when $\mathtt{s}_1$ is a variable. In all the other cases, no information can be gathered.

$$[\![\mathtt{contains}(x, \mathtt{s})]\!]^A \mathcal{A} = \mathsf{Clos}(\mathcal{A} \cup \{y \preceq x \mid y \in \mathsf{extr}(\mathtt{s})\})$$

Similarly, we can infer $\preceq$-relations in the abstract semantics of $\mathtt{s}_1 == \mathtt{s}_2$ only when either $\mathtt{s}_1$ or $\mathtt{s}_2$ is a string variable.

$$[\![x \texttt{ == } \mathtt{s}]\!]^A \mathcal{A} = [\![\mathtt{s} \texttt{ == } x]\!]^A \mathcal{A} = \begin{cases} \mathsf{Clos}(\mathcal{A} \cup \{y \preceq x, x \preceq y\}) & \text{if } \mathtt{s} = y \in X_{str} \\ \mathsf{Clos}(\mathcal{A} \cup \{y \preceq x \mid y \in \mathsf{extr}(\mathtt{s})\}) & \text{otherwise} \end{cases}$$

As far as the conjunctive and disjunctive expressions semantics, we rely on the least upper bound and greatest lower bound operators given in Definition. 6.1.

$$[\![\mathsf{e}_1 \texttt{ \&\& } \mathsf{e}_2]\!]^A \mathcal{A} = [\![\mathsf{e}_1]\!]^A \mathcal{A} \sqcup_A [\![\mathsf{e}_2]\!]^A \mathcal{A}$$
$$[\![\mathsf{e}_1 \texttt{ || } \mathsf{e}_2]\!]^A \mathcal{A} = [\![\mathsf{e}_1]\!]^A \mathcal{A} \sqcap_A [\![\mathsf{e}_2]\!]^A \mathcal{A}$$

Unlike the assignment case, the abstract semantics of Boolean expressions does not remove any previous substring relation, since such expressions do not affect the (concrete) memory.

Moreover, this general abstract semantics only holds for the abstract domains that we present in this chapter and it may not hold for other abstract domains generated within the framework proposed in this section.

6.3.2 String Length Relational Abstract Domain

Within the formal framework presented before, several relational string abstract domains can be generated. For instance, one may be interested in capturing the relations concerning the length of a string variable w.r.t. another, when they interact during the program execution. Formally, we are interested in identifying the relation $\preceq_{\mathsf{len}} \subseteq X_{str} \times X_{str}$ between string variables such that, given $x, y \in X_{str}$, $y \preceq_{\mathsf{len}} x$ iff the length of y is smaller than or equal to the length of x.[3] We instantiate the general abstract domain reported in

[3] $\preceq_{\mathsf{len}}$ is not a partial order but it is a pre-order binary relation. We will discuss later how this affects the abstract domain gathered information.

Greatest lower bound: $\mathcal{L}_1 \sqcap_{\mathsf{len}} \mathcal{L}_2 = \{\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \mid \mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \in \mathcal{L}_1 \vee \mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \in \mathcal{L}_2\}$
Least upper bound: $\mathcal{L}_1 \sqcup_{\mathsf{len}} \mathcal{L}_2 = \mathsf{Clos}(\{\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \mid \mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \in \mathcal{L}_1 \wedge \mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \in \mathcal{L}_2\})$
Partial order: $\mathcal{L}_1 \sqsubseteq_{\mathsf{len}} \mathcal{L}_2 \iff \mathcal{L}_1 \supseteq \mathcal{L}_2$

FIGURE 6.4: Lattice operations over Len.

Definition 6.1 over the order $\preceq_{\mathsf{len}}$. In particular, we replace the general string order $\preceq$ with $\preceq_{\mathsf{len}}$, obtaining the relational string length abstract domain $\mathsf{Len} = \mathcal{P}(\{\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \mid \mathrm{x}, \mathrm{y} \in \mathrm{X}_{\mathrm{str}}\}) \cup \{\bot_{\mathsf{len}}\}$, which ranges over the meta-variable $\mathcal{L}$, where the top element, denoted by $\top_{\mathsf{len}}$, is $\emptyset$, and $\bot_{\mathsf{len}}$ is the bottom element. Least upper bound $\sqcup_{\mathsf{len}}$ and greatest lower bound $\sqcap_{\mathsf{len}}$ operators and the partial order $\sqsubseteq_{\mathsf{len}}$ are given in Figure 6.4. It is straightforward to prove that $(\mathsf{Len}, \sqsubseteq_{\mathsf{len}}, \sqcup_{\mathsf{len}}, \sqcap_{\mathsf{len}}, \bot_{\mathsf{len}}, \top_{\mathsf{len}})$ is a complete lattice. Similarly, we define the concretization $\gamma_{\mathsf{len}} : \mathsf{Len} \to \mathcal{P}(\mathbb{M})$ and the abstraction $\alpha_{\mathsf{len}} : \mathcal{P}(\mathbb{M}) \to \mathsf{Len}$ functions of the relational string length abstract domain instantiating Equation 6.2 and 6.1 over $\preceq_{\mathsf{len}}$, namely:

$$\gamma_{\mathsf{len}}(\mathcal{L}) = \bigcap_{\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \in \mathcal{L}} \{\mathfrak{m} \mid \mathfrak{m}(\mathrm{x}), \mathfrak{m}(\mathrm{y}) \in \Sigma^*,\ |\mathfrak{m}(\mathrm{y})| \leq |\mathfrak{m}(\mathrm{x})|\}$$

$$\alpha_{\mathsf{len}}(\mathsf{M}) = \{\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ |\mathfrak{m}(\mathrm{y})| \leq |\mathfrak{m}(\mathrm{x})|,\ \mathrm{x}, \mathrm{y} \in \mathrm{X}_{\mathrm{str}}\}$$

$(\mathcal{P}(\mathbb{M}), \alpha_{\mathsf{len}}, \gamma_{\mathsf{len}}, \mathsf{Len})$ is a Galois Connection.

Extraction function of Len. Given $\mathrm{x} = \mathtt{s}$, we can see the string expression $\mathtt{s}$ as an ordered list of concatenated expressions $\mathtt{s}_0, \mathtt{s}_1, \ldots, \mathtt{s}_n$, and the string variables that surely have length less than or equal of x are the ones that are at the top-level of a concatenation appearing in $\mathtt{s}$. For instance, let us consider the assignment $\mathtt{x} = \mathtt{y} + \mathtt{z} + \mathtt{w}$. The substring relations we aim to capture from it are $\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x}$, $\mathrm{z} \preceq_{\mathsf{len}} \mathrm{x}$ and $\mathrm{w} \preceq_{\mathsf{len}} \mathrm{x}$, that is $\mathrm{y}, \mathrm{z}, \mathrm{w}$ have length less than or equal to the length of x. Such variables are collected by the extraction function $\mathsf{extr} : \textsc{se} \to \mathcal{P}(\mathrm{X}_{\mathrm{str}})$ defined as:

$$\mathsf{extr}(\mathtt{s}) = \begin{cases} \{\mathrm{y}\} & \text{if } \mathtt{s} = \mathrm{y} \in \mathrm{X}_{\mathrm{str}} \\ \mathsf{extr}(\mathtt{s}_1) \cup \mathsf{extr}(\mathtt{s}_2) & \text{if } \mathtt{s} = \mathtt{s}_1 + \mathtt{s}_2 \\ \emptyset & \text{otherwise} \end{cases}$$

The abstract semantics of Len is straightforward and reported in Appendix B.1.

6.3.3 Char Inclusion Relational Abstract Domain

In the following, we present the character inclusion relational abstract domain Char, tracking the characters included between a pair of string variables.

Greatest lower bound: $\mathcal{C}_1 \sqcap_{\mathsf{char}} \mathcal{C}_2 = \{\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \mid \mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \in \mathcal{C}_1 \vee \mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \in \mathcal{C}_2\}$
Least upper bound: $\mathcal{C}_1 \sqcup_{\mathsf{char}} \mathcal{C}_2 = \mathsf{Clos}(\{\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \mid \mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \in \mathcal{C}_1 \wedge \mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \in \mathcal{C}_2\})$
Partial order: $\mathcal{C}_1 \sqsubseteq_{\mathsf{char}} \mathcal{C}_2 \iff \mathcal{C}_1 \supseteq \mathcal{C}_2$

FIGURE 6.5: Lattice operations over Char.

Given $\mathrm{x}, \mathrm{y} \in \mathrm{X_{str}}$, we are also interested in capturing "if the characters which appear in y occur in x as well". Formally, we introduce the binary relation $\preceq_{\mathsf{char}} \subseteq \mathrm{X_{str}} \times \mathrm{X_{str}}$ such that $\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x}$ iff the set of characters of y is contained or equal to the set of characters of x. As $\preceq_{\mathsf{len}}$, also $\preceq_{\mathsf{char}}$ is not a partial order, but only a pre-order. We define the relational character inclusion string abstract domain $\mathsf{Char} = \mathcal{P}(\{\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \mid \mathrm{x}, \mathrm{y} \in \mathrm{X_{str}}\}) \cup \{\bot_{\mathsf{char}}\}$, ranging over the meta-variable $\mathcal{C}$, where the top element, denoted by $\top_{\mathsf{char}}$, is $\emptyset$, and $\bot_{\mathsf{char}}$ is the bottom element. Note that $(\mathsf{Char}, \sqsubseteq_{\mathsf{char}}, \sqcup_{\mathsf{char}}, \sqcap_{\mathsf{char}}, \bot_{\mathsf{char}}, \top_{\mathsf{char}})$ is also a complete lattice and its lattice operators are presented in Figure 6.5.

The concretization $\gamma_{\mathsf{char}} : \mathsf{Char} \to \mathcal{P}(\mathbb{M})$ and the abstraction $\alpha_{\mathsf{char}} : \mathcal{P}(\mathbb{M}) \to \mathsf{Char}$ functions are formally defined below, where $\mathsf{char}(\cdot)$ computes the set of characters of a string. As in the length case, $(\mathcal{P}(\mathbb{M}), \alpha_{\mathsf{char}}, \gamma_{\mathsf{char}}, \mathsf{Char})$ is a Galois Connection.

$$\gamma_{\mathsf{char}}(\mathcal{C}) = \bigcap_{\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \in \mathcal{C}} \{\mathfrak{m} \mid \mathfrak{m}(\mathrm{x}), \mathfrak{m}(\mathrm{y}) \in \Sigma^*, \mathsf{char}(\mathfrak{m}(\mathrm{y})) \subseteq \mathsf{char}(\mathfrak{m}(\mathrm{x}))\}$$

$$\alpha_{\mathsf{char}}(\mathsf{M}) = \{\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathsf{char}(\mathfrak{m}(\mathrm{y})) \subseteq \mathsf{char}(\mathfrak{m}(\mathrm{x})),\ \mathrm{x}, \mathrm{y} \in \mathrm{X_{str}}\}$$

The abstract semantics of Char relies on the extraction function defined for Len and can be found in Appendix B.2.

6.3.4 Substring Relational Abstract Domain

The abstract domains presented till now, namely Len and Char, are able to track relations about the lengths and the characters, between a pair of string variables. One main limit of such domains is that they are both based on pre-orders: hence, when we have relations of the form $\mathrm{x} \preceq \mathrm{y}$ and $\mathrm{y} \preceq \mathrm{x}$, we cannot assert that the variables x and y are equal. Moreover, Len loses any information about the content of a string variable, while Char loses any information about the shape of a string variable. Hence, we propose a strictly more precise partial order-based relational string abstract domain, still fitting in the formal framework presented in Section 6.3.1 and tackling the problems of Len and Char.

Given $\mathrm{x}, \mathrm{y} \in \mathrm{X_{str}}$, we introduce the binary relation $\preceq_{\mathsf{sub}}$: $\mathrm{X_{str}} \times \mathrm{X_{str}}$ such that $\mathrm{x} \preceq_{\mathsf{sub}} \mathrm{y}$ iff x is a substring of y. The relation $\preceq_{\mathsf{sub}}$ is a partial order, being reflexive, transitive and anti-symmetric. Thus, we define the relational string abstract domain $\mathsf{Sub} = \mathcal{P}(\{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \mid \mathrm{x}, \mathrm{y} \in \mathrm{X_{str}}\}) \cup \{\bot_{\mathsf{sub}}\}$, ranged over the meta-variable $\mathcal{S}$, where the top element, denoted

Greatest lower bound: $\mathcal{S}_1 \sqcap_{\mathsf{sub}} \mathcal{S}_2 = \{\mathrm{x} \preceq_{\mathsf{sub}} \mathrm{y} \mid \mathrm{x} \preceq_{\mathsf{sub}} \mathrm{y} \in \mathcal{S}_1 \vee \mathrm{x} \preceq_{\mathsf{sub}} \mathrm{y} \in \mathcal{S}_2\}$
Least upper bound: $\mathcal{S}_1 \sqcup_{\mathsf{sub}} \mathcal{S}_2 = \mathsf{Clos}(\{\mathrm{x} \preceq_{\mathsf{sub}} \mathrm{y} \mid \mathrm{x} \preceq_{\mathsf{sub}} \mathrm{y} \in \mathcal{S}_1 \wedge \mathrm{x} \preceq_{\mathsf{sub}} \mathrm{y} \in \mathcal{S}_2\})$
Partial order: $\mathcal{S}_1 \sqsubseteq_{\mathsf{sub}} \mathcal{S}_2 \iff \mathcal{S}_1 \supseteq \mathcal{S}_2$

FIGURE 6.6: Lattice operations over Sub.

by $\top_{\mathsf{sub}}$, corresponds to $\emptyset$, and $\bot_{\mathsf{sub}}$ is the bottom element. Sub is is a complete lattice, i.e., $(\mathsf{Sub}, \sqsubseteq_{\mathsf{sub}}, \sqcup_{\mathsf{sub}}, \sqcap_{\mathsf{sub}}, \bot_{\mathsf{sub}}, \top_{\mathsf{sub}})$ was operator are given in Figure 6.6.

The concretization $\gamma_{\mathsf{sub}} : \mathsf{Sub} \to \mathcal{P}(\mathbb{M})$ and the abstraction $\alpha_{\mathsf{sub}} : \mathcal{P}(\mathbb{M}) \to \mathsf{Sub}$ functions follow the definitions Eq. 6.1 and 6.2 reported in Section 6.3.1 and they are defined as:

$$\gamma_{\mathsf{sub}}(\mathcal{S}) = \bigcap_{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \in \mathcal{S}} \{\mathfrak{m} \mid \mathfrak{m}(\mathrm{x}), \mathfrak{m}(\mathrm{y}) \in \Sigma^*, \mathfrak{m}(\mathrm{y}) \curvearrowright \mathfrak{m}(\mathrm{x})\}$$

$$\alpha_{\mathsf{sub}}(\mathsf{M}) = \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}(\mathrm{y}) \curvearrowright \mathfrak{m}(\mathrm{x}), \mathrm{x}, \mathrm{y} \in \mathrm{X}_{\mathrm{str}}\}$$

$(\mathcal{P}(\mathbb{M}), \alpha_{\mathsf{sub}}, \gamma_{\mathsf{sub}}, \mathsf{Sub})$ is a Galois Connection.

Again, the abstract semantics of Sub relies on the extraction function defined for Len and can be found in Appendix B.3.

6.3.5 Extension to String Expressions

The abstract domains previously proposed are able to track only $\preceq$-relations between single string variables. Consider the fragment below .

```
x = y + y + w;   z = y + w;
```

If we analyze the above fragment with the substring abstract domain, the final abstract memory is $\{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x}, \mathrm{w} \preceq_{\mathsf{sub}} \mathrm{x}, \mathrm{y} \preceq_{\mathsf{sub}} \mathrm{z}, \mathrm{w} \preceq_{\mathsf{sub}} \mathrm{z}\}$, but other substring relations may be inferred, such as $\mathrm{z} \preceq_{\mathsf{sub}} \mathrm{x}$ or $\mathrm{y} + \mathrm{w} \preceq_{\mathsf{sub}} \mathrm{x}$. In the following we extend the framework introduced in Section 6.3.1 in order to catch also such relations.

We introduce the binary relation $\preceq_* \subseteq \mathrm{SE} \times \mathrm{X}_{\mathrm{str}}$ that relates string expressions and string variables. Note that also SE is limited to the expressions encountered in a program. Upon the binary relation $\preceq_*$ we build the new set of abstract memories $\mathrm{A}^* = \mathcal{P}(\{\mathtt{s} \preceq_* \mathrm{x} \mid \mathtt{s} \in \mathrm{SE}, \mathrm{x} \in \mathrm{X}_{\mathrm{str}}\}) \cup \{\bot_{\mathrm{A}}\}$, ranged over the meta-variable $\mathcal{A}^*$ where the top element, denoted by $\top_{\mathrm{A}^*}$, is $\emptyset$, and $\bot_{\mathrm{A}^*}$ is the bottom element. Note that, A^* is still a finite domain, since, given a program $\mathsf{P} \in \mathrm{IMP}$, both its string variables and string expressions are finite. Thus, A^* is a complete lattice, denoted by $(\mathrm{A}^*, \sqsubseteq_{\mathrm{A}^*}, \bot_{\mathrm{A}^*}, \top_{\mathrm{A}^*}, \sqcap_{\mathrm{A}^*}, \sqcup_{\mathrm{A}^*})$, whose operators and and partial order are reported in Figure 6.7. The concretization $\gamma_{\mathrm{A}^*} : \mathrm{A}^* \to \mathcal{P}(\mathbb{M})$ and the abstraction $\alpha_{\mathrm{A}^*} : \mathcal{P}(\mathbb{M}) \to \mathrm{A}^*$ functions form a Galois Connection and are defined as

Greatest lower bound: $\mathcal{A}_1^* \sqcap_{\mathrm{A}^*} \mathcal{A}_2^* = \{\mathtt{s} \preceq_* \mathrm{x} \mid \mathtt{s} \preceq_* \mathrm{x} \in \mathcal{A}_1^* \vee \mathtt{s} \preceq_* \mathrm{x} \in \mathcal{A}_2^*\}$
Least upper bound: $\mathcal{A}_1^* \sqcup_{\mathrm{A}^*} \mathcal{A}_2^* = \mathsf{Clos}(\{\mathtt{s} \preceq_* \mathrm{x} \mid \mathtt{s} \preceq_* \mathrm{x} \in \mathcal{A}_1^* \wedge \mathtt{s} \preceq_* \mathrm{x} \in \mathcal{A}_2^*\})$
Partial order: $\mathcal{A}_1^* \sqsubseteq_{\mathrm{A}^*} \mathcal{A}_2^* \iff \mathcal{A}_1^* \supseteq \mathcal{A}_2^*$

FIGURE 6.7: Lattice operations over A^*.

follows:

$$\gamma_{\mathrm{A}^*}(\mathcal{A}^*) = \bigcap_{\mathtt{s} \preceq_* \mathrm{x} \in \mathcal{A}^*} \{\mathfrak{m} \mid [\![\mathtt{s}]\!]\mathfrak{m}, \mathfrak{m}(\mathrm{x}) \in \Sigma^*, [\![\mathtt{s}]\!]\mathfrak{m} \sim \mathfrak{m}(\mathrm{x})\}$$

$$\alpha_{\mathrm{A}^*}(\mathsf{M}) = \{\mathtt{s} \preceq_* \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ [\![\mathtt{s}]\!]\mathfrak{m} \sim \mathfrak{m}(\mathrm{x}), \mathrm{x} \in \mathrm{X}_{\mathrm{str}}, \mathtt{s} \in \mathrm{SE}\}$$

At this point, we can define the abstract semantics of A^*. Given an assignment $\mathrm{x} = \mathtt{s}$, the abstract semantics of A^* follows the one defined for A: from the expression $\mathtt{s}$ we retrieve any sub-expression that is surely in a $\preceq_*$-relation with x, using the auxiliary function $\mathsf{extr}^\star$. The function $\mathsf{extr}^\star : \mathrm{SE} \to \mathcal{P}(\mathrm{SE})$ is an extension of the previously defined function extr, extracting from a string expression $\mathtt{s}$, in addition to its top-level variables (as extr), also its top-level sub-expressions. For instance, $\mathsf{extr}^\star(\mathtt{y + w + "ab"}) = \{\mathtt{y,\ w,\ w + "ab",\ y + w,\ y + w + "ab",\ "a",\ "b"}\}$. Hence, the abstract semantics of assignment is defined by the following steps:

- **[remove]**

$$\mathcal{A}_{\mathrm{r}}^* = \begin{cases} \mathcal{A}^* \setminus \{\mathtt{s'} \preceq_* \mathrm{z} \mid \mathrm{x} \text{ appears in } \mathtt{s'}\} & \text{if x appaers} \\ & \text{at the top-level of } \mathtt{s} \\ \mathcal{A}^* \setminus \{\mathtt{s'} \preceq_* \mathrm{z} \mid \mathrm{z} = \mathrm{x} \vee \mathrm{x} \text{ appears in } \mathtt{s'}'\} & \text{otherwise} \end{cases}$$

- **[add]**

$$\mathcal{A}_{\mathrm{a}}^* = \mathcal{A}_{\mathrm{r}}^* \cup \{\mathtt{s'} \preceq_* \mathrm{x} \mid \mathtt{s'} \in \mathsf{extr}^\star(\mathtt{s})\}$$

- **[inter-asg]**

$$\mathcal{A}_{\mathrm{i}}^* = \mathcal{A}_{\mathrm{a}}^* \cup \{\mathrm{x} \preceq_* \mathrm{y} \mid \mathcal{A}_{\mathrm{a}}^*(\mathrm{x}) \subseteq \mathcal{A}_{\mathrm{a}}^*(\mathrm{y})\}$$

- **[closure]**

$$[\![\mathrm{x} = \mathtt{s}]\!]^{\mathrm{A}^*}\mathcal{A}^* = \mathsf{Clos}(\mathcal{A}_{\mathrm{i}}^*)$$

The **[remove]**, **[add]** and **[closure]** phases are similar to the ones used in the definition of $[\![\ \cdot\ ]\!]^{\mathrm{A}^*}$. The intermediate phase **[inter-asg]** is the one that differs from the previous definition. If, from the previous steps, any $\preceq_*$-relation of x is contained in the ones of a string variable y, then we can safely say that x is a $\preceq_*$-related to y. Informally speaking, since x is exactly equal to $\mathtt{s}$, x is in relation with any sub-expression of $\mathtt{s}$: if the latter

Greatest lower bound: $\mathcal{S}_1^\star \sqcap_{\mathsf{sub}^\star} \mathcal{S}_2^\star = \{\mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x} \mid \mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x} \in \mathcal{S}_1^\star \vee \mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x} \in \mathcal{S}_2^\star\}$
Least upper bound: $\mathcal{S}_1^\star \sqcup_{\mathsf{sub}^\star} \mathcal{S}_2^\star = \mathsf{Clos}(\{\mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x} \mid \mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x} \in \mathcal{S}_1^\star \wedge \mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x} \in \mathcal{S}_2^\star\})$
Partial order: $\mathcal{S}_1^\star \sqsubseteq_{\mathsf{sub}^\star} \mathcal{S}_2^\star \iff \mathcal{S}_1^\star \supseteq \mathcal{S}_2^\star$

FIGURE 6.8: Lattice operations over $\mathsf{Sub}^\star$.

(denoted by $\mathcal{A}_{\mathrm{a}}^*(\mathrm{x})$) are fully contained in the $\preceq_*$-relations of another variable y (denoted by $\mathcal{A}_{\mathrm{a}}^*(\mathrm{y})$), this means that x is $\preceq_*$-related to y.

The abstract semantics of Boolean expressions is analogous to $[\![\cdot]\!]^{\mathrm{A}}$ and it is reported below.

$$
\begin{aligned}
[\![\mathtt{contains}(\mathrm{x},\mathtt{s})]\!]^{\mathrm{A}^*}\mathcal{A}^* &= \mathsf{Clos}(\mathcal{A}^* \cup \{\mathtt{s}' \preceq_* \mathrm{x} \mid \mathtt{s}' \in \mathsf{extr}^\star(\mathtt{s})\}) \\
[\![\mathrm{x} == \mathtt{s}]\!]^{\mathrm{A}^*}\mathcal{A}^* &= [\![\mathtt{s} == \mathrm{x}]\!]^{\mathrm{A}^*}\mathcal{A}^* = \mathsf{Clos}(\mathcal{A}^* \cup \{\mathtt{s}' \preceq_* \mathrm{x} \mid \mathtt{s}' \in \mathsf{extr}^\star(\mathtt{s})\}) \\
[\![\mathsf{e}_1 \ \mathtt{\&\&}\ \mathsf{e}_2]\!]^{\mathrm{A}^*}\mathcal{A}^* &= [\![\mathsf{e}_1]\!]^{\mathrm{A}^*}\mathcal{A}^* \sqcup_{\mathrm{A}^*} [\![\mathsf{e}_2]\!]^{\mathrm{A}^*}\mathcal{A}^* \\
[\![\mathsf{e}_1 \ \mathtt{||}\ \mathsf{e}_2]\!]^{\mathrm{A}^*}\mathcal{A}^* &= [\![\mathsf{e}_1]\!]^{\mathrm{A}^*}\mathcal{A}^* \sqcap_{\mathrm{A}^*} [\![\mathsf{e}_2]\!]^{\mathrm{A}^*}\mathcal{A}^*
\end{aligned}
$$

Extended Substring Relational Abstract Domain

We extend the substring relational abstract domain following A^*. The extended version of string length and character inclusion relational domains can be found in Appendix B.5 and B.6, respectively.

Let $\preceq_{\mathsf{sub}^\star} \subseteq \mathrm{SE} \times \mathrm{X_{str}}$ be the binary relation such that $\mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x}$ iff $\mathtt{s}$ is a substring of x. For instance, $\mathrm{y} + \mathrm{y} \preceq_{\mathsf{sub}^\star} \mathrm{x}$ means that $\mathrm{y} + \mathrm{y}$ is a substring of x. Morover, let $\mathsf{Sub}^\star = \mathcal{P}(\{\mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x} \mid \mathtt{s} \in \mathrm{SE}, \mathrm{x} \preceq_{\mathsf{sub}^\star} \mathrm{X_{str}}\}) \cup \{\bot_{\mathsf{sub}^\star}\}$ be the extended substring relational abstract domain, ranged over the meta-variable $\mathcal{S}^\star$, where the top element, denoted by $\top_{\mathsf{sub}^\star}$, is $\emptyset$, and $\bot_{\mathsf{sub}^\star}$ is the bottom element. Least upper bound and greatest lower bound operators and partial order are $\sqcup_{\mathsf{sub}^\star}$, $\sqcap_{\mathsf{sub}^\star}$ and $\sqsubseteq_{\mathsf{sub}^\star}$, respectively, and they are reported in Figure 6.8. Similarly to the previous cases, also $(\mathsf{Sub}^\star, \sqsubseteq_{\mathsf{sub}^\star}, \sqcup_{\mathsf{sub}^\star}, \sqcap_{\mathsf{sub}^\star}, \bot_{\mathsf{sub}^\star}, \top_{\mathsf{sub}^\star})$ is a complete lattice, and concretization $\gamma_{\mathsf{sub}^\star} : \mathsf{Sub}^\star \to \mathcal{P}(\mathbb{M})$ and abstraction $\alpha_{\mathsf{sub}^\star} : \mathcal{P}(\mathbb{M}) \to \mathsf{Sub}^\star$ functions, forming again a Galois Connection, are defined as follows:

$$
\gamma_{\mathsf{sub}^\star}(\mathcal{S}^\star) = \bigcap_{\mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x} \in \mathcal{S}^\star} \{\mathfrak{m} \mid [\![\mathtt{s}]\!]\mathfrak{m}, \mathfrak{m}(\mathrm{x}) \in \Sigma^*, [\![\mathtt{s}]\!]\mathfrak{m} \curvearrowright \mathfrak{m}(\mathrm{x})\}
$$

$$
\alpha_{\mathsf{sub}^\star}(\mathsf{M}) = \{\mathtt{s} \preceq_{\mathsf{sub}^\star} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ [\![\mathtt{s}]\!]\mathfrak{m} \curvearrowright \mathfrak{m}(\mathrm{x}), \mathrm{x} \in \mathrm{X_{str}}, \mathtt{s} \in \mathrm{SE}\}
$$

The abstract semantics of Sub is straightforward and can be found in Appendix B.4.

```
r = "not constant";
if ((*$*))
r = r + " passed";
else
r = r + " failed";
assert (r.contains("con"));
assert (r.contains("ed"));
```

Program NOTCON

```
v = readStr();
r = "Elem: \n" + v + "\n";
while ((*$*)) {
v = readStr();
r = r + v + "\n";
}
assert (r.contains("em"));
assert (r.contains("\n"));
assert (r.contains(v));
assert (r.contains("f"));
```

Program REP

FIGURE 6.9: Program samples used for domain comparison.

6.4 Experimental Evaluation

We have implemented $\overline{\mathbf{RSub}}$, a prototypal static analyzer for the toy language presented in Section 6.2 using the new relational string abstract domain $\mathsf{Sub}^\star$, since it subsumes, from a precision point of view, the other abstract domains and their extensions, inferring the set of substring relations holding at each program point. $\overline{\mathbf{RSub}}$ has been implemented and tested over several representative string manipulating fragments[4], both taken from real-world software and hand-crafted. A subset of these fragments are used and discussed in this section in order to show limits and strengths of the aforementioned relational string domain and the corresponding analysis. In Section 6.4.1, we compare our analysis with with prefix $\overline{\mathbf{PR}}$, suffix $\overline{\mathbf{SU}}$, char inclusion $\overline{\mathbf{CI}}$ and bricks $\overline{\mathbf{BR}}$ abstract domains [49] and with the finite state automata $\overline{\mathbf{FA}}$ abstract domain [17] on some simple yet representative code fragments, while in Section 6.4.2 we discuss $\overline{\mathbf{RSub}}$ on a real world example, comparing our analysis with the finite state automata abstract domain. $\overline{\mathbf{RSub}}$ implements trace partitioning [149], that will be used only in the second part of the evaluation.

6.4.1 Test Cases

We consider two code fragments manipulating strings, NOTCON and REP, reported in Fig. 6.9. NOTCON may update the variable `r` either appending the string `"passed"` or `"failed"`. REP iteratively appends to the variable `r` a read string stored in `v` (hence, statically unknown) concatenated with the string `"\n"`. In both the examples, the shortcut ? in the `while` and `if` statements denotes a statically unknown Boolean guard.

Table 6.1 reports the analyses results at the end of the execution of the program NOTCON. In particular, the second column reports the abstract value of `r` at the end of

[4]Available at `https://github.com/UniVE-SSV/rsubs/tree/main/src/test/resources/vmcai`

Domain	`r` abstract value	Asserts
$\overline{\mathbf{PR}}$	`not constant`	✗
$\overline{\mathbf{SU}}$	`ed`	✗
$\overline{\mathbf{CI}}$	$\{n, o, t, c, n, s, a, e, d\}, \{n, o, t, c, n, s, a, p, s, e, d, f, i, l\}$	✗
$\overline{\mathbf{BR}}$	{`not constant passed, not constant failed`)}(1,1)	✓
$\overline{\mathbf{FA}}$	`not constant(passed || failed)`	✓
$\overline{\mathbf{RSub}}$	{`not constant` $\preceq_{\mathsf{sub}}$ r, `ed` $\preceq_{\mathsf{sub}}$ r }	✓

TABLE 6.1: Results at the end of the analysis of NOTCON.

Domain	`r` abstract value	Asserts
$\overline{\mathbf{PR}}$	`Elem:`	✗
$\overline{\mathbf{SU}}$	ϵ (unknown)	✗
$\overline{\mathbf{CI}}$	$\{\,^{\circ}\}, \{\,^{\circ}\}$ (unknown)	✗
$\overline{\mathbf{BR}}$	$\{\star\}(0,+\infty)$ (unknown)	✗
$\overline{\mathbf{FA}}$	`Elem:` $\star$\n$(\star$\n$)^*$	✗
$\overline{\mathbf{RSub}}$	{`Elem:` $\preceq_{\mathsf{sub}}$ r, v $\preceq_{\mathsf{sub}}$ r, r + v + \n $\preceq_{\mathsf{sub}}$ r, v + \n $\preceq_{\mathsf{sub}}$ r, \n $\preceq_{\mathsf{sub}}$ r}	✓

TABLE 6.2: Analyses results of REP.

the analysis and the third column is labeled with ✓ if the corresponding analysis proves that all the `assert` conditions of NOTCON (lines 6 and 7) hold, or with ✗ otherwise. The analyses based on $\overline{\mathbf{PR}}$, $\overline{\mathbf{SU}}$ and $\overline{\mathbf{CI}}$ are not able to precisely verify all the assertions. $\overline{\mathbf{PR}}$ approximates the value of `r` with the common prefix, hence it is able to assert line 6 but not line 7. Similarly, $\overline{\mathbf{SU}}$ approximates the value of `r` with the common suffix, hence it is able to assert line 7 but not line 6. $\overline{\mathbf{CI}}$ is not able to assert neither line 6 nor line 7, since the assertions check the containment of two strings (namely "`con`" and "`ed`") but $\overline{\mathbf{CI}}$ only tracks the *single* characters that the variable `r` may or must contain. On the other hand, $\overline{\mathbf{BR}}$ and $\overline{\mathbf{FA}}$ precisely tracks the possible string values of the variable `r` and successfully verify the assertions. $\overline{\mathbf{RSub}}$ also successfully verifies all the assertions, since it tracks that the strings "`con`" and "`ed`" are definitely substrings of `r`. It is worth noting that also other substring relations of `r` are tracked by $\overline{\mathbf{RSub}}$ (e.g., any substring of the string "`not constant`") but they are not interesting for the assertions that must be verified and they have been omitted from Table 6.1 for space limitations.

Let us now consider REP, that differs from NOTCON since a fix-point computation is involved and it deals with statically unknown strings. In particular, four assertions must be checked: the ones at lines 7-9 certainly holds, the one at line 10 *may* be holding, since the value of `v` is statically unknown. The analyses results are reported in Table 6.2, where in the second column "unknown" means that the corresponding analysis has returned the top abstract value, and the symbol $\star$ denotes "any possible string". The third column is labelled with ✓ if the corresponding analysis proves all the `assert` conditions at lines 7-9 of REP or with ✗ otherwise.

$\overline{\mathbf{PR}}$ is able to verify the assertion at line 7 but not the ones at lines 8-9, since it loses

any information on the rest of the string, except for the common prefix and it does not track the fact that the variable `v` is certainly contained in `r`. $\overline{\mathbf{SU}}$, $\overline{\mathbf{CI}}$ and $\overline{\mathbf{BR}}$ analyses lose any information about the value of `r`, abstracting it as the corresponding top abstract value, making them unable to verify the assertions at lines 7-9. $\overline{\mathbf{FA}}$ abstracts the value of `r` as the regular expression reported in Table 6.2 hence it correctly verifies assertions at lines 7-8 but not the one at line 9, being unable to track the relation between the variables `r` and `v`. Even if statically unknown strings are manipulated by REP, $\overline{\mathbf{RSub}}$ behaves as $\overline{\mathbf{FA}}$ as far as assertions at lines 7-8 are concerned, since the strings `Elem:` and `\n` are definitely substrings of `r`, and it verifies the assertion at line 9, since it tracks that the variable `v`, independently from its abstract value, is a substring of the variable `r`. Analysis results for $\overline{\mathbf{RSub}}$ are reported in the last row of Table 6.2.

6.4.2 Evaluating a Real World Sample

We conclude this section evaluating and discussing $\overline{\mathbf{RSub}}$ on SECNAME, reported in Fig. 6.1 in the introduction, compared with the string analysis based on $\overline{\mathbf{FA}}$.

Table 6.3 reports the analysis results of SECNAME, where we denote by `res` the abstract value returned by SECNAME. The second column reports the ones obtained without trace partitioning, while the third column reports the results obtained applying trace partitioning. Let us focus on the first case. Since the function parameters are statically unknown, the Boolean guards at lines 2-4 are statically unknown as well. Consequently, $\overline{\mathbf{FA}}$ needs to take into account the values returned at lines 3, 5 and 7: unfortunately, being the parameter statically unknown, each line returns the top abstract value, losing any information about the value returned by SECNAME, as reported by the second line of Table 6.3. The same happens for $\overline{\mathbf{RSub}}$: each abstract value returned by each `return` statement (lines 3, 5 and 7) does not share any substring relation with the others, hence, lubbing the results, $\overline{\mathbf{RSub}}$ approximates the returned values of SECNAME as the top abstract value.

Now, let us consider the same analysis with trace partitioning, whose results are reported in the third column of Table 6.3; each **[i]** denotes a trace and the corresponding abstract value returned at line i. As far as $\overline{\mathbf{FA}}$ is concerned, the situation remains unaffected w.r.t. the analysis without trace partitioning. In particular, each returned abstract value still corresponds to the top abstract value. On the other hand, $\overline{\mathbf{RSub}}$ is able to precisely infer the information reported in the third column of Table 6.3, describing the possible substring relations at each `return` hotspot. In particular, three possible cases are detected: (i) when $\texttt{pr2} \preceq_{\mathsf{sub}} \texttt{javaName}$ the results definitely contains `pr1` and `javaName.substring(4)`, (ii) when $\texttt{pr1} \preceq_{\mathsf{sub}} \texttt{javaName}$ the results definitely contains `pr2` and `javaName.substring(4)`, (iii) in other cases, the results simply contains `javaName`.

Even if $\overline{\mathbf{FA}}$ provides a precise string analysis, it is limited when dealing with statically unknown inputs, as highlighted in the previous example. On the opposite, even if $\overline{\mathbf{RSub}}$ does not perform a *collection* of the possible string values of a variable, tracking only the

Domain	`res` abstract value	`res` abstract value (with trace partitioning)
$\overline{\mathbf{FA}}$	$\top$	**[2]** $\top$ **[5]** $\top$ **[7]** $\top$
$\overline{\mathbf{RSub}}$	$\top$	**[2]** $\left\{ \begin{array}{l} \texttt{pr1} \preceq_{\mathsf{sub}} \texttt{res} \\ \texttt{javaName.substring}(4) \preceq_{\mathsf{sub}} \texttt{res} \end{array} \right\}$ where $\{\texttt{pr2} \preceq_{\mathsf{sub}} \texttt{javaName}\}$ **[5]** $\left\{ \begin{array}{l} \texttt{pr2} \preceq_{\mathsf{sub}} \texttt{res} \\ \texttt{javaName.substring}(4) \preceq_{\mathsf{sub}} \texttt{res} \end{array} \right\}$ where $\{\texttt{pr1} \preceq_{\mathsf{sub}} \texttt{javaName}\}$ **[7]** $\{ \texttt{javaName} \preceq_{\mathsf{sub}} \texttt{res} \}$

TABLE 6.3: Analysis results of SECNAME.

definite substrings of a variable, it treats variables and expressions in a symbolic way, making itself able to infer useful information, such as in the SECNAME analysis. For instance, we can surely assert that the string `javaName.substring(4)` is a substring of the result and that the result surely includes part of the input (`javaName`).

6.5 Discussion

In this chapter, we introduced a general framework to generate new relational abstract domains starting from pre- and partial orders on string values. In particular, we introduced a new relational substring domain, $\mathsf{Sub}^\star$, showing its impact on the accuracy of the analysis with respect to state-of-art general-purpose string abstract domains. It is worth noting that, in this chapter, $\mathsf{Sub}^\star$ has been used as a *standalone* abstract domain. Of course, further accuracy can be achieved by combining $\mathsf{Sub}^\star$ with existing non-relational domains through the well-known reduced product operator. In fact, even the simple reduced product between the constant string abstract domain and $\mathsf{Sub}^\star$ would lead to more accurate analysis results, as it would be able to retrieve the exact value of some sub-expressions.

Chapter 7

STRING MANIPULATION IN WATERMARKING SCENARIOS

The last chapter of the thesis is somehow orthogonal with respect to the previous ones, as it moves to the usage of string manipulation in an applicative scenario. We investigate how a semantics-based approach to string manipulation may be effective when combined with watermarking techniques for ownership protection.

The content of this chapter reports contributions published in [85].

Context

Information security systems protect digital assets from piracy and prying eyes. As shown in [5], those systems can be distinguished into two categories: cryptography and information hiding. Cryptography aims to hide secret information, transforming a given text into a cypher one (encryption phase), with the possibility of reversing the transformation and thus restore the original information (decryption phase). On the other hand, information hiding, as its name suggests, looks for embedding unnoticeable data in the cover media such as: images, audio, video, and texts. Based on the reason why some information is hidden in some particular cover media, we can distinguish two techniques: steganography and watermarking. Steganography hides a secret message into a cover media in order to transmit it. Watermarking, on the contrary, hides data in cover media to protect their intellectual property from digital piracy (see [103]), and thus claim their ownership.

Thus, digital watermarking is that framework that aims to prove ownership of digital cover data by persistently embedding, visibly or not, a watermark[1] in the latter digital assets. The embedding has to guarantees usability of the watermarked data, leaving them intact and recognisable. Then, a detection and extraction process of the watermark may follow, to claim the intellectual property [53, 158].

[1]Precisely, the watermark is the set of all the marks introduced into a cover data by a watermarking technique [3].

Briefly, watermarking techniques were firstly proposed to protect multimedia data[2] [3], exploiting the information redundancy they contain, and the limits of the human visual and auditory systems, to embed the watermark [104]. Watermarking has been subsequently extended to digital texts, due to the increased demand of digital publishing [104]. Also software have been affected by unauthorised copies and illegal reuses, then software watermarking techniques have been developed [61]. In particular, software watermarking techniques, according to the nature of the watermark extraction process, can be distinguished in [53]: static, dynamic, and abstract. In [146], authors defined a software watermarking system based on program semantics and abstract interpretation that defines the previous techniques as its instances.

Piracy affected also relational data, and thus watermarking database techniques have been developed. Firstly meant for numerical data, relational database watermarking techniques are based on the assumption that it is tolerable that watermarked databases contain a small amount of errors [28], despite the possibility of compromising the answers of SQL queries based on numerical conditions.

Chapter Structure

Section 7.1 introduces the watermarking methodology for relational data to protect their integrity and intellectual property. Moreover, it highlights the problem of dealing with the semantic perturbation which may rise by watermarking relational data and explains our contribution. Section 7.2 defines the motivating examples that let clear the need of the elements we introduce in our technique. Section 7.3 gives an overview about semantic similarity theory and text watermarking techniques. Section 7.4 introduces our watermarking approach, emphasizing the definition of the elements related to the preservation of the semantic consistency, which constitutes a special feature of our work. Details related to the implementation are also given. Section 7.5 presents the validation of our approach through an experimental evaluation. Section 7.6 concludes.

7.1 Introduction

Relational databases are enterprise software systems, introduced in the 1970s [37], where data are stored and eventually analysed to find hidden relations among them which are useful to take strategic decisions. Data stored in relational databases can be pirated, illegally redistributed, tempered, and their ownership claimed, as happens to all the other digital assets (images, audio, video and texts). Indeed, the internet growth resulted in a multitude of web-based services through which data are continuously transmitted and easily accessible.

[2]Multimedia data refer to images, audio, and video.

Symbol	Description
SK	Secret key only known by the data owner.
PK	Primary key identifying the tuple.
η	Number of tuples in a relation R being watermarked.
γ	Fraction of tuples being watermarked $\gamma \in [1, \eta]$. Also known as TF.
ν	Number of attributes considered for the embedding.
ξ	Range of less significant bits (*lsb*) to embed the mark.
ω	Number of watermarked tuples after the embedding.

TABLE 7.1: AHK approach notation.

How can we protect the integrity and the intellectual property of data? Relational database watermarking [4] has been proposed as a new field of security, to protect data property value. The main element it uses is the watermark, i.e., a stream of binary elements called marks, and it consists of two phases: watermark embedding and watermark extraction [3]. Generally speaking, we can classify relational database watermarking techniques into two categories [130]: those embedding the watermark in the data, causing distortions, e.g., [106, 111, 189], and those that do not cause them, e.g., [28, 29, 89]. The distortion-based techniques are mostly oriented to resist aggressive attacks as they are conceived to protect data from false ownership claims [92]. Moreover, relational databases watermarking techniques may be distinguished by: the type of watermark information to be embedded, the type of the attribute into which marks are embedded, their granularity level (bit level, character level, attribute level, tuple level), the detection and verification process in use (deterministic or probabilistic, blindly or non-blindly, publicly or privately), and their ultimate purpose [93]

The first watermarking technique for relational data was proposed by [4]. Also known as the AHK algorithm, this distortion-based approach defines the notation of the elements belonging to a relation R (see Table 7.1), embeds the marks in numerical attributes, introduces a solid criteria for the selection of the places for the watermark embedding, and it has been used by many authors [33, 65, 165] as starting point for other watermarking proposals.

Relational database watermarking techniques deal with different issues [92]. Among them, three are of particular interest: the watermark capacity (i.e., the optimal amount of marks that can be embedded in a relational database), its robustness (i.e., the ability of relational watermarking techniques to resist against malicious or unintentionally cyber incidents), and imperceptibility (i.e., the ability of distortion-based relational watermarking techniques to not affect the usability of the data). Each of these features is highly linked to the other two, and this is the reason why it is necessary to consider a trade-off among them in the design of relational watermarking techniques. Indeed, for example, one way to increase the robustness of a watermarking technique is by embedding more marks, which also increases its capacity, but this may compromises the imperceptibility, affecting the

data usability and giving clues to attackers of the watermark presence. On the other hand, the higher the imperceptibility, the higher is the robustness of the watermark, but this happens despite of its capacity.

State of the Art

After the approach of [4], several relational database watermarking techniques were proposed using different types of attributes to embed the watermark. Despite that, dealing with semantic perturbations provoked by the distortion is an issue that has been ignored most of the time. Among the few works addressing this problem there are the techniques by [26], by [70] and by [69].

In [26], authors proposed a technique to protect the privacy and ownership of medical data. They worked with categorical attributes, and they performed the watermark embedding as permutations of categorical values using a Domain Hierarchy Tree (DHT) of the selected attribute. This scheme does not consider more complex data types (e.g., multi-word textual[3]), and despite the use of the DHT for reducing the semantic perturbations, the changes compromise the inter-attribute semantic consistency. Furthermore, the latter approach reduces the watermark capacity by marking only one attribute per tuple, and if there was no DHT, then the watermark synchronization would be compromised, as the DHT is the structure on which the embedding and the extraction processes rely.

In [69, 70] the distortion is controlled by using ontologies, seeking the preservation of the inter-attribute semantic consistency. In both the approaches, the requiring ontologies (defined for specific contexts) directly impact the blindness of the techniques, and make the watermark synchronization dependant on additional external information. Furthermore, just a single numerical attribute per tuple is selected to embed the watermark, which limits the application of the ontologies, considering the potential they may offer to increase the watermark capacity, among other things. In the end, these approaches depend on the PK of R to embed the watermark, which makes easy to compromise the watermark detection in scenarios where R is used separately from the database.

Contribution

In this chapter, we propose a semantic-driven approach for watermarking multi-word textual attributes in relational databases, to protect their ownership. In order to preserve the meaning, fluency, grammaticality, writing style and value [104] of the multi-word textual attributes, the distortion is meant as a substitution of words that are strongly semantically similar in a certain context (i.e., synonyms).

Our aims are to achieve a high degree of robustness without compromising the imperceptibility, increment the capacity taking care of the semantic of the data, and preserve

[3]We refer to multi-word attributes as textual attributes formed by one or more than one sentence.

Student					
IdNumber	**StudentName**	**StudentSurname**	**Subject**	**Score**	**ProfessorJudgment**
1001	John	Oliver	Mathematics	95	John has improved a lot.
1002	Justin	Fitzgerald	Physics	69	Justin has problems to pass Physics.
1003	Andrea	Russo	History	98	Andrea is the first in his History class.
1004	Karla	Olivare	Mathematics	100	Karla is an outstanding student.

TABLE 7.2: Motivating example.

the results of the queries performed over the watermarked data in comparison to those that are obtained using the same queries over the unwatermarked data. This avoids that the distortion caused by the watermark embedding from affecting the decision making of the organizations using and deploying the data.

The way our approach is conceived allow us to achieve a high watermark synchronization, making our scheme resilient against attacks based on the elimination of tuples and/or attributes. On the other hand, by involving other aspects to the watermarking process (e.g., the elements forming the relation structure, information corresponding to other data types, and the low redundancy of the stored data), our technique gets resilient against the *random synonym substitution attack*. Our approach's features increase the chaotic nature of the mark embedding places selection, making difficult for attackers to determine their locations and to compromise the watermark detection by overwriting them.

We introduce also a novel approach to analyse watermark capacity through the calculation of the index c_W, which expresses the technique's resilience degree to malicious operations. This new measure is described in Section 7.4 highlighting its differences to traditional capacity measurement for relational data watermarking. Combining our proposal with numeric cover type relational watermarking techniques, allows the increasing of the watermark capacity without compromising its imperceptibility, which increases also the robustness, making our technique effective for any practical scenarios.

7.2 Motivating Examples

Using numeric attributes to embed the watermark gives high coverage to relational watermarking techniques, making possible the increment of the watermark capacity. Nevertheless, accomplishing the imperceptibility requirement at plain sight, numerical distortion compromises SQL query results based on numeric conditions.

Example 7.1. Consider the relation **Student** depicted in Table 7.2, where the attribute **IdNumber** denotes the primary key of the relation itself. According to the query below for selecting the students who have passed a certain grade (**Score** ≥ 70), only the students {John, Andrea, Karla} are recovered.

```
SELECT StudentName
FROM Student
WHERE Score ≥ 70
```

In the case in which the attribute **Score** is selected to embed a mark, despite performing a passive distortion, e.g., by just using the two less significant bits (*lsb*), the result of the query above would be different. Indeed, Justin Fitzgerald could be given among the students passing the grade if the value of the 2nd *lsb* is modified, changing, for example, the score from 69 to 71.

⬡

The distortion caused by the change of the *lsb* of numerical values in a relation is not relevant when the values of the attribute chosen to embed the mark are not in the boundaries of some criteria for data recovery or their classification (e.g., changing the score of Andrea from 98 to 96 or 99, depending of the *lsb* selected as mark carrier, would not produce a different answer to the query above). But when this is not the case, such a distortion may lead to taking decisions based on wrong assumptions. Therefore, it follows that numerical distortions compromise the semantic of the tuples, despite the latter distortions being traditionally controlled by defining the maximum amount of tolerable error over the numerical attributes being watermarked.

Embedding the marks in textual attributes avoids compromising the results of queries based on numerical conditions, but applying the distortion over the *lsb* of a textual value compromises the watermark imperceptibility requirement.

Example 7.2. Given the relation defined in Table 7.2, consider the value of the attribute **ProfessorJudgment** for the tuple with **IdNumber** = 1002, i.e., "Justin has problem to pass Physics.". Changing one of the two *lsb* of this textual value will provoke changing "Physics" to "Physicr" or "Physicq" making perceptible the distortion and creating a meaningless word.

⬡

Notice that even when the marks are embedded in textual attributes exploiting the limitations of the human vision for increasing the watermark capacity (e.g., by adding extra white spaces between words [6] or using invisible characters according to the database encoding [131]), is easy for the attacker to detect the position of the marks through computational techniques. Thus, for the aforementioned reasons, when the marks have to be embedded into textual attributes, a different approach is required.

7.3 Preliminaries

As mentioned in Section 7.1, the relational watermarking technique we propose marks multi-word textual attributes and preserves the semantic consistency between the original

database and the watermarked one performing semantically similar words substitutions, taking care of the context in which the words fall. Thus, below we give a brief overview of semantic similarity theory and semantic-based text watermarking techniques.

7.3.1 Semantic Similarity Theory

Semantic similarity is about computing the resemblance between the meanings of textual entities (e.g., words, sentences, texts), that are not necessarily lexically similar [24, 96] and it has application in many research fields [2, 143, 155, 166] such as: Natural Language Processing tasks (e.g., Word Sense Disambiguation and Synonym Detection), Artificial Intelligence, Cognitive Science, Psychology, Information Retrieval and Bio-Informatics.

In the literature, several methods for measuring semantic similarity between textual entities have been proposed [96]. They depend on one or several knowledge sources (e.g., taxonomies, thesaurus, ontologies) and rely on different theoretical properties [24]. A measure of similarity takes as input two textual entities and returns a numeric score that quantifies how much they are alike [166]. Formally, let e_1 and e_2 be two textual entities, $\mathsf{Sim}(e_1, e_2)$ denotes the semantic similarity between e_1 and e_2.

Different words that have highly related meanings are called synonyms. Generally speaking, synonym words belong to the same node in a hierarchical knowledge organization scheme and their semantic similarity is maximized [159].

7.3.2 Text Watermarking

According to classifications of text watermarking techniques presented in [5, 104, 110, 192], we summarise them as follows:

a) Image-based approaches, where a text document, whose content is seen as a series of text images, is used to embed the watermark bits. In general, these methods are resilient against typical image watermarking attacks and format-based attacks.

b) Structure-based approaches, where imperceptible changes to the text structure, features and font are made, into which the watermarking information to be hidden is encoded. These techniques are more likely to be vulnerable to very simple attacks, such as: the text retyping attack and the copy paste to notepad attack, and to the use of Optical Character Recognition (OCR) technologies.

c) Syntactic-based approaches apply syntactic transformation to plain text document structures in order to embed the watermark. These schemes have been proved to be efficient with agglutinative languages like Turkish, but in general they are not adequate for English language.

d) Semantic-based approaches embed the watermark into the semantic structure of text documents, where text meaning analysis and text transformations are performed

using natural language processing algorithms. Semantic-based text watermarking schemes are resilient against retyping attacks or to the use of OCR programs, but prone to weaknesses related to natural language processing.

Notice that syntactic-based and semantic-based text (or content-based) watermarking schemes fall into the linguistic-based approach category. Therefore, they are highly dependant on the type of language in use, which represents a disadvantage in a scenario where languages rapidly evolve, and focused on do not (or minimally) alter the meaning of the cover text.

Synonym Substitution Approach

Image-based and structure-based approaches are useful when the text is forced to be displayed by using specific means. On the other hand, in the case in which the text is stored as content (e.g., stored as multi-word textual attribute in relational databases), independent from its graphical representation, the text can actually be displayed in multiple ways, making those techniques useless. Consequently, content-based text watermarking techniques are better suited to be used in the context of relational textual database watermarking. In particular, we focused on the synonym substitution method for watermarking textual documents, where certain words are replaced with their synonyms preserving the semantic consistency between the original cover text and the watermarked one.

Firstly exploited by steganography [180], synonym substitution technique was later used to watermark plain text document [169]. Still, this watermarking technique is limited to the English language and highly depends on the quality of the text processing tools, like the word sense disambiguator (i.e., a technique that aims to identify which sense of a word is used in a sentence). Moreover, synonym substitution watermarking techniques are vulnerable to synonym substitution attacks.

In [169], to overcome random synonym substitution attacks, authors proposed a lexical watermarking system based on substituting words with homographs[4] from their synonym set, and using meaning-preserving generalizing substitutions. Then, in order to guarantee the context-dependency between synonyms, they implemented a semi-automatic interactive encoding mechanism that allows a person designated to decide on the acceptability of the substitutions given by the system.

[4]Two or more words are homographs if they are spelled the same way but differ in meaning and origin, and sometimes in pronunciation [169].

7.4 Semantic-based Watermarking Approach

As pointed out in Example 7.2, modifying the *lsb* of a textual attribute value may, for example, introduce syntactic errors or cause semantic inconsistencies, leading to the imperceptibility requirement violation of the watermarking technique.

In this section, we present our semantic-based watermarking approach for relational data. Necessary condition for our technique to be applied is that the target relation must contain at least one multi-word textual attribute into which we will embed the watermark. The distortion is thought as synonym substitution.

Our proposal is focused on increasing the watermark capacity, without affecting its imperceptibility, achieving an high degree of robustness against typical relational watermarking attacks and textual watermarking attacks. Indeed, by considering multi-word textual attributes as cover type, multiple synonym substitutions can be performed over a single attribute value, resulting in the increment of the embedded marks, despite some sentences being composed of just few words, overcoming the downside of watermarking short documents, which reduces the watermark capacity [104]. Moreover, the capacity of the watermark increases when numerical attributes, in addition to multi-word textual attributes, are considered to embed the watermark. Also, taking care of preserving the meaning of the watermarked text (by using the synonym substitution approach with a proper word sense disambiguation), the imperceptibility remains untouched, which results in a direct increment of the technique's robustness.

Notice that when dealing with multi-word textual cover type, there are malicious operations focused on compromising the watermark embedded in textual documents (e.g., random synonym substitution attack) that must be considered to prove the effective robustness of our approach.

7.4.1 Architecture of the Proposal

Figure 7.1 depicts the architecture of the watermark embedding process of our proposal. As usual, the watermark extraction involves the same elements of the embedding procedure but it is performed in the opposite direction.

Our watermark embedding procedure (fully described in Section 7.4.2) relies on a relational database DB storing one or several relations R with at least one multi-word textual attribute, one or several knowledge sources N, and a watermark WM. The choice of the knowledge source(s) is let free, but it has to take care of the semantic links between words. So that, on the latter, we can use a similarity engine to verify semantic consistency properties. A word sense disambiguation WSD module is needed for selecting the proper set of synonyms, depending on the context in which the words that are candidates to be replaced fall.

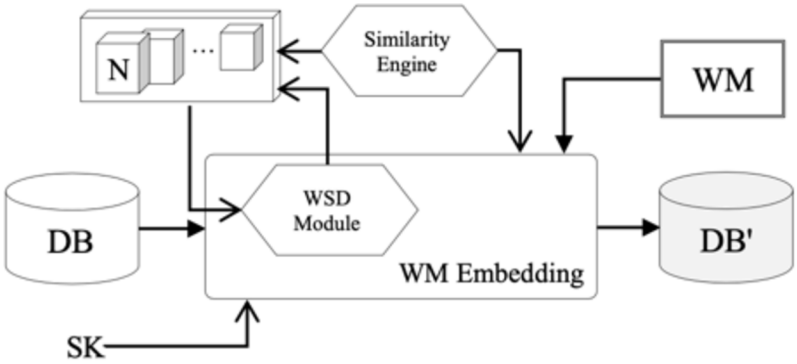


FIGURE 7.1: Embedding process architecture.

We also encourage the use of meaningful sources for the generation of the watermark considering that over this kind of signals can be applied methods to enhance the quality of the extracted watermark, contributing to its recognition despite the execution of aggressive attacks over the watermarked data. The secret key SK will be only known by the data owner and its complexity will be crucial against malicious operations (e.g., the brute force attack), as the security of watermarking techniques is based on the secrecy of the parameter values [92]. Finally, a distorted database DB' is produced. Notice that our watermarking technique modifies R only for the values that are selected for the embedding. Every other value remains the same.

Similarity Engine

Consider a relation R(PK, $A_1, \ldots A_m$) belonging to a database DB which stores at least one multi-word textual attribute. Precisely, let A_h (with $h \in [1,m]$) be a multi-word attribute in R, and let r_k be the k-th instance of R. Then, we refer to $r_k.A_h$ as the value of the attribute A_h with respect to the tuple r_k. Moreover, let s be a sentence in $r_k.A_h$, and assume that the word w part of s has been selected by a procedure P (see Algorithm 7 in Section 7.4.2) to embed the watermark, i.e., to be replaced with its synonym w' (see Algorithm 8 in Section 7.4.2). The embedding is performed if and only if the replacement complies the *intra-attribute consistency* and the *inter-attribute consistency* semantic properties.

Notice that, when it is possible, we replace only one word per sentence of a multi-word attribute value, and the semantic distortion can be embedded in more than one attribute per tuple.

Definition 7.1 (Intra-attribute consistency)**.** Let $r_k.A_h \in R$ be the value of the multi-word attribute A_h for the k-th instance of R. Let s be a sentence in $r_k.A_h$, $w \in s$ be the word to replace, and let w' be the candidate substitute word. Then, s* denotes the sentence s where replacement has been applied, i.e., s[w/w']. Finally $r_k.A_h$[s/s*] denotes the distorted result. We say that [w/w'] is a substitution intra-attribute consistent if the semantic similarity score between $r_k.A_h$ and $r_k.A_h$[s/s*] is higher than or equal to a threshold δ. Formally:

$$\mathsf{Sim}(r_k.A_h, r_k.A_h[s/s^*]) \geq \delta$$

△

Definition 7.2 (Inter-attribute consistency)**.** Given r_k, i.e., the k-th instance of R, where watermark will be embedded, let r_k^* be the distorted result, i.e., $r_k[r_k.A_h/r_k.A_h[s/s^*]]$. Moreover, let ϕ be a function mapping tuple values to their concatenation (by means of "␣and␣"). Following Definition 7.1, we say that [w/w'] is a substitution inter-attribute consistent if the semantic similarity score between $\phi(r_k)$ and $\phi(r_k^*)$ is higher than or equal to a certain threshold μ. Formally:

$$\mathsf{Sim}((\phi(r_k), \phi(r_k^*)) \geq \mu.$$

△

The value of the thresholds δ and μ depends on the chosen semantic similarity measure.

The Word Sense Disambiguation Module

The correct functioning of the WSD module is a key element for the success of our approach. Two important issues depend on the WSD module performance: (i) maintaining the semantic value of the database being protected (ii) and the guarantee of achieving a high watermark synchronization.

WSD is considered an open research field in natural language processing. The main challenges come due to the fact that words often change meanings depending on the context they are used. For example, the noun *tree* can be used to refer to the programming data structure, but also to the living organism belonging to the vegetable kingdom. Thus, the set of synonyms allowed to replace *tree* must be shrunken according to certain context. The same happens with words used as adjectives. The adjective *hard* can be used to refer to someone with a sturdy temperament, to express determinism in business dealings, or to describe a feature of a solid object. If the ambiguity of the word is not taken away considering the context, there is a high probability the value of the text will be compromised once the word replacement is performed [104].

Let D be a function that returns the ordered set $\mathcal{Z}$ of synonyms of a word w given a context ς, denoted by $\mathcal{Z} \leftarrow D(w, \varsigma)$. The following rules need to be accomplished:

1. $w \in D(w, \varsigma)$
2. $\forall t \in D(w, \varsigma) : D(t, \varsigma) = D(w, \varsigma)$

We denote by $\mathcal{Z}[t]$ the t-th element of the set $\mathcal{Z}$.

For any word belonging to $\mathcal{Z}$, the set of synonyms for the given context must be the same to maintain the semantic value of the text from where w and ς were selected. Of course, if the WSD module does not work properly, these rules can be violated considering

the same word can be part of different synonym sets given by other contexts (see Fig. 7.2 where the word "point" has multiple sets of synonyms relying on different contexts in which it can be used). In general, the correct functioning of the WSD module will depend on the accuracy of the implementation of the function D, responsible for obtaining the set of synonyms of w that more fit the context ς.

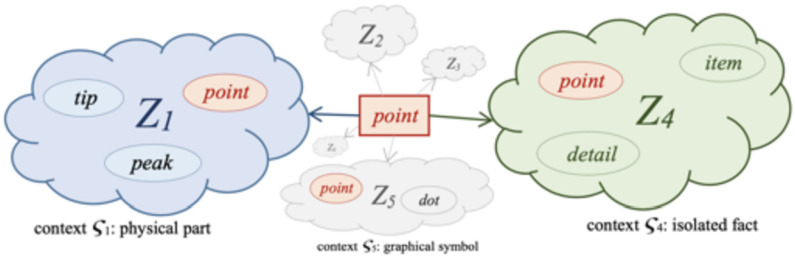


FIGURE 7.2: Synonyms sets linked to a word w.

On the other hand, synchronization is the process of aligning two signals in time or space [60]. Considering the embedded and extracted watermarks as those signals, achieving a high watermark synchronization relies on extracting the exact same marks that were embedded. If WSD fails, rules 1. and 2. are not accomplished. Then, for example, there is the possibility that a mark embedded considering the synonyms of the set $\mathcal{Z}_1$ is extracted looking at synonyms of the set $\mathcal{Z}_4$. Because of that, wrong mark values will be detected, compromising the quality of the extracted watermark and its synchronization. On the contrary, if previous rules are not violated, the value of the textual attribute being watermarked is preserved and the watermark synchronization is guaranteed.

Data Quality Preservation

The keeping of watermarked data quality will mainly depend on the WSD module and the parameters defining the maximum allowable semantic distortion to perform the marks embedding. According to [169], using ambiguous words increases the resilience of the watermarking scheme against attacks, but not all documents being watermarked tolerate this kind of operation, which reduces the options to perform word replacement.

On the other hand, as long as rules 1. and 2. are accomplished, the word used to replace w will be obtained from $\mathcal{Z}$, and the value of the mark detected during the extraction process will match the embedded one. Also, by defining the maximum tolerated semantic distortion, it is avoided the use of words belonging to $\mathcal{Z}$ that might involve ambiguity, increasing the probability of falling in another synonym set. Thus, the equivalency above guarantees the preservation of the data quality, no matter the nature of the text of R being protected.

7.4.2 Watermarking Procedure

Distortion-based relational watermarking techniques consist of two processes: (i) the embedding of the watermark (ii) and its extraction [92]. To achieve the watermark synchronization it is required the use of the same parameter values in both processes. Moreover, the extraction is performed when it is required to demonstrate the watermark presence in the data, as evidence in case of ownership claims, among others.

Algorithm 7 watermarkingEmbedding procedure

Input: R, SK, γ, $\mathcal{A}$, WM, φ, χ, N, $\mathcal{C}$, δ, μ

Output: R', i.e., the watermarked relation

1: **for** tuple $r \in R$ **do**
2: $k_r = VPK(SK \circ r_K)$
3: **if** $(k_r \bmod \gamma) = 0$ **then**
4: $\mathcal{A}' \leftarrow \Theta(\mathcal{A}, r_K, \varphi)$
5: **for** $v \in \mathcal{A}'$ **do**
6: $\mathcal{S} \leftarrow \Lambda(v, \chi)$
7: **for** sentence $s \in \mathcal{S}$ **do**
8: $k_s = H(SK \circ k_r \circ \Gamma(s))$
9: $i = k_s \bmod \Upsilon(s).length$
10: $w = \Upsilon(s)[i]$
11: $\varsigma \leftarrow$ getSense(w, s, N)
12: $\mathcal{Z} \leftarrow$ getCandidates(ς, w, N)
13: $w' =$ getSubstitute(WM, $k_s, \mathcal{Z}, \mathcal{C}, \vartheta$)
14: $s^* \leftarrow s[w/w']$
15: $v^* \leftarrow v[s/s^*]$
16: **if** $Sim(v, v^*) < \delta$ **then**
17: rollback embedding
18: **else**
19: $r^* \leftarrow r[v/v^*]$
20: **if** $Sim(\phi(r), \phi(r^*)) \geq \mu$ **then**
21: $r \leftarrow r^*$
22: $v \leftarrow v^*$
23: commit embedding
24: **else**
25: rollback embedding

Watermark Embedding

Algorithm 7 presents the details of the watermark embedding procedure of our approach. Given a relation R, for each tuple $r \in R$, the values of the multi-word textual attributes composing the list $\mathcal{A}$ are analyzed. Then, the virtual primary key k_r is generated using the VPK function (line 2). The input of the latter function results from the concatenation

($\circ$) between a secret key SK, and data represented by r_K identifying the tuple r (e.g., the relation's PK or other virtual primary keys generated by external schemes).

Following, in the case in which the `if-statement` condition is satisfied, a filter φ is applied (by the function Θ) to the multi-word attribute values in $\mathcal{A}$, in order to exclude those considering their content and links with the other attributes of the tuple (e.g., exclusion of sentences containing acronyms or abbreviations). In this way, we add an extra step to help maintain *inter-attribute consistency* while high unpredictability is incorporated into the technique [5]. Attributes passing the latter filter are stored in the set $\mathcal{A}'$ (lines 3-4).

Similarly as above, for each multi-word attribute value v in $\mathcal{A}'$, a filter χ is applied (by the function Λ) to the sentences in v, to exclude those that do not accomplish the conditions to be considered for the embedding process (e.g., involving just sentences composed of more than certain number of words). Sentences accomplishing the conditions defined in χ are finally considered for the embedding, and they are stored in the set $\mathcal{S}$ (lines 5-6).

For each sentence in $\mathcal{S}$ the key k_s is generated (from a one-way hash function H that takes as input the concatenation of SK, k_Γ and the elements of the sentence do not tolerating changes obtained by the Γ function) that identifies the sentence inside the multi-word attribute value, and using k_s the word w to be replaced is selected (lines 7-10). Notice that Υ is a function that returns, as an array of words, the elements of a sentence tolerating substitutions.

Then, according to the sense ς of the selected word in the sentence under consideration (obtained using the getSense method), the set $\mathcal{Z}$ of synonyms of w is obtained (by using the getCandidates method) (lines 11-12). Both getSense and getCandidates functions are based on the set of rules and the definitions given by the knowledge source(s) N.

Algorithm 8 getSubstitute procedure

Input: $WM, k_s, \mathcal{Z}, \mathcal{C}, \vartheta$

Output: w'

```
i = k_s mod WM.length
m ← WM[i]  set_order(Z, C)
if m = 1 then
    return Z[0]
else
    return Z[ϑ]
```

Finally, the mark to be embedded and the new word w' to replace w are selected (line 13). Algorithm 8 defines the function getSubstitute where the mark is selected according to the value of k_s. The set of candidate substitute words is sorted according to the criteria $\mathcal{C}$, and the new word is chosen depending on the value of the mark to be embedded. In lines 14-25, the replacement of the sentence in the carrier attribute and in the corresponding

[5] Each attribute's value contributes itself in varying the elements of R selected as mark carriers.

tuple is performed. The substitution will only be carried out if *intra-attribute* and *inter-attribute* consistencies properties are not violated.

On the other hand, if a word is not detected in the knowledge sources (i.e., *cross-linguality problem* [2]), our approach ignores the position from the marking's candidates and proceed with the rest of the data stored in the relation.

Watermark Extraction

The watermark extraction process is similar to the embedding process but is performed in the opposite direction. Also, it must be performed using the same parameter values employed to embed the watermark to guarantee its right synchronization. In the extraction process, for the same mark position in the watermark, several elements are recovered, and before assigning the mark final value, a majority voting is performed. In this way, the effect of attacks based on low aggressive data modifications are avoided.

The extraction is performed with no need to check the semantic distortion between the words replaced to carry out the marks embedding (i.e., without considering the similarity metrics value). Nevertheless, the extracted watermark quality depends on the knowledge source(s) and on the precision of the WSD module, since words can be assigned to a set of synonyms different from those considered for the embedding, adding noise to the extracted signal. This is because, in the new set of synonyms, the original word can occupy a different position, assigning to the extracted mark a wrong value.

7.4.3 Analysis of the Watermark Capacity

Since relational data have no fixed order, sequential watermarking approaches are vulnerable to subset reverse order attacks [92]. To overcome this vulnerability, techniques have been designed performing a *pseudo-random selection* of both the watermark source elements used to generate the marks and the embedding locations in R. In general, this operation has been achieved by using a specific definition of Equation (7.1). Nevertheless, besides contributing to robustness against subset reverse order attacks, *pseudo-random selection* makes it difficult for the attackers to predict embedding locations for overwriting or removing of marks.

$$\mathcal{V}(\mathrm{x}, \mathrm{y}) \bmod \mathcal{M}_{\mathrm{X}} \tag{7.1}$$

where, $\mathcal{V}(\mathrm{x}, \mathrm{y})$ is a value generated using data from a generic position (x, y) of R, and $\mathcal{M}_{\mathrm{X}}$ is the maximum value of a given range. An example of a particular definition of this expression can be seen in line 10 of Algorithm 7, where $\mathcal{V}(\mathrm{x}, \mathrm{y})$ is defined in line 9 as $\mathrm{k_s}$.

Because of *pseudo-random selection*, during the embedding process some marks are selected more than once while others are entirely ignored. Embedding the same mark multiple times leads the technique to be resilient against update attacks, if a majority voting

is performed over each mark position in the extraction process. On the other hand, if the number of excluded marks is too high, the watermark synchronization can be compromised. Indeed, the latter process would suffer from the the same negative consequences as when aggressive attacks based on updating or deleting data are performed.

The *pseudo-random selection* downside can be reduced if the number of times the watermark source elements are considered increases. This can be achieved by marking a higher volume of data while the same watermark source is used.

In our approach, new elements to increase the watermark capacity, without affecting the imperceptibility requirement, are introduced. Moreover, we propose a new metric for measuring the watermark capacity which considers the different number of times each mark is selected during the embedding process. The latter metric allows the evaluation of the capacity in function of the technique's robustness, since it assigns to each mark a weight depending on the number of times it is embedded in R. In this way, the difficulty of compromising each mark in the watermarked data is reflected in the metric.

For techniques embedding one mark per selected tuple, the number of embedded marks E is equal to the number of marked tuples ω. If, besides the numeric cover type, multi-word textual attributes are considered (following the approach we proposed above), the number of embedded marks increases for each tuple according to Equation (7.2), where λ_n represents the number of numeric attributes, λ_s the number of marked sentences, and $\eth$ the number of marks embedded in each sentence. If the watermark capacity increases due to marks embedded on multi-word textual attributes (besides those embedded on numerical attributes), using an effective WSD module, the technique becomes more resilient without compromising data usability.

$$E \approx \omega * (\lambda_n + \eth * \lambda_s) \tag{7.2}$$

Even if no numerical attributes are considered and only one multi-word textual attribute is selected to perform the embedding, more than one mark can be embedded per tuple, according to $\eth$, which still constitutes an increment of the capacity compared to other watermarking techniques.

On the other hand, the number of marks selected for the embedding (denoted by m_e) using as reference the watermark size (denoted by n) is commonly used to measure the technique's watermark capacity. We define that metric as the binary capacity, given by c_b according to Equation (7.3). The downside of c_b is that all marks present the same weight, and only their inclusion/exclusion represents information for the measurement.

$$c_b = m_e * 100/n \tag{7.3}$$

We consider the number of times each mark is embedded and we propose the weight-based capacity metric, given by c_w according to Equation (7.4).

$$c_w = \sum_{i=0}^{n-1} (\varkappa(m_i))/n \tag{7.4}$$

where, $\varkappa(m_i)$ represents the number of times the mark m_i was embedded. This is given since not all marks are selected the same number of times. In Figure 7.3 we used a scale of colors to illustrate with an example the differences between c_b and c_w. The value of c_w corresponds to the mean of the number of times all marks of the watermark are embedded.

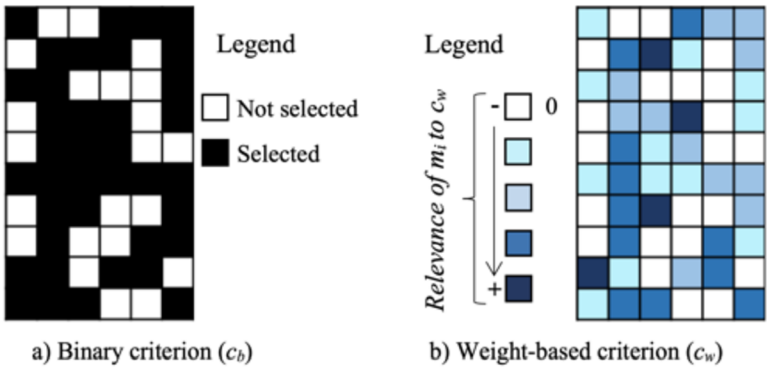


FIGURE 7.3: Criteria to evaluate the watermark capacity.

The higher c_w the better, but it is also required $\rho_w \approx 0$, being ρ_w the standard deviation of the number of times each mark is embedded. This tells us that each mark was selected multiple times evenly, adding higher relevance to the watermark recognition, increasing the probability of its detection despite the execution of benign updates and attacks over R.

Notice that the highest value of c_w will be difficult to achieve as this would mean embedding each mark evenly the maximum possible number of times, and since the process presents *pseudo-random nature*, this is not expected.

In general, c_b does not reflect how embedding each mark multiple times contributes to obtain a different degree of resilience as c_w does. This is important to measure as when more redundancy is achieved for the embedding of a mark value, it is more difficult to compromise its value with update attacks, if a majority voting of all recovered values is performed during watermark extraction.

7.4.4 Considerations for the Adversary Model

One of the major challenges textual watermarking techniques based on synonyms substitution face is the *random synonyms substitution* attacks. This vulnerability is linked to the tolerance a text has with respect to synonyms substitution on it, depending on the

context in which the data was used. Performing embedding of marks in multi-word textual attributes through semantically similar words replacement, we must consider adding resilience to this threat, as well as to the rest of the malicious operations an attacker may perform over the database relation.

Depending on the context in which data are used, we have a margin of marks allowed to be embedded. If the number of marks that can be embedded is high, an high degree of robustness can be achieved, otherwise, the technique's resilience gets compromised. Moreover, the data context also reduces the attacker's freedom to perform aggressive operations if he wants to preserve the quality of the data. Therefore, using textual watermarking to relational data increases the difficulty to perform effective attacks.

Increasing the capacity by considering both, the numerical and textual cover types, allows achieving higher robustness, making hard to compromise the watermark detection. Several elements of the relation, such as attributes storing float numbers, single sentences or several paragraphs, can be selected to embed the marks, based on multiple features (e.g., numerical ranges, minimum number of nouns required per sentence, among others) which highly increase the complexity to detect the marks embedding locations. Furthermore, beyond the complexity of the *pseudo-random selection* of the marks embedding places in the relation, we take advantage of the multi-word textual data structure to add a high entropy to the embedding procedures.

[4] defined Equation (7.5) to get the probability for the attacker to successfully detect the embedding locations used by the data owner. While ω refers to the number of tuples watermarked by the data owner, γ_A, ν_A, and ξ_A denote respectively the tuple fraction, the number of attributes, and the *lsb* considered by the attacker.

$$P\{\text{success}|\omega\} = \left(1 - \frac{1}{2\gamma_A \nu_A \xi_A}\right)^{\omega} \tag{7.5}$$

Considering all those elements, it is difficult to detect the marks embedding positions used by the data owner. Despite that, we increase the difficulty by adding the element ζ_A to Equation (7.5). Equation (7.6) extends Equation (7.5) as follows:

$$P\{\text{success}|\omega\} = \left(1 - \frac{1}{2\gamma_A \nu_A (\xi_A + \zeta_A)}\right)^{\omega} \tag{7.6}$$

The term ζ_A defines the complexity that derives from the consideration of marks embedding locations among multi-word textual cover types. It expresses the possibility of embedding one mark in the whole attribute value, or marking each sentence with one mark, or embedding multiple marks in each sentence. Since all embedding positions for textual attributes are also generated using *pseudo-random selection* and considering the increment of the number of elements to know by the attacker, the probability of performing successful attacks reduces. Furthermore, by accomplishing security and public system

requirements [3], we add secrecy to parameters' values, increasing the difficulty of attackers to detect embedding locations.

7.5 Experimental Results

Following the recommendation given in Section 7.4.1 of considering meaningful sources to generate the watermark WM, we validated our approach by using binary images as WM sources. Besides the benefits previously mentioned, this type of data allows taking the simplest pixel value for the mark generation, which contributes to perform less aggressive distortions during the watermark embedding compared to techniques generating marks from color (e.g., [190]) or gray-scale images (e.g., [191]).

To analyze the effect of the watermark length variation, images of different sizes were used. Samples of them are shown in Figure 7.4: a) the Universiti Teknologi Malaysia (UTM) logo (82×80 pixels), b) the logo of the World Wildlife Fund (WWF) (40×45 pixels), and c) the Chinese character dào (20×21 pixels). By convention, we used the red color to highlight the missed pixels due to watermark incomplete embedding or malicious operations by attackers.

FIGURE 7.4: Samples of the binary images used as WM sources.

To know the quality of the extracted WM two metrics were used: the Correction Factor (CF) and the Structural Similarity Index (SSIM). The Correction Factor, defined by Equation (7.7), is used to compare the pixels of the image generated from the embedded WM (given by $\mathrm{Img_{emb}}$) against the pixels of the image generated from the extracted WM (given by $\mathrm{Img_{ext}}$). In the equation, variables h and w represent the height and the width of the images respectively. The maximum value of CF is 100, meaning that the WMs are identical. In the case in which CF = 0, then the embedded and the extracted WMs are completely different.

$$\mathrm{CF} = \frac{\sum_{i=1}^{h}\sum_{j=1}^{w}(\mathrm{Img_{emb}}(i,j) \oplus \overline{\mathrm{Img_{ext}}(i,j)})}{h \times w} \times 100 \tag{7.7}$$

The SSIM is oriented to obtain an appreciation of the image's quality closer to human perception. The index is calculated according to Equation (7.8), using multiple windows. The windows are defined by x and y and present common size $N \times N$. The range of possible values taken by this metric in this work is between 0 and 1, where 1 means there

exists a perfect structural similarity between the embedded and the extracted image, and 0 indicates no structural similarity.

$$\mathrm{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1) + (2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{7.8}$$

The symbols μ_x and μ_y represent the average of x and y respectively, σ_x^2 and σ_y^2 their respective variance, and σ_{xy} their covariance. The elements C_1 and C_2 are two stabilization constants.

The data to embed the marks was the data set *Amazon Fine Food Reviews*. The structure is depicted in Table 7.3, from where we mostly used the attribute '*Text*', also storing the text with the highest length. We also used the first 30.000 tuples out of 500,000 to compare our results with previous works.

We used the relation's PK to perform the watermark synchronization. To avoid the use of the PK we recommend the generation of virtual primary keys by using the Ext-Scheme [81] or the HQR-Scheme [83]. These schemes were originally proposed for numerical attributes, but they can be applied by combining numerical and textual cover types as well.

Attribute	**Type**	**Description**
ProductId	String	Id. of the product
UserId	String	Id of the user
ProfileName	String	Name of the user
HelpfulnessNumerator	Numeric	Numerator of the fraction of users who found the review helpful
HelpfulnessDenominator	Numeric	Denominator of the fraction of users who found the review helpful
Score	Numeric	Rating of the product
Time	Numeric	Time of the review (unix time)
Summary	String	Review summary
Text	String	Text of the review

TABLE 7.3: Structure of the dataset "Amazon Fine Food Reviews".

On the other hand, as knowledge source, we used WordNet [132], which consists of a lexical database of the English language, where nouns, verbs, adjectives, and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept [147]. So, given a word w, and the context where w is used (i.e., the meaning of the sentence to which w is part of), WordNet returns the appropriate synset of w. Notice that, in our experiment, the set of synonyms $\mathcal{Z}$ (introduced in Section 7.4) will correspond to a synset in WordNet.

For the evaluation of our approach, we implemented a client-server architecture application using Java 1.8 programming language for the client-side, the Oracle Database

12c for the server-side. We used WordNet 3.1 database files with the Java API jwnl 1.4.1 rc2 for accessing and working with WordNet resources and ws4j 1.0.1 for using Semantic Relatedness/Similarity algorithms already developed. The WSD module was based on the Lesk algorithm [171], which compares the word definitions with the definitions of the rest of the words presented in the sentence, finding the more convenient context for its use. According to that, the most appropriate set of synonyms can be chosen. Finally, the runtime environment was a 3.60 GHz Intel i7-4790 PC with 16.0 GB of RAM running on Windows 10 OS.

7.5.1 Improvement of the Watermark Capacity

We performed the watermark capacity analysis comparing our approach with two other techniques, [151], which uses only one attribute, and [82] with two attributes. Of all techniques using an image to generate the watermark (i.e., Image-Based Watermarking (IBW) [92]) these techniques constitute a representation of the ones more recent, used to mark one or several attributes per tuple. We selected only one attribute to mark with our approach to show WM capacity improves even compared to two-attributes embedding, thanks to the selected cover type. The techniques we compare with performed the watermark embedding on the numerical cover type, but by involving the same number of tuples we can appreciate how much the watermark capacity increases for our approach.

Table 7.4 shows the value of c_b (see Equation (7.3) Section 7.4.3) obtained for each technique. In the table, columns titles "S & H"' refers to Sardroudi and Ibrahim's technique, "G. et al."' to Pèrez Gort et al.'s and "Prop." to our proposal. Given that, the number of marks missed by using our approach is lower than the number of marks missed by using the other techniques. Indeed, there are fewer red pixels in the images of the WMs synchronized by our proposal. The main reason for WM improvement is because of for some cases the values stored in the attribute *"Text"* are composed of more than one sentence. If allowed, we only embed one mark per sentence selecting a common *nouns* from it. By more than one mark can be used as the carrier, in which case WM capacity will be even higher.

Table 7.5 shows the value of c_w (see Equation (7.4) Section 7.4.3) with the correspondent ρ_w for each case, giving a clear idea about how we not only improve the value for c_b but for c_w as well, increasing the probability of overcoming attacks based on data updates.

The experiments to register the capacity values were applied over a set of 30000 tuples. For the case of Sardroudi & Ibrahim's and Pérez Gort et al.'s, it was used the numerical data set *Forest Cover Type* [39] as these techniques were designed for marking numerical values. Also, the watermark embedding with Pérez Gort et al.'s technique was performed with Attribute Fraction equal to 5 in order to mark only two attributes per tuple.

Once WM capacity increase was proven, it is critical to guarantee marks detection, otherwise, it will not be possible to recognize the WM signal in the protected data. In

TF	UTM			WWF			Dào		
	S & H	G et al.	Prop.	S & H	G et al.	Prop.	S & H	G et al.	Prop.
2									
	88.34%	98.61%	99.91%	99.94%	99.94%	99.94%	99.76%	99.76%	99.76%
5									
	57.93%	81.46%	93.67%	95.16%	99.72%	99.89%	99.76%	99.76%	99.76%
10									
	35.64%	56.99%	75.35%	78.77%	94.88%	99.44%	99.29%	99.76%	99.76%
20									
	19.25%	34.28%	48.96%	53.88%	78.66%	91.83%	97.38%	99.52%	99.76%
40									
	9.66%	18.14%	27.99%	31.27%	51.88%	69.56%	80.95%	94.04%	98.81%

TABLE 7.4: Value of c_b for different techniques.

TF	UTM			WWF			Dào		
	S & H	G et al.	Proposal	S & H	G et al.	Proposal	S & H	G et al.	Proposal
2	2 (±1.47)	4 (±2.12)	6 (±2.73)	7 (±2.95)	15 (±4.20)	24 (±5.12)	33 (±5.92)	67 (±8.96)	105 (±11.18)
5	0 (±1.27)	1 (±1.51)	2 (±1.87)	3 (±1.81)	6 (±2.56)	10 (±3.27)	13 (±3.65)	26 (±5.46)	43 (±6.79)
10	0 (±0.80)	0 (±1.26)	1 (±1.23)	1 (±1.43)	3 (±1.79)	5 (±2.28)	6 (±2.72)	13 (±3.73)	21 (±4.67)
20	0 (±0.51)	0 (±0.78)	0 (±1.07)	0 (±1.18)	1 (±1.36)	2 (±1.64)	3 (±1.86)	6 (±2.72)	10 (±3.20)
40	0 (±0.34)	0 (±0.50)	0 (±0.66)	0 (±0.72)	0 (±1.16)	1 (±1.12)	1 (±1.37)	3 (±1.89)	5 (±2.33)

TABLE 7.5: Value of c_w with its correspondent ρ_w for each experiment.

the following, the results focused on testing the WM detectability are shown. It is also analyzed the way the WSD module precision determines the quality of the extracted WM.

7.5.2 Detectability Analysis

The detectability of marks in our approach is directly linked to the precision of the WSD module. Since for WM embedding are used set of synonyms of the selected word, it is not expected data quality degradation, but if for WM extraction, the WSD module does not assign the same set of synonyms used for the embedding, then several marks will be recovered with wrong values, adding noise to the extracted WM signal. If the signal is too noisy, WM synchronization can be compromised, making impossible its identification.

The main goal of this work is not to improve the WSD algorithms, but to use those already defined that guarantee high precision for marks detection. The WSD module precision will be denoted as $\mathcal{P}$, which is obtained according to Equation (7.9), where $\mathcal{W}_T$

TF	UTM	WWF	Dào
2	0.9479	0.9498	0.9518
5	0.9478	0.9500	0.9517
10	0.9518	0.9513	0.9516
20	0.9482	0.9422	0.9466
40	0.9532	0.9472	0.9463

TABLE 7.6: WSD precision during WM detection.

represents the number of tagged words (i.e., words selected for sense disambiguation) and $\mathcal{W}_{\mathrm{CT}}$ the number of words correctly tagged (i.e., words that during the extraction process were linked to the same synonym set used for WM embedding).

$$\mathcal{P} = \mathcal{W}_{\mathrm{CT}}/\mathcal{W}_{\mathrm{T}} \tag{7.9}$$

The WSD module we used is based on the Lesk algorithm [171]. The precision described by this module was registered through a set of experiments, whose results are shown in Table 7.6.

Notice that when combining WSD along with the watermarking technique, WSD lack of precision can be compensated by the majority voting performed over WM extraction. Then, the higher c_w with ρ_w closer to zero, the stronger the effect of the majority voting to overcoming low WSD precision. Table 7.7 shows the results experimentally supporting this point, linking the quality of the detected WM to the results of Table 7.5 despite the precision weaknesses shown in Table 7.6.

The results shown in Table 7.7 correspond to a set of experiments with different values for TF and WM sources. For each case, it is shown the embedded WM (the small image), the detected one (the big image), and the value of SSIM with a percentage corresponding to the number of pixels not matching between the two images. The low value of this metric compared to those shown in Table 7.6 directly endorses our statement.

Previous results show how wrong mark values are not reflected in red pixels, but in black and white, added to wrong regions of the image. This can be understood as a *Gaussian noise* which degree is directly linked to the precision of the WSD module. In Table 7.8 are shown other results, which describe the quality of the extracted WM in more detail. This time, the quality of the detected WM is given respect to both, the embedded WM and the original one. Of course, since depending on the parameter's values the original WM is not usually entirely embedded, the higher quality will be the one given respect to the embedded WM.

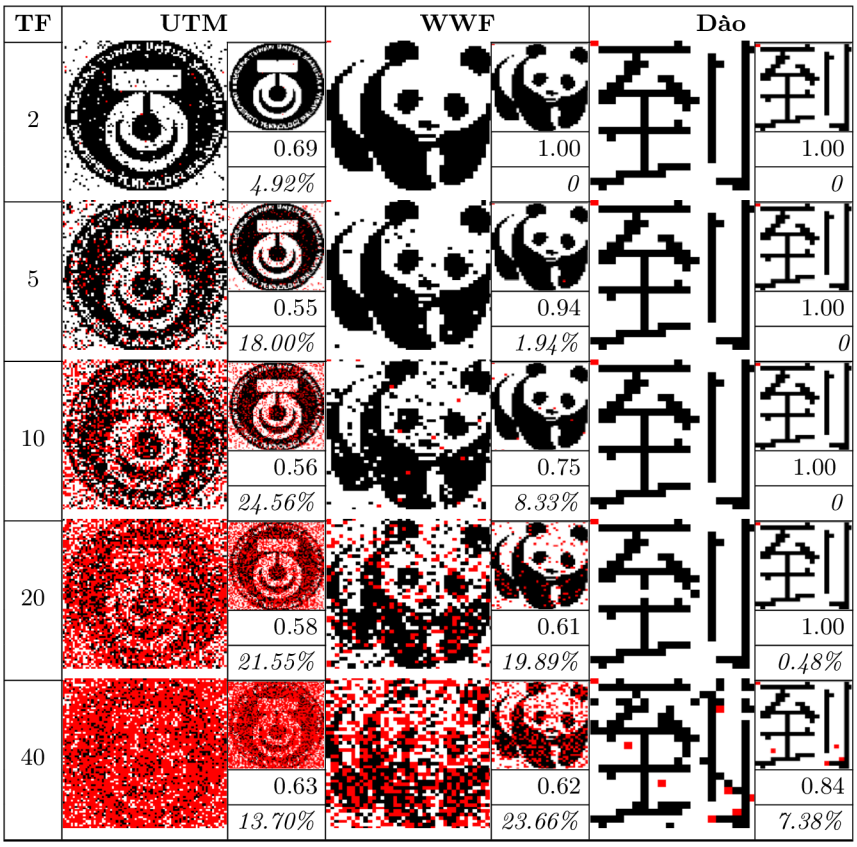


TABLE 7.7: Quality of the detected WM with respect to the embedded one.

TF	UTM				WWF				Dào			
	vs. Embedded		vs. Original		vs. Embedded		vs. Original		vs. Embedded		vs. Original	
	CF	SSIM	CF	SSIM	CF	SSIM	CF	SSIM	CF	SSIM	CF	SSIM
2	99.42%	0.69	95.06%	0.68	100%	1	99.94%	0.99	100%	1	99.76%	0.99
5	88.34%	0.55	80.06%	0.49	100%	0.94	98.00%	0.93	100%	1	99.76%	0.99
10	71.11%	0.56	61.32%	0.37	98.77%	0.75	91.72%	0.75	100%	1	99.76%	0.99
20	54.42%	0.58	37.85%	0.25	83.54%	0.61	76.66%	0.54	100%	1	99.28%	0.99
40	46.51%	0.63	21.9%	0.17	67.41%	0.62	55.33%	0.41	98.07%	0.84	91.9%	0.83

TABLE 7.8: Detectability achieved for each WM over different numbers of marked tuples.

Data detectability is benefited from the combination between the WSD module, which takes care of imperceptibility, and majority voting, which allows being tolerated WSD lack of precision. The positive impact of this effect increase when WM size decreases or the number of tuples being watermarked increases. The following experiments are meant to analyze how data usability and watermark imperceptibility are maintained. This two WM requirements are critical, since if are not accomplished, the technique becomes useless for

TF	UTM	WWF	Dào
2	0.3752	0.3655	0.3146
5	0.3765	0.3652	0.3142
10	0.3763	0.3597	0.3153
20	0.3768	0.3591	0.3094
40	0.3961	0.3590	0.3068

TABLE 7.9: Value of T_w for previous experiments.

TF	Iter.	WUP	JCN	LCH	LIN	RES	PATH	LESK
2	44510	44.64×10^3	4.73×10^{11}	16.40×10^4	44.45×10^3	38.02×10^4	44.46×10^3	44.22×10^4
5	18321	18.39×10^3	1.96×10^{11}	67.55×10^3	18.31×10^3	15.71×10^4	18.31×10^3	18.17×10^4
10	9091	91.20×10^2	9.72×10^{10}	33.52×10^3	90.84×10^2	78.00×10^3	90.85×10^2	89.75×10^3
20	4427	44.37×10^2	4.73×10^{10}	16.32×10^3	44.22×10^2	38.00×10^3	44.23×10^2	43.05×10^3
40	2156	21.63×10^2	2.31×10^{10}	79.53×10^2	21.55×10^2	18.65×10^3	21.56×10^2	21.45×10^3

TABLE 7.10: Similarities metrics for WM UTM.

practical scenarios.

7.5.3 Watermark Imperceptibility

As it was mentioned before, the embedding process is carried out through the replacement of a *pseudo-randomly* selected word by a synonym from a specific set of synonyms. The word selected from the latter set will depend on the value of the mark extracted from the binary image. Occasionally, the word selected from the set of synonyms is the same one that was selected from the sentence. That is the scenario that best contributes to WM imperceptibility since the mark is embedded without any modification in the data. We compute the rate of marks embedded without performing word replacement through the rate of fixed words given by $T_w = \mathcal{W}_F/\mathcal{W}_T$, where $\mathcal{W}_F$ represents the words that do not change during the embedding process and $\mathcal{W}_T$, as previously defined, represents the number of tagged words. The value of T_w for each one of the experiments is shown in Table 7.9, where more or less for all cases a third of the selected words allows mark embedding without being replaced, which positively contributes to achieving WM imperceptibility.

From the knowledge source point of view, it is also possible to detect the quality of the WM imperceptibility. For this case, since we are using WordNet, we can use a set of similarity metrics that are defined to measure the relatedness or similarity between words. According to WordNet structure, and the way the metrics are defined, when two words are selected from the same synonym set, the metrics report the highest possible value between them. Table 7.10 gives the accumulated value for some metrics commonly used, for the experiments performed during mark embedding using UTM as WM source. The table's column "Iter." refers to iterations, meaning the number of times the measurement between words was carried out.

Thanks to the use of similarity metrics it is possible to determine and control the amount of distortion introduced during WM embedding. This contributes to maintaining data usability and WM imperceptibility, goals that highly depend on the knowledge source, the similarity engine and the WSD module. For our case, as long as the words belong to the same set of synonyms, quality results are guaranteed.

Even though, since our technique is meant to be used for relational data copyright protection, we have to guarantee robustness against malicious operations. The core of our approach has been proved to be resilient against common malicious operations oriented to compromise relational data watermarking techniques [4, 82, 151]. Nevertheless, we need to consider another threat more focused on compromising WM detection over text documents. The next section is oriented to analyze the resilience of our technique.

7.5.4 Technique's Robustness

The major threat our technique faces is linked to *random synonym substitution* attacks from watermarking techniques created for document protection. In a similar way that WSD and majority voting combined allow overcoming WSD lack of precision, using textual as WM cover type in relational data reduces the probability of performing a successful *random synonym substitution* attack. This is because to successfully compromise the mark value, the right words need to be selected. But first, it is required to detect the right tuple, and the attribute selected for marking inside the tuple. The high number of parameters involved in the technique makes that very difficult to achieve.

This is one of the benefits of combining both cover types. The probability of successfully overwriting marks decreases if, besides relational elements, textual's are considered. To that, once the position is correctly detected in the relation, it is necessary to know the type of word selected for the embedding (e.g., noun, adverbs, adjectives, etc.), the sentence itself, and break the secrecy of k_s (see Algorithm 7). As we mentioned in Section 7.4.4, this is ruled by the adversary model obtained as a result of extending Equation (7.5) to Equation (7.6).

To study our approach's resilience to *random updates* we performed two types of experiments, random tuple deletion and random actualization of words stored in the same attribute we use for marking. In Table 7.11 is depicted the degree of damage the WM gets while the number of pseudo-randomly deleted tuples increases. This experiment was performed attacking the relation marked with the WM generated from the image WWF with parameters TF = 2, detected with quality of SSIM = 1 with no pixels in contradiction with respect to the embedded WM. (see Table 7.7).

Datum	Percentage of tuples deleted (attack degree).									
	0	10%	20%	30%	40%	50%	60%	70%	80%	90%
Image										
SSIM	1	0.99	1	0.99	0.94	0.93	0.90	0.82	0.74	0.55
CF	100%	100%	100%	100%	100%	100%	100%	99.88%	97.10%	78.65%

TABLE 7.11: WMs detected after pseudo-random tuple deletion attacks.

The second robustness experiment was performed under the same conditions of the previous one, but updating the value of the attribute, randomly selecting the tuple according to the attack's percentage. In this case, the update operation is based on selecting the same attribute value but replacing a word from the sentence for a synonym. The selection of the tuple, the word being replaced, and the synonym was made pseudo-randomly. This experiment was performed this way to simulate random synonym substitution attacks focused on compromising textual watermarking. If the mark value during the WM detection is given by exclusion (assigning 1 if the word is the first one in the synonym set and 0 if it is not), the probability of success for this type of attack decreases considerably.

Datum	Percentage of the attributes updated (attack degree)									
	0	10%	20%	30%	40%	50%	60%	70%	80%	90%
Image										
SSIM	1	1	0.99	0.99	0.97	0.92	0.89	0.89	0.78	0.78
CF	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

TABLE 7.12: WMs detected after pseudo-random update attacks.

From the experiments performed in this section, it is clear our technique describes a resilience that guarantees WM preservation despite data degradation due to malicious operations (see Tables 7.11 and 7.12). Considering attackers are also interested in maintaining data quality, it is not expected they will exceed the degree of damage caused to the data in the experiments we performed. Then, we claim our technique is resilient, being recommended for practical scenarios to guarantee copyright protection.

7.5.5 Scalability and Complexity

Since WM embedding and extraction processes are similar in complexity, in this section we report the time required for WM embedding. We carried out a set of experiments,

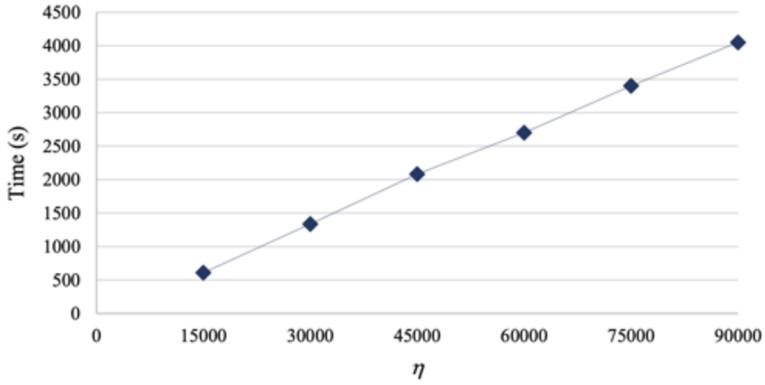


FIGURE 7.5: Correlation described by the times required for watermarking different amount of tuples.

η	**Time (s)**	
	Average	**Proportion**
15000	609.75 (±2.70)	none
30000	1336.00 (±7.35)	2.19
45000	2081.57 (±3.40)	1.55
60000	2699.07 (±7.90)	1.30
75000	3401.25 (±14.49)	1.26
90000	4050.67 (±27.41)	1.19

TABLE 7.13: Time required for WM embedding involving different tuples number.

performing WM embedding multiple times involving a different number of tuples. Figure 7.5 depicts the linear correlation between the time required by WM embedding and the number of tuples in R. This experiment was performed using $\gamma = 5$, the WM source WWF and the attribute "Text", increasing the value of η of 15000 units each time.

For each amount of tuples, the same experiment was performed several times until reaching a standard deviation of the time required as close as zero as possible. Table 7.13 shows along with the standard deviation, the mean of the time recorded. According to column "Proportion", which compares the average of the time required for marking the tuples with respect to the average of time required for the previous row, our approach describes a linear behavior.

The approach's complexity will depend on the Similarity Engine, the WSD module, and the number of sentences stored in the attribute being watermarked. Despite all these factors, for the conditions given by the experimental set up to validate our work, it is recorded a linear behavior. Then, it can be established that the overall time complexity corresponds to O(n), being reliable its application to different sizes of data stored in R.

7.6 Discussion

In this chapter, we proposed a watermarking technique for relational data that uses multi-word textual attributes as cover type. The embedding of the marks in our approach is performed by substitutions of synonym words in sentences, guaranteeing the semantic preservation of the data, and the total imperceptibility of the watermark. Despite multiple attributes can be considered for each tuple, when paragraphs are stored, the selection of one word per sentence allows the increment of the watermark capacity with respect to previous techniques, and with it, its robustness.

This technique works in combination with a WSD module, a semantic similarity engine, and one or several knowledge sources, linking its complexity and precision to the behavior of those external elements. For the experimental validation we used WordNet as knowledge source and the Lesk algorithm for WSD. The results show that how our technique guarantees the watermark embedding, its detection, and robustness against subset attacks and random synonym substitution attacks, making it a valuable tool for ownership protection, and the data integrity validation. For the case of random synonym substitution attacks, which constitute a serious threat for techniques focused on watermarking textual documents, the combination of the relational data structure and the multi-word textual data type guarantees the watermark persistence independently the attack's degree. The preservation of the semantic was defined as the main priority for this approach, adding as a feature the tolerance to the watermark embedding in a way no other technique has considered before, according to the literature published so far.

Chapter 8

CONCLUSION AND FUTURE WORK

In this thesis we studied the problem of string analysis for software verification, which is relevant also from a security point of view. Indeed, more precise string approximations allow to detect program errors that may lead to significant exploitations (voluntary or otherwise). We adopted the Abstract Interpretation theory both to design novel string approximations and to refine the existing ones.

From this thesis it emerges that the analysis of programs manipulating strings is a complex problem, far from being considered solved. One of the main reasons of its complexity lies in the fact that programming languages represent strings in different ways, as well as the operations on them. This is clearly an obstacle since it enormously increases the number of possible scenarios to be tackled, making the design of effective analyses that would not be language-specific quite difficult.

In this context, the semantic-based approach used in this thesis has led to interesting results as we have already detailed throughout this whole work. We investigated different aspects of the string analysis problem, such as the correct management of character arrays in `C` programs and the enhancement of existing string analyses. This brought us to the conclusion that semantic-based approaches can be effectively implemented and fully incorporated in the software development life-cycle. One of the advantages of the Abstract Interpretation approach is that the degree of freedom in the choice of the abstract domain allows to adequately address the trade-off between efficiency and accuracy of the analysis.

Among the countless scenarios that we could explore in depth we consider the following logical developments of this thesis, as they are an expansion of the work done so far. In fact, in the short term we aim to:

- Further enhance the effectiveness of the M-String domain by combining it by reduced product with other either numerical or symbolic domains.
- Design a tailoring method to select what domain combination better suits the string analysis of a given program based on a pre-process of the operations it applies on string and character variables.

- Investigate forward completeness [80], the property that guarantees that no loss of precision occurs during the output abstraction process of a given operation. We aim to integrate the two completeness methodologies within an industrial JavaScript static analyser in order to and to deeply evaluate their actual cost and overall impact. Moreover, we aim to define a precision measure between string abstractions.

- Investigate suitable combinations of relational string domains with non-relational ones and/or with complementary numerical domains. We also aim at optimizing the efficiency of the implementation, by taking full benefit from the results discussed in [126] for numerical domains by representing relations as multi-valued maps.

- Design a module of semantic preservation for numerical and textual data, considering the elements of watermarking techniques and approximate query processing.

Appendix A

STRING ABSTRACT DOMAINS

We formally define the String Length complete lattice introduced in Section 4.4.1 and its abstract semantics. The soundness of the String Length domain abstract semantics is also proved. Moreover, we recall the Character Inclusion (cf. Section 4.4.2) and the Prefix (cf. Section 4.4.3) complete lattices and their abstract semantics [49].

A.1 String Length

The String Length abstract domain $\overline{\mathbf{SL}}$ (cf. Section 4.4.1) is the complete lattice [11]:

$$(\overline{\mathbf{SL}}, \sqsubseteq_{\overline{\mathbf{SL}}}, \bot_{\overline{\mathbf{SL}}}, \top_{\overline{\mathbf{SL}}}, \sqcap_{\overline{\mathbf{SL}}}, \sqcup_{\overline{\mathbf{SL}}})$$

where:

- $\overline{\mathbf{SL}} \triangleq \{[\mathrm{m}, \mathrm{M}] : \mathrm{m} \in \mathbb{N} \wedge \mathrm{M} \in \mathbb{N} \cup \{\infty\} \wedge \mathrm{m} \leq \mathrm{M}\} \cup \{\bot_{\overline{\mathbf{SL}}}\}$
- Let $[\mathrm{m}_1, \mathrm{M}_1]$ and $[\mathrm{m}_2, \mathrm{M}_2]$ be two abstract values in the String Length domain then:

$$[\mathrm{m}_1, \mathrm{M}_1] \sqsubseteq_{\overline{\mathbf{SL}}} [\mathrm{m}_2, \mathrm{M}_2] \Leftrightarrow [\mathrm{m}_1, \mathrm{M}_1] = \bot_{\overline{\mathbf{SL}}} \vee (\mathrm{m}_2 \leq \mathrm{m}_1 \wedge \mathrm{M}_1 \leq \mathrm{M}_2)$$

- $\bot_{\overline{\mathbf{SL}}}$ is a special element denoting the bottom element of the considered lattice.
- $\top_{\overline{\mathbf{SL}}}$ denotes the top element of the considered lattice, and:

$$\top_{\overline{\mathbf{SL}}} = [0, \infty]$$

- $\sqcap_{\overline{\mathbf{SL}}}$ represents the meet operator, that defines the greatest lower bound between two string approximations in the String Length abstract domain, such that:

$$[m_1, M_1] \sqcap_{\overline{\mathbf{SL}}} [m_2, M_2] =$$

$$\begin{cases} [\mathsf{max}(\{m_1, m_2\}), \mathsf{min}(\{M_1, M_2\})] & \text{if } \mathsf{max}(\{m_1, m_2\}) \leq \mathsf{min}(\{M_1, M_2\}) \\ \bot_{\overline{\mathbf{SL}}} & \text{otherwise} \end{cases}$$

- $\sqcup_{\overline{\mathbf{SL}}}$ represents the join operator, that defines the least upper bound between two string approximations in the String Length abstract domain, such that:

$$[m_1, M_1] \sqcup_{\overline{\mathbf{SL}}} [m_2, M_2] = [\mathsf{min}(\{m_1, m_2\}), \mathsf{max}(\{M_1, M_2\})]$$

Abstraction

Let X be a set of concrete string values. The abstraction function on the String Length abstract domain $\alpha_{\overline{\mathbf{SL}}}$ maps X to $\bot_{\overline{\mathbf{SL}}}$ in the case in which X is equal to the empty set, otherwise to the interval $[\mathsf{min}(\mathsf{L}_X), \mathsf{max}(\mathsf{L}_X)]$, where $\mathsf{L}_X = \{|\sigma| : \sigma \in X\}$. Notice that, if X contains only one string element, i.e., if $X = \{\sigma\}$, then $\alpha_{\overline{\mathbf{SL}}}(X) = [|\sigma|, |\sigma|]$.

Concretization

The concretization function on the String Length abstract domain $\gamma_{\overline{\mathbf{SL}}}$ maps an abstract element to a set of strings as follows: $\gamma_{\overline{\mathbf{SL}}}(\bot_{\overline{\mathbf{SL}}}) = \emptyset$, while is $\{\sigma : \sigma \in \Sigma^* \wedge m \leq |\sigma| \leq M\}$, i.e., the set of all the possible strings whose length goes from m to M.

Galois connection

We follow Theorem 1.1 of [49] and we prove that $(\alpha_{\overline{\mathbf{SL}}}, \gamma_{\overline{\mathbf{SL}}})$ is a Galois Connection.

Theorem A.1. Let the abstraction function $\alpha_{\overline{\mathbf{SL}}}$ be defined as:

$$\alpha_{\overline{\mathbf{SL}}} = \lambda X. \sqcap_{\overline{\mathbf{SL}}} \{[m, M] : X \subseteq \gamma_{\overline{\mathbf{SL}}}([m, M])\}$$

Then, $(\mathcal{P}(\Sigma^*), \subseteq) \underset{\alpha_{\overline{\mathbf{SL}}}}{\overset{\gamma_{\overline{\mathbf{SL}}}}{\leftrightarrows}} (\overline{\mathbf{SL}}, \sqsubseteq_{\overline{\mathbf{SL}}})$.

Proof.
By Theorem 1.1 of [49], we only need to prove that $\gamma_{\overline{\mathbf{SL}}}$ is a complete meet morphism. Formally, we have to prove that:

$$\gamma_{\overline{\mathbf{SL}}}\left(\bigsqcap_{\overline{\mathbf{SL}}}{}_{[m,M] \in \overline{X}} [m, M]\right) = \bigcap_{[m,M] \in \overline{X}} \gamma_{\overline{\mathbf{SL}}}([m, M])$$

$\gamma_{\overline{\mathbf{SL}}}\left(\bigsqcap_{\overline{\mathbf{SL}}}{}_{[m,M] \in \overline{X}} [m, M]\right)$

by definition of $\sqcap_{\overline{\mathbf{SL}}}$

$= \gamma_{\overline{\mathbf{SL}}}([m', M'])$ where $m' = \mathsf{max}_{[m,M] \in \overline{X}}\ m$ and $M' = \mathsf{min}_{[m,M] \in \overline{X}}\ M$

$$
\begin{array}{rcl}
\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{new String}(\sigma)]\!]() & = & [|\sigma|, |\sigma|] \\
\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{concat}]\!]([m_1, M_1], [m_2, M_2]) & = & [m_1 + m_2, M_1 + M_2] \\
\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{substring}_{\texttt{b}}^{\texttt{e}}]\!]([m, M]) & = & \begin{cases} [n, n] & \text{if } n \in [m, M] \\ \bot_{\overline{\mathbf{SL}}} & \text{otherwise} \end{cases} \\
 & & \text{where } n = (\texttt{e} - \texttt{b}) + 1 \\
\mathfrak{B}_{\overline{\mathbf{SL}}}[\![\texttt{contains}_{\texttt{c}}]\!]([m, M]) & = & \top_{\mathbf{B}}
\end{array}
$$

TABLE A.1: $\overline{\mathbf{SL}}$ abstract semantics

by definition of $\gamma_{\overline{\mathbf{SL}}}$

$= \{\sigma \mid \sigma \in \Sigma^* \wedge m' \leq |\sigma| \leq M'\}$

by definition of $\cap$

$= \bigcap_{[m,M] \in \overline{X}} \{\sigma : \sigma \in \Sigma^* \wedge m \leq |\sigma| \leq M\}$

by definition of $\gamma_{\overline{\mathbf{SL}}}$

$= \bigcap_{[m,M] \in \overline{X}} \gamma_{\overline{\mathbf{SL}}}([m, M])$

□

Widening operator

The String Length abstract domain has infinite height and does not respect the Ascending Chain Condition, therefore it needs to be equipped with a widening operator. The widening $\nabla_{\overline{\mathbf{SL}}}$ for the string length analysis is as follows:

$$
\begin{aligned}
[m_1, M_1] \nabla_{\overline{\mathbf{SL}}} [m_2, M_2] = [&\text{if } m_2 < m_1 \text{ then } 0 \text{ else } m_1, \\
&\text{if } M_2 > M_1 \text{ then } \infty \text{ else } M_1]
\end{aligned}
$$

Abstract semantics

We define the abstract semantics of the String Length domain and we prove its soundness. The semantics $\mathfrak{S}_{\overline{\mathbf{SL}}}$ and $\mathfrak{B}_{\overline{\mathbf{SL}}}$ are the abstract counterparts of $\mathfrak{S}$ and $\mathfrak{B}$ (cf. Section **??**).

Table A.1 formalizes the abstract semantics of $\overline{\mathbf{SL}}$. Precisely,

- $\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{new String}(\sigma)]\!]() = [|\sigma|, |\sigma|]$

  Let σ be a sequence of characters.

  The semantics $\mathfrak{S}_{\overline{\mathbf{SL}}}$, when applied to `new String(σ)`, returns the interval $[|\sigma|, |\sigma|]$, as when a new string constant is evaluated both its minimum and its maximum length coincide and are equal to the length of σ.

- $\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{concat}]\!]([m_1, M_1], [m_2, M_2]) = [m_1 + m_2, M_1 + M_2]$

Let $[m_1, M_1], [m_2, M_2] \in \overline{\mathbf{SL}}$.

The semantics $\mathfrak{S}_{\overline{\mathbf{SL}}}$, when applied to $\texttt{concat}([m_1, M_1], [m_2, M_2])$, returns the sum of the input intervals, i.e., $[m_1, M_1] + [m_2, M_2] = [m_1 + m_2, M_1 + M_2]$, as the length of a string resulting from the concatenation between two other strings is the sum of the lengths of the latter, i.e., $|\sigma_1\sigma_2| = |\sigma_1| + |\sigma_2|$.

- $\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{substring}_{\texttt{b}}^{\texttt{e}}]\!]([m, M]) = \begin{cases} [n, n] & \text{if } n \in [m, M] \\ \bot_{\overline{\mathbf{SL}}} & \text{otherwise} \end{cases}$

  Let $[m, M] \in \overline{\mathbf{SL}}$ and let n be equal to $(\texttt{e} - \texttt{b}) + 1$.

  The semantics $\mathfrak{S}_{\overline{\mathbf{SL}}}$, when applied to $\texttt{substring}_{\texttt{b}}^{\texttt{e}}([m, M])$, returns the interval $[n, n]$ if n belongs to the input interval $[m, M]$, otherwise it returns $\bot_{\overline{\mathbf{SL}}}$, as the substring of a string from the b-th to the e-th index has length equal to n.

- $\mathfrak{B}_{\overline{\mathbf{SL}}}[\![\texttt{contains}_{\texttt{c}}]\!]([m, M]) = \top_{\mathbf{B}}$

  Let $[m, M] \in \overline{\mathbf{SL}}$.

  The semantics $\mathfrak{B}_{\overline{\mathbf{SL}}}$ applied to $\texttt{contains}_{\texttt{c}}([m, M])$ returns $\top_{\mathbf{B}}$, as we can not infer any information about the content of the strings approximated by $[m, M]$.

Theorem A.2. $\mathfrak{S}_{\overline{\mathbf{SL}}}$ and $\mathfrak{B}_{\overline{\mathbf{SL}}}$ are sound over-approximations of $\mathfrak{S}$ and $\mathfrak{B}$ respectively. Formally,

$$\gamma_{\overline{\mathbf{SL}}}(\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\text{Stm}]\!]([m, M])) \supseteq \{\mathfrak{S}[\![\text{Stm}]\!](\sigma) \mid \sigma \in \gamma_{\overline{\mathbf{SL}}}([m, M])\}$$
$$\gamma_{\overline{\mathbf{SL}}}(\mathfrak{B}_{\overline{\mathbf{SL}}}[\![\text{Stm}]\!]([m, M])) \supseteq \{\mathfrak{B}[\![\text{Stm}]\!](\sigma) \mid \sigma \in \gamma_{\overline{\mathbf{SL}}}([m, M])\}$$

Proof.
We prove the soundness separately for each operator.

- Consider the `new String` operator and let σ be a sequence of characters. We have to prove that,

  $$\gamma_{\overline{\mathbf{SL}}}(\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{new String}(\sigma)]\!]()) \supseteq \{\mathfrak{S}[\![\texttt{new String}(\sigma)]\!]()\}$$

  This holds from the definitions of $\mathfrak{S}_{\overline{\mathbf{SL}}}$ and of $\gamma_{\overline{\mathbf{SL}}}$.

- Consider the `concat` operation and let $[m_1, M_1], [m_2, M_2] \in \overline{\mathbf{SL}}$. We have to prove that,

  $$\gamma_{\overline{\mathbf{SL}}}(\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{concat}]\!]([m_1, M_1], [m_2, M_2]))$$
  $$\supseteq$$
  $$\{\mathfrak{S}[\![\texttt{concat}]\!](\sigma_1, \sigma_2) \mid \sigma_1 \in \gamma_{\overline{\mathbf{SL}}}([m_1, M_1]) \wedge \sigma_2 \in \gamma_{\overline{\mathbf{SL}}}([m_2, M_2])\}$$

Let $\sigma_1\sigma_2$ be a generic element in $\{\mathfrak{S}[\![\texttt{concat}]\!](\sigma_1, \sigma_2) : \sigma_1 \in \gamma_{\overline{\mathbf{SL}}}([m_1, M_1]) \wedge \sigma_2 \in \gamma_{\overline{\mathbf{SL}}}([m_2, M_2])\}$. As σ_1 is any string whose length goes from m_1 to M_1 and σ_2 is any string whose length goes from m_2 to M_2, the concatenation of σ_1 and σ_2 is any string whose length goes from $m_1 + m_2$ to $M_1 + M_2$. Therefore $\sigma_1\sigma_2$ belongs to $\gamma_{\overline{\mathbf{SL}}}(\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{concat}]\!]([m_1, M_1], [m_2, M_2]))$, because $\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{concat}]\!]([m_1, M_1], [m_2, M_2]) = [m_1 + m_2, M_1 + M_2]$, by definition of $\mathfrak{S}_{\overline{\mathbf{SL}}}$. Indeed, $\gamma_{\overline{\mathbf{SL}}}([m_1 + m_2, M_1 + M_2])$ contains all the strings whose length goes from $m_1 + m_2$ to $M_1 + M_2$, by definition of $\gamma_{\overline{\mathbf{SL}}}$.

- Consider the `substring` operation and let $[m, M] \in \overline{\mathbf{SL}}$. we have to prove that,

$$\gamma_{\overline{\mathbf{SL}}}(\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{substring}_{\mathtt{b}}^{\mathtt{e}}]\!]([m, M])) \supseteq \{\mathfrak{S}[\![\texttt{substring}_{\mathtt{b}}^{\mathtt{e}}]\!](\sigma) \mid \sigma \in \gamma_{\overline{\mathbf{SL}}}([m, M])\}$$

  Let $\sigma_{\mathtt{b}}...\sigma_{\mathtt{e}}$ be a generic element in $\left\{ \mathfrak{S}[\![\texttt{substring}_{\mathtt{b}}^{\mathtt{e}}]\!](\sigma) \right\} \sigma \in \gamma_{\overline{\mathbf{SL}}}([m, M])$. As σ is any string whose length goes from m to M, its substring from the b-th to the e-th index, if it exists, i.e., if $n \leq |\sigma|$, has length equal to n. Therefore $\sigma_{\mathtt{b}}...\sigma_{\mathtt{e}}$ belongs to $\gamma_{\overline{\mathbf{SL}}}(\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{substring}_{\mathtt{b}}^{\mathtt{e}}]\!]([m, M]))$, because $\mathfrak{S}_{\overline{\mathbf{SL}}}[\![\texttt{substring}_{\mathtt{b}}^{\mathtt{e}}]\!]([m, M]) = [n, n]$ if $n \in [m, M]$, by definition of $\mathfrak{S}_{\overline{\mathbf{SL}}}$. Indeed, $\gamma_{\overline{\mathbf{SL}}}([n, n])$ contains all the strings of length n and $\gamma_{\overline{\mathbf{SL}}}(\bot_{\overline{\mathbf{SL}}})$ is the emptyset, by definition of $\gamma_{\overline{\mathbf{SL}}}$.

- Consider the `contains` operator and let $[m, M] \in \overline{\mathbf{SL}}$. We have to prove that,

$$\gamma_{\overline{\mathbf{SL}}}(\mathfrak{B}_{\overline{\mathbf{SL}}}[\![\texttt{contains}_{\mathtt{c}}]\!]([m, M])) \supseteq \{\mathfrak{B}[\![\texttt{contains}_{\mathtt{c}}]\!](\sigma) \mid \sigma \in \gamma_{\overline{\mathbf{SL}}}([m, M])\}$$

  This holds as the abstract semantics returns the top element $\top_{\mathbf{B}}$ that approximates any possible result of the concrete semantics.

□

A.2 Character Inclusion

The Character Inclusion abstract domain $\overline{\mathbf{CI}}$ is the complete lattice [49]:

$$(\overline{\mathbf{CI}}, \sqsubseteq_{\overline{\mathbf{CI}}}, \bot_{\overline{\mathbf{CI}}}, \top_{\overline{\mathbf{CI}}}, \sqcap_{\overline{\mathbf{CI}}}, \sqcup_{\overline{\mathbf{CI}}})$$

where:

- $\overline{\mathbf{CI}} \triangleq \{(C, MC) : C, MC \in \mathcal{P}(\Sigma) \wedge C \subseteq MC\} \cup \{\bot_{\overline{\mathbf{CI}}}\}$

  Where Σ denotes the alphabet.

- Let (C_1, MC_1) and (C_2, MC_2) be two abstract values in the Character Inclusion domain then:

$$(C_1, MC_1) \sqsubseteq_{\overline{\mathbf{CI}}} (C_2, MC_2) \Leftrightarrow (C_1, MC_1) = \bot_{\overline{\mathbf{CI}}} \vee (C_1 \supseteq C_2 \wedge MC_1 \subseteq MC_2)$$

- $\bot_{\overline{\mathbf{CI}}}$ denotes the bottom element of the considered lattice, and:

$$\bot_{\overline{\mathbf{CI}}} = \{(\mathrm{C}, \mathrm{MC}) \mid \mathrm{C} \nsubseteq \mathrm{MC}\}$$

- $\top_{\overline{\mathbf{CI}}}$ denotes the top element of the considered lattice, and:

$$\top_{\overline{\mathbf{CI}}} = (\emptyset, \Sigma)$$

- $\sqcap_{\overline{\mathbf{CI}}}$ represents the meet operator, that defines the greatest lower bound between two string approximations in the Character Inclusion abstract domain, such that:

$$\sqcap_{\overline{\mathbf{CI}}}((\mathrm{C}_1, \mathrm{MC}_1), (\mathrm{C}_2, \mathrm{MC}_2)) =$$

$$\begin{cases} (\mathrm{C}_1 \cup \mathrm{C}_2, \mathrm{MC}_1 \cap \mathrm{MC}_2) & \text{if } \mathrm{C}_1 \subseteq \mathrm{MC}_2 \text{ and } \mathrm{C}_2 \subseteq \mathrm{MC}_1 \\ \bot_{\overline{\mathbf{CI}}} & \text{otherwise} \end{cases}$$

- $\sqcup_{\overline{\mathbf{CI}}}$ represents the join operator, that defines the least upper bound between two string approximations in the Character Inclusion abstract domain, such that:

$$\sqcup_{\overline{\mathbf{CI}}}((\mathrm{C}_1, \mathrm{MC}_1), (\mathrm{C}_2, \mathrm{MC}_2)) = (\mathrm{C}_1 \cap \mathrm{C}_2, \mathrm{MC}_1 \cup \mathrm{MC}_2).$$

Abstraction

Let X be a set of concrete string values. The abstraction function on the Character Inclusion abstract domain $\alpha_{\overline{\mathbf{CI}}}$ maps X to $\bot_{\overline{\mathbf{CI}}}$ in the case in which X is equal to the empty set, otherwise to the pair of sets $(\bigcap \mathsf{H}_\mathrm{X}, \bigcup \mathsf{H}_\mathrm{X})$ where $\mathsf{H}_\mathrm{X} = \{\mathsf{char}(\sigma) : \sigma \in \mathrm{X}\}$. Notice that, if X contains just one string element, then $\alpha_{\overline{\mathbf{CI}}}(\mathrm{X}) = (\mathsf{H}_\mathrm{X}, \mathsf{H}_\mathrm{X})$.

Concretization

The concretization function on the Character Inclusion abstract domain $\gamma_{\overline{\mathbf{CI}}}$ maps an abstract element to a set of strings as follows: $\gamma_{\overline{\mathbf{CI}}}(\bot_{\overline{\mathbf{CI}}}) = \emptyset$, while is $\{\sigma : \sigma \in \Sigma^*,\ \mathrm{C} \subseteq \mathsf{char}(\sigma) \subseteq \mathrm{MC}\}$, i.e., the set of all the possible strings certainly containing the elements in C and possibly containing the elements in MC.

Galois connection

The functions $\gamma_{\overline{\mathbf{CI}}}$ and $\alpha_{\overline{\mathbf{CI}}}$ form a Galois Connection (Theorem 4.2 of [49]).

Widening operator

The Character Inclusion abstract domain has finite height, therefore its widening operator coincides with its least upper bound operator.

$$
\begin{aligned}
\mathfrak{S}_{\overline{\mathbf{CI}}}[\![\texttt{new String}(\sigma)]\!]() &= (\mathsf{char}(\sigma), \mathsf{char}(\sigma)) \\
\mathfrak{S}_{\overline{\mathbf{CI}}}[\![\texttt{concat}]\!]((\mathrm{C}_1, \mathrm{MC}_1), (\mathrm{C}_2, \mathrm{MC}_2)) &= (\mathrm{C}_1 \cup \mathrm{C}_2, \mathrm{MC}_1 \cup \mathrm{MC}_2) \\
\mathfrak{S}_{\overline{\mathbf{CI}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!]((\mathrm{C}, \mathrm{MC})) &= (\emptyset, \mathrm{MC}) \\
\mathfrak{B}_{\overline{\mathbf{CI}}}[\![\texttt{contains}_{\mathsf{c}}]\!]((\mathrm{C}, \mathrm{MC})) &= \begin{cases} \texttt{true} & \text{if } \mathsf{c} \in \mathrm{C} \\ \texttt{false} & \text{if } \mathsf{c} \notin \mathrm{MC} \\ \top_{\mathbf{B}} & \text{otherwise} \end{cases}
\end{aligned}
$$

TABLE A.2: $\overline{\mathbf{CI}}$ abstract semantics

Abstract semantics

We recall the Character Inclusion domain abstract semantics. Table A.2 formalizes the abstract semantics of $\overline{\mathbf{CI}}$. Precisely,

- $\mathfrak{S}_{\overline{\mathbf{CI}}}[\![\texttt{new String}(\sigma)]\!]() = (\mathsf{char}(\sigma), \mathsf{char}(\sigma))$

  Let σ be a sequence of characters.

  The semantics of $\mathfrak{S}_{\overline{\mathbf{CI}}}$, when applied to $\texttt{new String}(\sigma)$ returns the pair $(\mathsf{char}(\sigma), \mathsf{char}(\sigma))$, as when a new constant string is evaluated both the sets of characters it certainly and probably contains are those appearing in σ.

- $\mathfrak{S}_{\overline{\mathbf{CI}}}[\![\texttt{concat}]\!]((\mathrm{C}_1, \mathrm{MC}_1), (\mathrm{C}_2, \mathrm{MC}_2)) = (\mathrm{C}_1 \cup \mathrm{C}_2, \mathrm{MC}_1 \cup \mathrm{MC}_2)$

  Let $(\mathrm{C}_1, \mathrm{MC}_1), (\mathrm{C}_2, \mathrm{MC}_2) \in \overline{\mathbf{CI}}$.

  The semantics $\mathfrak{S}_{\overline{\mathbf{CI}}}$, when applied to $\texttt{concat}((\mathrm{C}_1, \mathrm{MC}_1), (\mathrm{C}_2, \mathrm{MC}_2))$, returns the pair $(\mathrm{C}_1 \cup \mathrm{C}_2, \mathrm{MC}_1 \cup \mathrm{MC}_2)$, as the characters which occur in a string resulting from the concatenation between two other strings are those appearing in the first string plus those appearing in the second string, i.e., $\mathsf{char}(\sigma_1\sigma_2) = \mathsf{char}(\sigma_1) \cup \mathsf{char}(\sigma_2)$.

- $\mathfrak{S}_{\overline{\mathbf{CI}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!]((\mathrm{C}, \mathrm{MC})) = (\emptyset, \mathrm{MC})$

  Let $(\mathrm{C}, \mathrm{MC}) \in \overline{\mathbf{CI}}$.

  The semantics $\mathfrak{S}_{\overline{\mathbf{CI}}}$, when applied to $\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}((\mathrm{C}, \mathrm{MC}))$ returns the pair $(\emptyset, \mathrm{MC})$, as we do not have any information about the position of the characters in C.

- $\mathfrak{B}_{\overline{\mathbf{CI}}}[\![\texttt{contains}_{\mathsf{c}}]\!]((\mathrm{C}, \mathrm{MC})) = \begin{cases} \texttt{true} & \text{if } \mathsf{c} \in \mathrm{C} \\ \texttt{false} & \text{if } \mathsf{c} \notin \mathrm{MC} \\ \top_{\mathbf{B}} & \text{otherwise} \end{cases}$

  Let $(\mathrm{C}, \mathrm{MC}) \in \overline{\mathbf{CI}}$.

  The semantics $\mathfrak{B}_{\overline{\mathbf{CI}}}$ applied to $\texttt{substring}_{\mathsf{c}}((\mathrm{C}, \mathrm{MC}))$ returns `true` if the set C contains the character `c`, `false` if the set MC does not contain the character `c`, otherwise $\top_{\mathbf{B}}$, as the characters which occur in C are those certainly appearing in all the strings approximated by $(\mathrm{C}, \mathrm{MC})$.

The Character Inclusion domain abstract semantics is a sound over-approximation of the string domain concrete semantics (Theorem 4.3 of [49]).

A.3 Prefix

A prefix is a sequence of characters followed by an asterisk $*$. Since the asterisk at the end of the literal part is always present, it is not included in the domain. The Prefix abstract domain $\overline{\mathbf{PR}}$ is the complete lattice [49]:

$$(\overline{\mathbf{PR}}, \sqsubseteq_{\overline{\mathbf{PR}}}, \bot_{\overline{\mathbf{PR}}}, \top_{\overline{\mathbf{PR}}}, \sqcap_{\overline{\mathbf{PR}}}, \sqcup_{\overline{\mathbf{PR}}})$$

where:

- $\overline{\mathbf{PR}} \triangleq \Sigma^* \cup \{\bot_{\overline{\mathbf{PR}}}\}$

- Let $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ be two abstract values in the Prefix domain then:

  $$\overline{\mathsf{p}}_1 \sqsubseteq_{\overline{\mathbf{PR}}} \overline{\mathsf{p}}_2 \Leftrightarrow \overline{\mathsf{p}}_1 = \bot_{\overline{\mathbf{PR}}} \vee (\mathsf{len}(\overline{\mathsf{p}}_2) \leq \mathsf{len}(\overline{\mathsf{p}}_1) \wedge (\forall \mathsf{i} \in [0, \mathsf{len}(\overline{\mathsf{p}}_2) - 2] : \overline{\mathsf{p}}_2[\mathsf{i}] = \overline{\mathsf{p}}_1[\mathsf{i}]))$$

  Notice that $\mathsf{len}(\overline{\mathsf{p}})$ corresponds to the number of characters spelt out in the prefix.

- $\bot_{\overline{\mathbf{PR}}}$ is a special element denoting the bottom element of the considered lattice.

- $\top_{\overline{\mathbf{PR}}}$ denotes the top element of the considered lattice and it corresponds to the empty prefix value, such that:

  $$\top_{\overline{\mathbf{PR}}} = *$$

- $\sqcap_{\overline{\mathbf{PR}}}$ represents the meet operator, that defines the greatest lower bound between two string approximations in the Prefix abstract domain, such that:

  $$\sqcap_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2) = \begin{cases} \overline{\mathsf{p}}_1 & \text{if } \overline{\mathsf{p}}_1 \sqsubseteq_{\overline{\mathbf{PR}}} \overline{\mathsf{p}}_2 \\ \overline{\mathsf{p}}_2 & \text{if } \overline{\mathsf{p}}_2 \sqsubseteq_{\overline{\mathbf{PR}}} \overline{\mathsf{p}}_1 \\ \bot_{\overline{\mathbf{PR}}} & \text{otherwise} \end{cases}$$

- $\sqcup_{\overline{\mathbf{PR}}}$ represents the join operator, that defines the least upper bound between two string approximations in the Prefix abstract domain, such that:

  $$\sqcup_{\overline{\mathbf{PR}}}(\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2) = \begin{cases} \top_{\overline{\mathbf{PR}}} & \text{if } \overline{\mathsf{p}}_1[0] \neq \overline{\mathsf{p}}_2[0] \\ \mathsf{lcp}(\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2) & \text{otherwise} \end{cases}$$

  where $\mathsf{lcp}(\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2)$ denotes the longest common prefix of $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$.

$$
\begin{array}{rcl}
\mathfrak{S}_{\overline{\mathbf{PR}}}[\![\texttt{new String}(\sigma)]\!]() & = & \sigma \\
\mathfrak{S}_{\overline{\mathbf{PR}}}[\![\texttt{concat}]\!](\overline{p}_1, \overline{p}_2) & = & \overline{p}_1 \\
\mathfrak{S}_{\overline{\mathbf{PR}}}[\![\texttt{substring}_b^e]\!](\overline{p}) & = & \begin{cases} \overline{p}[b...e] & \text{if } e < \mathsf{len}(\overline{p}) \\ \overline{p}[b...(\mathsf{len}(\overline{p}) - 1)] & \text{if } e \geq \mathsf{len}(\overline{p}) \wedge b < \mathsf{len}(\overline{p}) \\ \top_{\overline{\mathbf{PR}}} & \text{otherwise} \end{cases} \\
\mathfrak{B}_{\overline{\mathbf{PR}}}[\![\texttt{contains}_c]\!](\overline{p}) & = & \begin{cases} \texttt{true} & \text{if } c \in \mathsf{char}(\overline{p}) \\ \top_{\mathbf{B}} & \text{otherwise} \end{cases}
\end{array}
$$

TABLE A.3: $\overline{\mathbf{PR}}$ abstract semantics

Abstraction

Let X be a set of concrete string values. The abstraction function on the Prefix abstract domain $\alpha_{\overline{\mathbf{PR}}}$ maps X to $\bot_{\overline{\mathbf{PR}}}$ in the case in which X is equal to the empty set, otherwise to the longest common prefix of all the strings in X (which is followed by $*$). Notice that, if X contains just one string concrete value, its abstract value will be equal to any possible sequence of characters having that string as prefix.

Concretization

The concretization function on the Prefix abstract domain $\gamma_{\overline{\mathbf{PR}}}$ maps an abstract element to a set of strings as follows: $\gamma_{\overline{\mathbf{PR}}}(\bot_{\overline{\mathbf{PR}}}) = \emptyset$, while is $\{\sigma : \sigma \in \Sigma^*,\ \mathsf{len}(\sigma) \geq \mathsf{len}(\overline{p}) \wedge \forall i \in [0, \mathsf{len}(\overline{p}) - 1] : \sigma[i] = \overline{p}[i]\}$, i.e., the set of all the possible sequences of characters sharing the same prefix.

Galois connection

The functions $\gamma_{\overline{\mathbf{PR}}}$ and $\alpha_{\overline{\mathbf{PR}}}$ form a Galois Connection (Theorem 4.7 of [49]).

Widening operator

The Prefix abstract domain has infinite height but it respects the Ascending Chain Condition, therefore there is no need to define a widening operator to ensure the convergence of the analysis.

Abstract semantics

We recall the Prefix domain abstract semantics.

Table A.3 formalizes the abstract semantics of $\overline{\mathbf{PR}}$. Precisely,

- $\mathfrak{S}_{\overline{\mathbf{PR}}}[\![\texttt{new String}(\sigma)]\!]() = \sigma$

  Let σ be a sequence of characters.

The semantics $\mathfrak{S}_{\overline{\mathbf{PR}}}$, when applied to `new String(σ)`, returns the σ, as when a new string constant is evaluated its most precise prefix corresponds to the string itself.

- $\mathfrak{S}_{\overline{\mathbf{PR}}}[\![\texttt{concat}]\!](\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2) = \overline{\mathsf{p}}_1$

  Let $\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2 \in \overline{\mathbf{PR}}$.

  The semantics $\mathfrak{S}_{\overline{\mathbf{PR}}}$, when applied to $\texttt{concat}(\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2)$, returns $\overline{\mathsf{p}}_1$, as a string resulting from the concatenation of two other strings will always begin with the first string involved in the concatenation.

- $$\mathfrak{S}_{\overline{\mathbf{PR}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!](\overline{\mathsf{p}}) = \begin{cases} \overline{\mathsf{p}}[\mathsf{b}...\mathsf{e}] & \text{if } \mathsf{e} < \mathsf{len}(\overline{\mathsf{p}}) \\ \overline{\mathsf{p}}[\mathsf{b}...(\mathsf{len}(\overline{\mathsf{p}}) - 1)] & \text{if } \mathsf{e} \geq \mathsf{len}(\overline{\mathsf{p}}) \wedge \mathsf{b} < \mathsf{len}(\overline{\mathsf{p}}) \\ \top_{\overline{\mathbf{PR}}} & \text{otherwise} \end{cases}$$

  Let $\overline{\mathsf{p}} \in \overline{\mathbf{PR}}$.

  The semantics $\mathfrak{S}_{\overline{\mathbf{PR}}}$, when applied to $\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}(\overline{\mathsf{p}})$, returns the subprefix from the index b to the index e if $\mathsf{e} < \mathsf{len}(\overline{\mathsf{p}})$, as the substring is completely contained in $\overline{\mathsf{p}}$; the subprefix from the index b to the index $\mathsf{len}(\overline{\mathsf{p}})$ minus 1 if $\mathsf{e} \geq \mathsf{len}(\overline{\mathsf{p}})$ and $\mathsf{b} < \mathsf{len}(\overline{\mathsf{p}})$, as the substring is not entirely contained in $\overline{\mathsf{p}}$; otherwise it returns $\top_{\overline{\mathbf{PR}}}$.

- $$\mathfrak{B}_{\overline{\mathbf{PR}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\overline{\mathsf{p}}) = \begin{cases} \texttt{true} & \text{if } \mathsf{c} \in \text{char}(\overline{\mathsf{p}}) \\ \top_{\mathbf{B}} & \text{otherwise} \end{cases}$$

  Let $\overline{\mathsf{p}} \in \overline{\mathbf{PR}}$.

  The semantics $\mathfrak{B}_{\overline{\mathbf{PR}}}$ applied to $\texttt{contains}_{\mathsf{c}}(\overline{\mathsf{p}})$ returns `true` if the prefix contain the character c, otherwise $\top_{\mathbf{B}}$, as we can not infer any information about the content of the strings approximated by $\overline{\mathsf{p}}$ after the prefix itself.

The Prefix domain abstract semantics is a sound over-approximation of the string domain concrete semantics (Theorem 4.5 of [49]).

Appendix B

RELATIONAL STRING ABSTRACT DOMAINS

B.1 Abstract Semantics of Len

Let $[\![\texttt{st}]\!]^{\mathsf{len}} : \mathsf{Len} \to \mathsf{Len}$ be the function that given an input abstract memory returns an abstract memory containing the new string length relations introduced by `st`.

- **Assignment abstract semantics**

  Consider the assignment $\mathrm{x} = \mathtt{s}$ such that $\mathrm{x} \in \mathrm{X}_{\mathrm{str}}$ and $\mathtt{s} \in \mathrm{SE}$. Formally, its abstract semantics, i.e., $[\![\mathrm{x} = \mathtt{s}]\!]^{\mathsf{len}}\mathcal{L}$, is given by the following steps:

  - **[remove]**

    $$\mathcal{L}_{\mathrm{r}} = \begin{cases} \mathcal{L} \setminus \{\mathrm{w} \preceq_{\mathsf{len}} \mathrm{z} \mid \mathrm{w} = \mathrm{x}\} & \text{if x appears at the top-level of } \mathtt{s} \\ \mathcal{L} \setminus \{\mathrm{w} \preceq_{\mathsf{len}} \mathrm{z} \mid \mathrm{w} = \mathrm{x} \vee \mathrm{z} = \mathrm{x}\} & \text{otherwise} \end{cases}$$

  - **[add]**

    $$\mathcal{L}_{\mathrm{a}} = \mathcal{L}_{\mathrm{r}} \cup \{\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\mathtt{s})\}$$

    where extr is the extraction function defined in Section 6.3.2.

  - **[closure]**

    $$[\![\mathrm{x} = \mathtt{s}]\!]^{\mathsf{len}}\mathcal{L} = \mathsf{Clos}(\mathcal{L}_{\mathrm{a}})$$

- **Boolean expressions abstract semantics**

  - *contains abstract semantics*

    $$[\![\mathtt{contains}(\mathrm{x}, \mathtt{s})]\!]^{\mathsf{len}}\mathcal{L} = \mathsf{Clos}(\mathcal{L} \cup \{\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\mathtt{s})\})$$

- *equality abstract semantics*

$$[\![\mathrm{x} \texttt{ == } \texttt{s}]\!]^{\mathsf{len}}\mathcal{L} = \begin{cases} \mathsf{Clos}(\mathcal{L} \cup \{\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x}, \mathrm{x} \preceq_{\mathsf{len}} \mathrm{y}\}) & \text{if } \texttt{s} = \mathrm{y} \in \mathrm{X}_{\mathrm{str}} \\ \mathsf{Clos}(\mathcal{L} \cup \{\mathrm{y} \preceq_{\mathsf{len}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\texttt{s})\}) & \text{otherwise} \end{cases}$$

The same applies for $[\![\texttt{s} \texttt{ == } \mathrm{x}]\!]^{\mathsf{len}}\mathcal{L}$.

- *conjunction abstract semantics*

$$[\![\mathsf{e}_1 \texttt{ \&\& } \mathsf{e}_2]\!]^{\mathsf{len}}\mathcal{L} = [\![\mathsf{e}_1]\!]^{\mathsf{len}}\mathcal{L} \sqcup_{\mathsf{len}} [\![\mathsf{e}_2]\!]^{\mathsf{len}}\mathcal{L}$$

- *disjunction abstract semantics*

$$[\![\mathsf{e}_1 \texttt{ || } \mathsf{e}_2]\!]^{\mathsf{len}}\mathcal{L} = [\![\mathsf{e}_1]\!]^{\mathsf{len}}\mathcal{L} \sqcap_{\mathsf{len}} [\![\mathsf{e}_2]\!]^{\mathsf{len}}\mathcal{L}$$

Theorem B.1. The abstract semantics of Len is sound. Indeed, it holds that:

$$\forall \mathsf{M} \in \mathcal{P}(\mathbb{M}).\ \alpha_{\mathsf{len}}([\![\mathrm{x} = \texttt{s}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{len}} ([\![\mathrm{x} = \texttt{s}]\!]^{\mathsf{len}}\alpha_{\mathsf{len}}(\mathsf{M}))$$

and

$$\forall \mathsf{M} \in \mathcal{P}(\mathbb{M}).\ \alpha_{\mathsf{len}}([\![\mathsf{b}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{len}} ([\![\mathsf{b}]\!]^{\mathsf{len}}\alpha_{\mathsf{len}}(\mathsf{M}))$$

Proof.
The proof is analogous to the one provided for Theorem B.3.

□

B.2 Abstract Semantics of Char

Let $[\![\texttt{st}]\!]^{\mathsf{char}} : \mathsf{Char} \to \mathsf{Char}$ be the function that given an input abstract memory returns an abstract memory containing the new character inclusion relations introduced by `st`.

- **Assignment abstract semantics**

  Consider the assignment $\mathrm{x} = \texttt{s}$ such that $\mathrm{x} \in \mathrm{X}_{\mathrm{str}}$ and $\texttt{s} \in \mathrm{SE}$. Formally, its abstract semantics, i.e., $[\![\mathrm{x} = \texttt{s}]\!]^{\mathsf{char}}\mathcal{C}$ is given by the following steps:

  - **[remove]**

$$\mathcal{C}_{\mathrm{r}} = \begin{cases} \mathcal{C} \setminus \{\mathrm{w} \preceq_{\mathsf{char}} \mathrm{z} \mid \mathrm{w} = \mathrm{x}\} & \text{if x appears at the top-level of } \texttt{s} \\ \mathcal{C} \setminus \{\mathrm{w} \preceq_{\mathsf{char}} \mathrm{z} \mid \mathrm{w} = \mathrm{x} \vee \mathrm{z} = \mathrm{x}\} & \text{otherwise} \end{cases}$$

  - **[add]**

$$\mathcal{C}_{\mathrm{a}} = \mathcal{C}_{\mathrm{r}} \cup \{\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\texttt{s})\}$$

where extr is the extraction function defined in Section 6.3.2.

- **[closure]**

$$[\![\mathrm{x} = \mathtt{s}]\!]^{\mathsf{char}}\mathcal{C} = \mathsf{Clos}(\mathcal{C}_{\mathrm{a}})$$

- **Boolean expressions abstract semantics**

  - *contains abstract semantics*

$$[\![\mathtt{contains}(\mathrm{x}, \mathtt{s})]\!]^{\mathsf{char}}\mathcal{C} = \mathsf{Clos}(\mathcal{C} \cup \{\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\mathtt{s})\})$$

  - *equality abstract semantics*

$$[\![\mathrm{x} \texttt{ == } \mathtt{s}]\!]^{\mathsf{char}}\mathcal{C} = \begin{cases} \mathsf{Clos}(\mathcal{C} \cup \{\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x}, \mathrm{x} \preceq_{\mathsf{char}} \mathrm{y}\}) & \text{if } \mathtt{s} = \mathrm{y} \in \mathrm{X}_{\mathrm{str}} \\ \mathsf{Clos}(\mathcal{C} \cup \{\mathrm{y} \preceq_{\mathsf{char}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\mathtt{s})\}) & \text{otherwise} \end{cases}$$

  The same applies for $[\![\mathtt{s} \texttt{ == } \mathrm{x}]\!]^{\mathsf{char}}\mathcal{C}$.

  - *conjunction abstract semantics*

$$[\![\mathsf{e}_1 \texttt{ \&\& } \mathsf{e}_2]\!]^{\mathsf{char}}\mathcal{C} = [\![\mathsf{e}_1]\!]^{\mathsf{char}}\mathcal{C} \sqcup_{\mathsf{char}} [\![\mathsf{e}_2]\!]^{\mathsf{char}}\mathcal{C}$$

  - *disjunction abstract semantics*

$$[\![\mathsf{e}_1 \texttt{ || } \mathsf{e}_2]\!]^{\mathsf{char}}\mathcal{C} = [\![\mathsf{e}_1]\!]^{\mathsf{char}}\mathcal{C} \sqcap_{\mathsf{char}} [\![\mathsf{e}_2]\!]^{\mathsf{char}}\mathcal{C}$$

Theorem B.2. The abstract semantics of Char is sound. Indeed, it holds that:

$$\forall \mathsf{M} \in \mathcal{P}(\mathbb{M}).\ \alpha_{\mathsf{char}}([\![\mathrm{x} = \mathtt{s}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{char}} ([\![\mathrm{x} = \mathtt{s}]\!]^{\mathsf{char}}\alpha_{\mathsf{char}}(\mathsf{M}))$$

and

$$\forall \mathsf{M} \in \mathcal{P}(\mathbb{M}).\ \alpha_{\mathsf{char}}([\![\mathsf{b}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{sub}} ([\![\mathtt{b}]\!]^{\mathsf{char}}\alpha_{\mathsf{char}}(\mathsf{M}))$$

Proof.
The proof is analogous to the one provided for Theorem B.3.

□

B.3 Abstract Semantics of Sub

Let $[\![\mathtt{st}]\!]^{\mathsf{sub}} : \mathsf{Sub} \to \mathsf{Sub}$ be the function that given an input abstract memory returns an abstract memory containing the new substring relations introduced by st.

- **Assignment abstract semantics**

Consider the assignment $\mathrm{x} = \mathtt{s}$ such that $\mathrm{x} \in \mathrm{X_{str}}$ and $\mathtt{s} \in \mathrm{SE}$. Formally, its abstract semantics, i.e., $[\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}}\mathcal{S}$ is given by the following steps:

– **[remove]**

$$\mathcal{S}_{\mathrm{r}} = \begin{cases} \mathcal{S} \setminus \{\mathrm{w} \preceq_{\mathsf{sub}} \mathrm{z} \mid \mathrm{w} = \mathrm{x}\} & \text{if x appears at the top-level of } \mathtt{s} \\ \mathcal{S} \setminus \{\mathrm{w} \preceq_{\mathsf{sub}} \mathrm{z} \mid \mathrm{w} = \mathrm{x} \vee \mathrm{z} = \mathrm{x}\} & \text{otherwise} \end{cases}$$

– **[add]**

$$\mathcal{S}_{\mathrm{a}} = \mathcal{S}_{\mathrm{r}} \cup \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\mathtt{s})\}$$

where extr is the extraction function defined in Section 6.3.2.

– **[closure]**

$$[\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}}\mathcal{S} = \mathsf{Clos}(\mathcal{S}_{\mathrm{a}})$$

- **Boolean expressions abstract semantics**

  - *contains abstract semantics*

$$[\![\ \mathtt{contains}(\mathrm{x}, \mathtt{s})\ ]\!]^{\mathsf{sub}}\mathcal{S} = \mathsf{Clos}(\mathcal{S} \cup \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\mathtt{s})\})$$

  - *equality abstract semantics*

$$[\![\ \mathrm{x}\ \mathtt{==}\ \mathtt{s}\ ]\!]^{\mathsf{sub}}\mathcal{S} = \begin{cases} \mathsf{Clos}(\mathcal{S} \cup \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x}, \mathrm{x} \preceq_{\mathsf{sub}} \mathrm{y}\}) & \text{if } \mathtt{s} = \mathrm{y} \in \mathrm{X_{str}} \\ \mathsf{Clos}(\mathcal{S} \cup \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\mathtt{s})\}) & \text{otherwise} \end{cases}$$

  The same applies for $[\![\ \mathtt{s}\ \mathtt{==}\ \mathrm{x}\ ]\!]^{\mathsf{sub}}\mathcal{S}$.

  - *conjunction abstract semantics*

$$[\![\ \mathsf{e}_1\ \mathtt{\&\&}\ \mathsf{e}_2\ ]\!]^{\mathsf{sub}}\mathcal{S} = [\![\ \mathsf{e}_1\ ]\!]^{\mathsf{sub}}\mathcal{S} \sqcup_{\mathsf{sub}} [\![\ \mathsf{e}_2\ ]\!]^{\mathsf{sub}}\mathcal{S}$$

  - *disjunction abstract semantics*

$$[\![\ \mathsf{e}_1\ \mathtt{||}\ \mathsf{e}_2\ ]\!]^{\mathsf{sub}}\mathcal{S} = [\![\ \mathsf{e}_1\ ]\!]^{\mathsf{sub}}\mathcal{S} \sqcap_{\mathsf{sub}} [\![\ \mathsf{e}_2\ ]\!]^{\mathsf{sub}}\mathcal{S}$$

Theorem B.3. The abstract semantics of Sub is sound, namely $\forall \mathsf{M} \in \mathcal{P}(\mathbb{M})$

$$\alpha_{\mathsf{sub}}([\![\mathrm{x} = \mathtt{s}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{sub}} [\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}}\alpha_{\mathsf{sub}}(\mathsf{M})$$

and,

$$\alpha_{\mathsf{sub}}([\![\mathsf{b}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{sub}} [\![\ \mathsf{b}\ ]\!]^{\mathsf{sub}}\alpha_{\mathsf{sub}}(\mathsf{M})$$

Proof.
The soundness proofs for the Boolean expressions (`contains`, `==`, disjunction and conjunction) are trivial and straightforward. Hence, we focus only the soundness proof of assignment.

First, we observe that, given a string expression $\mathtt{s}$, any sub-expression $\mathtt{s'}$ of $\mathtt{s}$ is substring of $\mathtt{s}$, for any $\mathfrak{m} \in \mathbb{M}$, namely

$$\forall \mathfrak{m} \in \mathbb{M} : [\![\mathtt{s'}]\!]\mathfrak{m} \curvearrowright [\![\mathtt{s}]\!]\mathfrak{m} \tag{B.1}$$

Hence, we prove that:

$$\alpha_{\mathsf{sub}}([\![\mathrm{x} = \mathtt{s}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{sub}} ([\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}}\alpha_{\mathsf{sub}}(\mathsf{M}))$$

Without loss of generality, let us suppose that x does not appear at the top-level of the expression $\mathtt{s}$. Note that the case in which x appears at the top-level of the expression $\mathtt{s}$ is analogous.

$$\begin{aligned}
&\alpha_{\mathsf{sub}}([\![\mathrm{x} = \mathtt{s}]\!]\mathsf{M}) = \\
&= \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{y}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z})\} && [\text{Def. } \alpha_{\mathsf{sub}}] \\
&= \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{y}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathrm{z} \neq \mathrm{x}\} \\
&\cup \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{y}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{x})\} && [\text{Set prop.}] \\
\sqsubseteq_{\mathsf{sub}}\ &\mathsf{Clos}((\{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{y}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathrm{z} \neq \mathrm{x}\} \\
&\setminus \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{z} \mid \mathrm{y} = \mathrm{x} \vee \mathrm{z} = \mathrm{x}\}) \\
&\cup \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{y}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{x})\}) \\
&= \mathsf{Clos}((\{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{y}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathrm{z} \neq \mathrm{x}\} \\
&\setminus \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{z} \mid \mathrm{y} = \mathrm{x} \vee \mathrm{z} = \mathrm{x}\}) \\
&\cup \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{y}) \curvearrowright [\![\mathtt{s}]\!]\mathfrak{m}\}) && [\text{Def. } \mathfrak{m}(\mathrm{x})] \\
\sqsubseteq_{\mathsf{sub}}\ &\mathsf{Clos}((\{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{y}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathrm{z} \neq \mathrm{x}\} \\
&\setminus \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{z} \mid \mathrm{y} = \mathrm{x} \vee \mathrm{z} = \mathrm{x}\}) \cup \{\mathrm{y} \preceq_{\mathsf{sub}} \mathrm{x} \mid \mathrm{y} \in \mathsf{extr}(\mathtt{s})\}) && [\text{Eq. B.1}] \\
&= \mathsf{Clos}(\mathcal{S}_{\mathrm{a}}) = [\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}}\alpha_{\mathsf{sub}}(\mathsf{M}) && [\text{Def. } [\![\ \cdot\ ]\!]^{\mathsf{sub}}]
\end{aligned}$$

□

B.4 Abstract Semantics of $\mathsf{Sub}^\star$

Let $[\![\ \mathtt{st}\ ]\!]^{\mathsf{sub}^\star} : \mathsf{Sub}^\star \to \mathsf{Sub}^\star$ be the function that given an input abstract memory returns an abstract memory containing the new substring relations introduced by $\mathtt{st}$.

- **Assignment abstract semantics**

  Consider the assignment $\mathrm{x} = \mathtt{s}$ such that $\mathrm{x} \in \mathrm{X}_{\mathrm{str}}$ and $\mathtt{s} \in \textsc{se}$. Formally, its abstract semantics, i.e., $[\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}}\mathcal{S}$ is given by the following steps:

  – **[remove]**

$$\mathcal{S}^{\star}_{\mathrm{r}} = \begin{cases} \mathcal{S}^{\star} \setminus \{\mathtt{s'} \preceq_{\mathsf{sub}^{\star}} \mathrm{z} \mid \mathrm{x} \text{ appears in } \mathtt{s'}\} & \text{if x appears} \\ & \text{at the top-level of } \mathtt{s} \\ \mathcal{S}^{\star} \setminus \{\mathtt{s'} \preceq_{\mathsf{sub}^{\star}} \mathrm{z} \mid \mathrm{z} = \mathrm{x} \vee \mathrm{x} \text{ appears in } \mathtt{s'}\} & \text{otherwise} \end{cases}$$

  – **[add]**

$$\mathcal{S}^{\star}_{\mathrm{a}} = \mathcal{S}^{\star}_{\mathrm{r}} \cup \{\mathtt{s'} \preceq_{\mathsf{sub}^{\star}} \mathrm{x} \mid \mathtt{s'} \in \mathsf{extr}^{\star}(\mathtt{s})\}$$

  – **[inter-asg]**

$$\mathcal{S}^{\star}_{\mathrm{i}} = \mathcal{S}^{\star}_{\mathrm{a}} \cup \{\mathrm{x} \preceq_{\mathsf{sub}^{\star}} \mathrm{y} \mid \mathcal{S}^{\star}_{\mathrm{a}}(\mathrm{x}) \subseteq \mathcal{S}^{\star}_{\mathrm{a}}(\mathrm{y})\}$$

  – **[closure]**

$$[\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star} = \mathsf{Clos}(\mathcal{S}^{\star}_{\mathrm{i}})$$

- **Boolean expressions abstract semantics**

  - *contains abstract semantics*

$$[\![\ \mathtt{contains}(\mathrm{x}, \mathtt{s})\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star} = \mathsf{Clos}(\mathcal{S}^{\star} \cup \{\mathtt{s'} \preceq_{\mathsf{sub}^{\star}} \mathrm{x} \mid \mathtt{s'} \in \mathsf{extr}^{\star}(\mathtt{s})\})$$

  - *equality abstract semantics*

$$[\![\ \mathrm{x} == \mathtt{s}\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star} = \mathsf{Clos}(\mathcal{S}^{\star} \cup \{\mathtt{s'} \preceq_{\mathsf{sub}^{\star}} \mathrm{x} \mid \mathtt{s'} \in \mathsf{extr}^{\star}(\mathtt{s})\})$$

    Thee same applies for $[\![\ \mathtt{s} == \mathrm{x}\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star}$.

  - *conjunction abstract semantics*

$$[\![\ \mathsf{e}_1\ \&\&\ \mathsf{e}_2\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star} = [\![\ \mathsf{e}_1\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star} \sqcup_{\mathsf{sub}^{\star}} [\![\ \mathsf{e}_2\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star}$$

  - *disjunction abstract semantics*

$$[\![\ \mathsf{e}_1\ ||\ \mathsf{e}_2\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star} = [\![\ \mathsf{e}_1\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star} \sqcap_{\mathsf{sub}^{\star}} [\![\ \mathsf{e}_2\ ]\!]^{\mathsf{sub}^{\star}}\mathcal{S}^{\star}$$

Theorem B.4. The abstract semantics of $\mathsf{Sub}^{\star}$ is sound, namely $\forall \mathsf{M} \in \mathcal{P}(\mathbb{M})$

$$\alpha_{\mathsf{sub}^{\star}}([\![\mathrm{x} = \mathtt{s}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{sub}^{\star}} [\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}^{\star}}\alpha_{\mathsf{sub}^{\star}}(\mathsf{M})$$

and,

$$\alpha_{\mathsf{sub}^\star}([\![\mathtt{b}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{sub}^\star} [\![\ \mathtt{b}\ ]\!]^{\mathsf{sub}^\star}\, \alpha_{\mathsf{sub}^\star}(\mathsf{M})$$

Proof. In the following we prove that:

$$\alpha_{\mathsf{sub}^\star}([\![\mathrm{x} = \mathtt{s}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{sub}^\star} ([\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}^\star}\, \alpha_{\mathsf{sub}^\star}(\mathsf{M}))$$

The proof is similar to the one reported above, except for the phase **[inter-asg]**. As above, we assume that x does not appear at the top-level of the expression $\mathtt{s}$ and the contrary in analogous.

$$\begin{aligned}
& \alpha_{\mathsf{sub}^\star}([\![\mathrm{x} = \mathtt{s}]\!]\mathsf{M}) = \\
= \ & \{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ [\![\mathtt{s'}]\!]((\,)\mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}]) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z})\} \\
& [\text{Def. } \alpha_{\mathsf{sub}^\star}] \\
= \ & \{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathtt{s}'(\mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}]) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathtt{s'} \neq \mathrm{x}\} \\
& \cup \{\mathrm{x} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{x}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z})\} \\
& [\text{Set prop.}] \\
= \ & \{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathtt{s'}(\mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}]) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathtt{s}' \neq \mathrm{x}, \mathrm{z} \neq \mathrm{x}\} \\
\cup \ & \{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathtt{s'}(\mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}]) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathtt{s'} \neq \mathrm{x}\} \\
\cup \ & \{\mathrm{x} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{x}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z})\} \\
& [\text{Set prop.}] \\
\sqsubseteq_{\mathsf{sub}^\star} \ & \mathsf{Clos}(\{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathtt{s'}(\mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}]) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathtt{s'} \neq \mathrm{x}, \mathrm{z} \neq \mathrm{x}\} \\
& \setminus \{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{x} \mid \mathrm{z} = \mathrm{x} \vee \mathrm{x} \text{ appears in } \mathtt{s}\} \\
& \cup \{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathtt{s'}(\mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}]) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathtt{s'} \neq \mathrm{x}\} \\
& \cup \{\mathrm{x} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{x}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z})\}) \\
\sqsubseteq_{\mathsf{sub}^\star} \ & \mathsf{Clos}((\{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathtt{s'}(\mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}]) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z}), \mathtt{s'} \neq \mathrm{x}, \mathrm{z} \neq \mathrm{x}\} \\
& \setminus \{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{x} \mid \mathrm{z} = \mathrm{x} \vee \mathrm{x} \text{ appears in } \mathtt{s}\}) \\
& \cup \{\mathtt{s'} \preceq_{\mathsf{sub}^\star} \mathrm{x} \mid \mathtt{s'} \in \mathsf{extr}^\star(\mathtt{s})\} \\
& \cup \{\mathrm{x} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{x}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z})\}) \\
& [\text{Eq. B.1}] \\
= \ & \mathsf{Clos}(\mathcal{S}^\star_{\mathrm{a}} \cup \{\mathrm{x} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{x}) \curvearrowright \mathfrak{m}[\mathrm{x} \leftarrow [\![\mathtt{s}]\!]\mathfrak{m}](\mathrm{z})\}) \\
& [\text{Def. } \mathcal{S}^\star_{\mathrm{a}}] \\
\sqsubseteq_{\mathsf{sub}^\star} \ & \mathsf{Clos}(\mathcal{S}^\star_{\mathrm{a}} \cup \{\mathrm{x} \preceq_{\mathsf{sub}^\star} \mathrm{z} \mid \mathcal{S}^\star_{\mathrm{a}}(\mathrm{x}) \subseteq \mathcal{S}^\star_{\mathrm{a}}(\mathrm{z})\}) = [\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{sub}^\star}\, \alpha_{\mathsf{sub}^\star}(\mathsf{M}) \\
& [\text{Def. } [\![\ \cdot\ ]\!]^{\mathsf{sub}^\star}]
\end{aligned}$$

□

B.5 $\mathsf{Len}^\star$ Abstract Domain

The Len relational string abstract domain can be extended in order to track also relations between string expressions and variables, e.g., if the length of the concatenation between two string variables is smaller than or equal to the length of another string variable. Thus, similarly to Section 6.3.5, we introduce the binary relation $\preceq_{\mathsf{len}^\star} \subseteq \text{SE} \times \text{X}_{\text{str}}$ and the extended abstract domain $\mathsf{Len}^\star = \mathcal{P}(\{\mathtt{s} \preceq_{\mathsf{len}^\star} \text{x} \mid \text{x} \in \text{X}_{\text{str}},\ \mathtt{s} \in \text{SE}\}) \cup \{\bot_{\mathsf{len}^\star}\}$, which ranges over the meta-variable $\mathcal{L}^\star$, where the top element, denoted by $\top_{\mathsf{len}^\star}$, is $\emptyset$, and $\bot_{\mathsf{len}^\star}$ is the bottom element. $\mathsf{Len}^\star$ inherits the properties of Len. $\mathsf{Len}^\star$ lattice operations are $\sqcup_{\mathsf{len}^\star}$ (lub), $\sqcap_{\mathsf{len}^\star}$ (glb) and $\preceq_{\mathsf{len}^\star}$ (partial order) and are similar to the ones of Len reported in Figure 6.4. Its concretization $\gamma_{\mathsf{len}^\star} : \mathsf{Len}^\star \to \mathcal{P}(\mathbb{M})$ and abstraction $\alpha_{\mathsf{len}^\star} : \mathcal{P}(\mathbb{M}) \to \mathsf{Len}^\star$ functions, which form a Galois Connection, are defined as follows:

$$\gamma_{\mathsf{len}^\star}(\mathcal{L}^\star) = \bigcap_{\mathtt{s} \preceq_{\mathsf{len}^\star} \text{x} \in \mathcal{L}^\star} \{\mathfrak{m} \mid \mathfrak{m}(\text{x}), [\![\mathtt{s}]\!]\mathfrak{m} \in \Sigma^*,\ |[\![\mathtt{s}]\!]\mathfrak{m}| \leq |\mathfrak{m}(\text{x})|\}$$

$$\alpha_{\mathsf{len}^\star}(\mathsf{M}) = \{\mathtt{s} \preceq_{\mathsf{len}^\star} \text{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ |[\![\mathtt{s}]\!]\mathfrak{m}| \leq |\mathfrak{m}(\text{x})|,\ \text{x} \in \text{X}_{\text{str}},\ \mathtt{s} \in \text{SE}\}$$

The $\mathsf{Len}^\star$ abstract semantics extends the one defined for Len and it is defined as follows. Let $[\![\ \mathtt{st}\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star : \mathsf{Len}^\star \to \mathsf{Len}^\star$.

- **Assignment abstract semantics**

  Consider the assignment $\text{x} = \mathtt{s}$ such that $\text{x} \in \text{X}_{\text{str}}$ and $\mathtt{s} \in \text{SE}$. Formally, its abstract semantics, i.e., $[\![\ \text{x} = \mathtt{s}\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star$, is given by the following steps:

  - **[remove]**

$$\mathcal{L}^\star_{\text{r}} = \begin{cases} \mathcal{L}^\star \setminus \{\mathtt{s'} \preceq_{\mathsf{len}^\star} \text{z} \mid \text{x appears in } \mathtt{s'}\} & \text{if x appears} \\ & \text{at the top-level of } \mathtt{s} \\ \mathcal{L}^\star \setminus \{\mathtt{s'} \preceq_{\mathsf{len}^\star} \text{z} \mid \text{z} = \text{x} \vee \text{x appears in } \mathtt{s}\} & \text{otherwise} \end{cases}$$

  - **[add]**

$$\mathcal{L}^\star_{\text{a}} = \mathcal{L}^\star_{\text{r}} \cup \{\mathtt{s'} \preceq_{\mathsf{len}^\star} \text{x} \mid \mathtt{s'} \in \mathsf{extr}^\star(\mathtt{s})\}$$

  - **[inter-asg]**

$$\mathcal{L}^\star_{\text{i}} = \mathcal{L}^\star_{\text{a}} \cup \{\text{x} \preceq_{\mathsf{len}^\star} \text{y} \mid \mathcal{L}^\star_{\text{a}}(\text{x}) \subseteq \mathcal{L}^\star_{\text{a}}(\text{y})\}$$

  - **[closure]**

$$[\![\ \text{x} = \mathtt{s}\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star = \mathsf{Clos}(\mathcal{L}^\star_{\text{i}})$$

- **Boolean expressions abstract semantics**

- *contains abstract semantics*

$$[\![\ \texttt{contains}(\mathrm{x}, \mathtt{s})\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star = \mathsf{Clos}(\mathcal{L}^\star \cup \{\mathtt{s}' \preceq_{\mathsf{len}^\star} \mathrm{x} \mid \mathtt{s}' \in \mathsf{extr}^\star(\mathtt{s})\})$$

- *equality abstract semantics*

$$[\![\ \mathrm{x}\ \texttt{==}\ \mathtt{s}\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star = \mathsf{Clos}(\mathcal{L}^\star \cup \{\mathtt{s}' \preceq_{\mathsf{len}^\star} \mathrm{x} \mid \mathtt{s}' \in \mathsf{extr}^\star(\mathtt{s})\})$$

  The same applies for $[\![\ \mathtt{s}\ \texttt{==}\ \mathrm{x}\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star$.

- *conjunction abstract semantics*

$$[\![\ \mathsf{e}_1\ \texttt{\&\&}\ \mathsf{e}_2\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star = [\![\ \mathsf{e}_1\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star \sqcup_{\mathsf{len}^\star} [\![\ \mathsf{e}_2\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star$$

- *disjunction abstract semantics*

$$[\![\ \mathsf{e}_1\ \texttt{||}\ \mathsf{e}_2\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star = [\![\ \mathsf{e}_1\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star \sqcap_{\mathsf{len}^\star} [\![\ \mathsf{e}_2\ ]\!]^{\mathsf{len}^\star}\mathcal{L}^\star$$

Theorem B.5. The abstract semantics of $\mathsf{Len}^\star$ is sound. Indeed, it holds that:

$$\forall \mathsf{M} \in \mathcal{P}(\mathbb{M}).\ \alpha_{\mathsf{len}^\star}([\![\mathrm{x} = \mathtt{s}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{len}^\star} ([\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{len}^\star}\alpha_{\mathsf{len}^\star}(\mathsf{M}))$$

and

$$\forall \mathsf{M} \in \mathcal{P}(\mathbb{M}).\ \alpha_{\mathsf{len}^\star}([\![\mathtt{b}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{len}^\star} ([\![\ \mathtt{b}\ ]\!]^{\mathsf{len}^\star}\alpha_{\mathsf{len}^\star}(\mathsf{M}))$$

Proof.
The proof is analogous to the one provided for Theorem B.4.

□

B.6 $\mathsf{Char}^\star$ Abstract Domain

We define $\preceq_{\mathsf{char}^\star}$ as the extension of the binary relation $\preceq_{\mathsf{char}}$ to string expressions such that $\preceq_{\mathsf{char}^\star} \subseteq \textsc{se} \times \mathrm{X}_{\mathrm{str}}$. Moreover, Char becomes $\mathsf{Char}^\star = \mathcal{P}(\{\mathtt{s} \preceq_{\mathsf{char}^\star} \mathrm{x} \mid \mathrm{x} \in \mathrm{X}_{\mathrm{str}},\ \mathtt{s} \in \textsc{se}\}) \cup \{\bot_{\mathsf{char}^\star}\}$, where the top element, denoted by $\top_{\mathsf{char}^\star}$, is $\emptyset$, and $\bot_{\mathsf{char}^\star}$ is the bottom element. The abstract domain $\mathsf{Char}^\star$ ranges over the meta-variable $\mathcal{C}^\star$. The extended relational character inclusion string abstract domain is a complete lattice and its lattice operations ($\sqcup_{\mathsf{char}^\star}$(lub), $\sqcap_{\mathsf{char}^\star}$(glb), $\sqsubseteq_{\mathsf{char}^\star}$ (partial order)) are similar to the ones of Char. The concretization $\gamma_{\mathsf{char}^\star} : \mathsf{Char}^\star \to \mathcal{P}(\mathbb{M})$ and abstraction $\alpha_{\mathsf{char}^\star} : \mathcal{P}(\mathbb{M}) \to \mathsf{Char}^\star$ functions, defined below, form a Galois Connection.

$$\gamma_{\mathsf{char}^\star}(\mathcal{C}^\star) = \bigcap_{\mathtt{s} \preceq_{\mathsf{char}^\star} \mathrm{x} \in \mathcal{C}^\star} \{\mathfrak{m} \mid \mathfrak{m}(\mathrm{x}), [\![\mathtt{s}]\!]\mathfrak{m} \in \Sigma^*, \mathrm{char}([\![\mathtt{s}]\!]\mathfrak{m}) \subseteq \mathrm{char}(\mathfrak{m}(\mathrm{x}))\}$$

$$\alpha_{\mathsf{char}^\star}(\mathsf{M}) = \{\mathtt{s} \preceq_{\mathsf{char}^\star} \mathrm{x} \mid \forall \mathfrak{m} \in \mathsf{M}.\ \mathrm{char}([\![\mathtt{s}]\!]\mathfrak{m}) \subseteq \mathrm{char}(\mathfrak{m}(\mathrm{x})),\ \mathrm{x} \in \mathrm{X_{str}},\ \mathtt{s} \in \mathrm{SE}\}$$

As usual, the abstract semantics is captured by the function $[\![\ \mathtt{st}\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star : \mathsf{Char}^\star \to \mathsf{Char}^\star$. and it is defined as follows.

- **Assignment abstract semantics**

  Consider the assignment $\mathrm{x} = \mathtt{s}$ such that $\mathrm{x} \in \mathrm{X_{str}}$ and $\mathtt{s} \in \mathrm{SE}$. Formally, its abstract semantics, i.e., $[\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star$ is given by the following steps:

  – **[remove]**

$$\mathcal{C}^\star_{\mathrm{r}} = \begin{cases} \mathcal{C}^\star \setminus \{\mathtt{s'} \preceq_{\mathsf{char}^\star} \mathrm{z} \mid \mathrm{x} \text{ appears in } \mathtt{s'}\} & \text{if x appears} \\ & \text{at the top-level of } \mathtt{s} \\ \mathcal{C}^\star \setminus \{\mathtt{s'} \preceq_{\mathsf{char}^\star} \mathrm{z} \mid \mathrm{z} = \mathrm{x} \vee \mathrm{x} \text{ appears in } \mathtt{s'}\} & \text{otherwise} \end{cases}$$

  – **[add]**

$$\mathcal{C}^\star_{\mathrm{a}} = \mathcal{C}^\star_{\mathrm{r}} \cup \{\mathtt{s'} \preceq_{\mathsf{char}^\star} \mathrm{x} \mid \mathtt{s'} \in \mathsf{extr}^\star(\mathtt{s})\}$$

  – **[inter-asg]**

$$\mathcal{C}^\star_{\mathrm{i}} = \mathcal{C}^\star_{\mathrm{a}} \cup \{\mathrm{x} \preceq_{\mathsf{char}^\star} \mathrm{y} \mid \mathcal{C}^\star_{\mathrm{a}}(\mathrm{x}) \subseteq \mathcal{C}^\star_{\mathrm{a}}(\mathrm{y})\}$$

  – **[closure]**

$$[\![\ \mathrm{x} = \mathtt{s}\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star = \mathsf{Clos}(\mathcal{C}^\star_{\mathrm{i}})$$

- **Boolean expressions abstract semantics**

  - *contains abstract semantics*

$$[\![\ \mathtt{contains}(\mathrm{x}, \mathtt{s})\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star = \mathsf{Clos}(\mathcal{C}^\star \cup \{\mathtt{s'} \preceq_{\mathsf{char}^\star} \mathrm{x} \mid \mathtt{s'} \in \mathsf{extr}^\star(\mathtt{s})\})$$

  - *equality abstract semantics*

$$[\![\ \mathrm{x}\ \mathtt{==}\ \mathtt{s}\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star = \mathsf{Clos}(\mathcal{C}^\star \cup \{\mathtt{s'} \preceq_{\mathsf{char}^\star} \mathrm{x} \mid \mathtt{s'} \in \mathsf{extr}^\star(\mathtt{s})\})$$

    The same applies for $[\![\ \mathtt{s}\ \mathtt{==}\ \mathrm{x}\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star$.

  - *conjunction abstract semantics*

$$[\![\ \mathsf{e}_1\ \mathtt{\&\&}\ \mathsf{e}_2\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star = [\![\ \mathsf{e}_1\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star \sqcup_{\mathsf{char}^\star} [\![\ \mathsf{e}_2\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star)$$

  - *disjunction abstract semantics*

$$[\![\ \mathsf{e}_1\ \mathtt{||}\ \mathsf{e}_2\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star = [\![\ \mathsf{e}_1\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star \sqcap_{\mathsf{char}^\star} [\![\ \mathsf{e}_2\ ]\!]^{\mathsf{char}^\star}\mathcal{C}^\star$$

Theorem B.6. The abstract semantics of $\mathsf{Char}^\star$ is sound. Indeed, it holds that:

$$\forall \mathsf{M} \in \mathcal{P}(\mathbb{M}).\ \alpha_{\mathsf{char}^\star}([\![\mathrm{x} = \mathsf{s}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{char}^\star} ([\![\ \mathrm{x} = \mathsf{s}\ ]\!]^{\mathsf{char}^\star} \alpha_{\mathsf{char}^\star}(\mathsf{M}))$$

and

$$\forall \mathsf{M} \in \mathcal{P}(\mathbb{M}).\ \alpha_{\mathsf{char}^\star}([\![\mathsf{b}]\!]\mathsf{M}) \sqsubseteq_{\mathsf{sub}^\star} ([\![\ \mathsf{b}\ ]\!]^{\mathsf{char}^\star} \alpha_{\mathsf{char}^\star}(\mathsf{M}))$$

Proof.
The proof is analogous to the one provided for Theorem B.4.

□

BIBLIOGRAPHY

[1] P. A. Abdulla et al. "Norn: An SMT Solver for String Constraints". In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. 2015, pp. 462–469. DOI: 10.1007/978-3-319-21690-4_29.

[2] E. Agirre et al. "A Study on Similarity and Relatedness Using Distributional and WordNet-based Approaches". In: Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, May 31 - June 5, 2009, Boulder, Colorado, USA. The Association for Computational Linguistics, 2009, pp. 19–27.

[3] R. Agrawal, P. J. Haas, and J. Kiernan. "Watermarking Relational Data: Framework, Algorithms and Analysis". In: VLDB J. 12.2 (2003), pp. 157–169. DOI: 10.1007/s00778-003-0097-x.

[4] R. Agrawal and J. Kiernan. "Watermarking Relational Databases". In: Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002. Morgan Kaufmann, 2002, pp. 155–166. DOI: 10.1016/B978-155860869-6/50022-6.

[5] M. T. Ahvanooey et al. "A Comparative Analysis of Information Hiding Techniques for Copyright Protection of Text Documents". In: Security and Communication Networks 2018 (2018), 5325040:1–5325040:22. DOI: 10.1155/2018/5325040.

[6] A. Al-Haj and A. Odeh. "Robust and Blind Watermarking of Relational Database Systems". In: Journal of Computer Science 4.12 (2008), pp. 1024–1029.

[7] F. E. Allen. "Control flow analysis". In: SIGPLAN Notices. Vol. 5. 1970, pp. 1–19. DOI: 10.1145/390013.808479.

[8] R. Amadini, G. Gange, and P. J. Stuckey. "Dashed Strings for String Constraint Solving". In: Artificial Intelligence. Vol. 289. 2020. DOI: https://doi.org/10.1016/j.artint.2020.103368.

[9] R. Amadini et al. "A Novel Approach to String Constraint Solving". In: Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. 2017, pp. 3–20. URL: 10.1007/978-3-319-66158-2_1.

[10] R. Amadini et al. "Combining String Abstract Domains for JavaScript Analysis: An Evaluation". In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. 2017, pp. 41–57. DOI: 10.1007/978-3-662-54577-5_3.

[11] R. Amadini et al. "Reference Abstract Domains and Applications to String Analysis". In: Fundam. Inform. 158.4 (2018), pp. 297–326. DOI: 10.3233/FI-2018-1650.

[12] V. Arceri. "Taming Strings in Dynamic Languages - An Abstract Interpretation-based Static Analysis Approach". PhD thesis. May 2020. DOI: 10.13140/RG.2.2.27093.45286.

[13] V. Arceri and S. Maffeis. "Abstract Domains for Type Juggling". In: Electron. Notes Theor. Comput. Sci. 331 (2017), pp. 41–55. DOI: 10.1016/j.entcs.2017.02.003.

[14] V. Arceri and I. Mastroeni. "A Sound Abstract Interpreter for Dynamic Code". In: SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, Brno, Czech Republic, March 30 - April 3, 2020. Ed. by C. Hung et al. ACM, 2020, pp. 1979–1988. DOI: 10.1145/3341105.3373964.

[15] V. Arceri and I. Mastroeni. "An Automata-based Abstract Semantics for String Manipulation Languages". In: Proceedings Seventh International Workshop on Verification and Program Transformation, VPT@Programming 2019, Genova, Italy, 2nd April 2019. Ed. by A. Lisitsa and A. P. Nemytykh. Vol. 299. EPTCS. 2019, pp. 19–33. DOI: 10.4204/EPTCS.299.5.

[16] V. Arceri and I. Mastroeni. "Static Program Analysis for String Manipulation Languages". In: Proceedings Seventh International Workshop on Verification and Program Transformation, Genova, Italy, 2nd April 2019. Ed. by A. Lisitsa and A. Nemytykh. Vol. 299. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2019, pp. 19–33. DOI: 10.4204/EPTCS.299.5.

[17] V. Arceri, I. Mastroeni, and S. Xu. "Static Analysis for ECMAScript String Manipulation Programs". In: Appl. Sci. 10 (2020), p. 3525. DOI: 10.3390/app10103525.

[18] V. Arceri et al. "Completeness of Abstract Domains for String Analysis of JavaScript Programs". In: Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings. Ed.

by R. M. Hierons and M. Mosbah. Vol. 11884. Lecture Notes in Computer Science. Springer, 2019, pp. 255–272. DOI: 10.1007/978-3-030-32505-3_15.

[19] N. Ayewah et al. "Using Static Analysis to Find Bugs". In: IEEE Software 25.5 (2008), pp. 22–29. DOI: 10.1109/MS.2008.130.

[20] R. Bagnara, P. M. Hill, and E. Zaffanella. "Exact Join Detection for Convex Polyhedra and Other Numerical Abstractions". In: Comput. Geom. 43.5 (2010), pp. 453–473. DOI: 10.1016/j.comgeo.2009.09.002.

[21] R. Bagnara, P. M. Hill, and E. Zaffanella. "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems". In: Sci. Comput. Program. 72.1-2 (2008), pp. 3–21. DOI: 10.1016/j.scico.2007.08.001.

[22] C. Baier and J. P. Katoen. "Principles of Model Checking". MIT Press, 2008.

[23] Z. Baranová et al. "Model Checking of C and C++ with DIVINE 4". In: Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings. 2017, pp. 201–207. DOI: 10.1007/978-3-319-68167-2_14.

[24] M. Batet and D. Sánchez. "A Review on Semantic Similarity". In: Encyclopedia of Information Science and Technology, Third Edition. IGI Global, 2015, pp. 7575–7583.

[25] S. Bautista, T. Jensen, and B. Montagu. "Numeric Domains Meet Algebraic Data Types". In: Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains. NSAD 2020. Virtual, USA: Association for Computing Machinery, 2020, 12–16. DOI: 10.1145/3427762.3430178.

[26] E. Bertino et al. "Privacy and Ownership Preserving of Outsourced Medical Data". In: Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan. Ed. by K. Aberer, M. J. Franklin, and S. Nishio. IEEE Computer Society, 2005, pp. 521–532. DOI: 10.1109/ICDE.2005.111.

[27] A. Bessey et al. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World". In: Commun. ACM 53.2 (2010), pp. 66–75. DOI: 10.1145/1646353.1646374.

[28] S. Bhattacharya and A. Cortesi. "A Distortion Free Watermark Framework for Relational Databases". In: ICSOFT 2009 - Proceedings of the 4th International Conference on Software and Data Technologies, Volume 2, Sofia, Bulgaria, July 26-29, 2009. Ed. by B. Shishkov, J. Cordeiro, and A. Ranchordas. INSTICC Press, 2009, pp. 229–234.

[29] S. Bhattacharya and A. Cortesi. "A Generic Distortion Free Watermarking Technique for Relational Databases". In: Information Systems Security, 5th International Conference, ICISS 2009, Kolkata, India, December 14-18, 2009, Proceedings. Ed. by A. Prakash and I. Gupta. Vol. 5905. Lecture Notes in Computer Science. Springer, 2009, pp. 252–264. DOI: 10.1007/978-3-642-10772-6_19.

[30] T. Bultan et al. "String Analysis for Software Verification and Security". Springer, 2017. DOI: 10.1007/978-3-319-68670-7.

[31] S. Cass. "IEEE Spectrum Ranking 2019". https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019. Last check: November 23rd, 2020.

[32] I. Casso et al. "Computing Abstract Distances in Logic Programs". In: Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8-10, 2019, Revised Selected Papers. Ed. by Maurizio Gabbrielli. Vol. 12042. Lecture Notes in Computer Science. Springer, 2019, pp. 57–72. DOI: 10.1007/978-3-030-45260-5_4.

[33] C. Chang, T. S. Nguyen, and C. Lin. "A Blind Robust Reversible Watermark Scheme for Textual Relational Databases with Virtual Primary Key". In: Digital-Forensics and Watermarking - 13th International Workshop, IWDW 2014, Taipei, Taiwan, October 1-4, 2014. Revised Selected Papers. Ed. by Y. Shi et al. Vol. 9023. Lecture Notes in Computer Science. Springer, 2014, pp. 75–89. DOI: 10.1007/978-3-319-19321-2_6.

[34] A. S. Christensen, A. Møller, and M. I. Schwartzbach. "Precise Analysis of String Expressions". In: Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings. 2003, pp. 1–18. DOI: 10.1007/3-540-44898-5_1.

[35] E. M. Clarke, O. Grumberg, and D. E. Long. "Model Checking and Abstraction". In: ACM Trans. Program. Lang. Syst. 16.5 (1994), pp. 1512–1542. DOI: 10.1145/186025.186051.

[36] E. M. Clarke et al. "Symbolic Model Checking". In: Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings. 1996, pp. 419–427. DOI: 10.1007/3-540-61474-5_93.

[37] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: Commun. ACM 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685.

[38] M. Codish et al. "Improving Abstract Interpretations by Combining Domains". In: ACM Trans. Program. Lang. Syst. 17.1 (1995), pp. 28–44. DOI: 10.1145/200994.200998.

[39] Colorado-State-University. "Forest CoverType, The UCI KDD Archive". http://kdd.ics.uci.edu/databases/covertype/covertype.html. Information and Computer Science. University of California, Irvine. Remote Sensing and GIS Program Department of Forest Sciences. College of Natural Resources. Colorado State University. Fort Collins, CO 80523. June 1999.

[40] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. "Combinations of Abstract Domains for Logic Programming". In: Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994. 1994, pp. 227–239. DOI: 10.1145/174675.177880.

[41] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. "Combinations of Abstract Domains for Logic Programming: Open Product and Generic Pattern Construction". In: Sci. Comput. Program. 38.1-3 (2000), pp. 27–71. DOI: 10.1016/S0167-6423(99)00045-3.

[42] A. Cortesi, G. Costantini, and P. Ferrara. "A Survey on Product Operators in Abstract Interpretation". In: Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, Manhattan, Kansas, USA, 19-20th September 2013. 2013, pp. 325–336. DOI: 10.4204/EPTCS.129.19.

[43] A. Cortesi, G. Costantini, and P. Ferrara. "The Abstract Domain of Trapezoid Step Functions". In: Computer Languages, Systems & Structures 43 (2015), pp. 41–68. DOI: 10.1016/j.cl.2015.04.002.

[44] A. Cortesi, G. Filé, and W. H. Winsborough. "Comparison of Abstract Interpretations". In: Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings. 1992, pp. 521–532. DOI: 10.1007/3-540-55719-9_101.

[45] A. Cortesi, G. Filé, and W. H. Winsborough. "The Quotient of an Abstract Interpretation". In: Theor. Comput. Sci. 202.1-2 (1998), pp. 163–192. DOI: 10.1016/S0304-3975(97)00137-0.

[46] A. Cortesi and M.Zanioli. "Widening and Narrowing Operators for Abstract Interpretation". In: Comput. Lang. Syst. Struct. 37.1 (2011), pp. 24–42. DOI: 10.1016/j.cl.2010.09.001.

[47] A. Cortesi and M. Olliaro. "M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs". In: 2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018. Ed. by J. Pang et al. IEEE Computer Society, 2018, pp. 1–8. DOI: 10.1109/TASE.2018.00009.

[48] A. Cortesi et al. "String Abstraction for Model Checking of C Programs". In: Model Checking Software - 26th International Symposium, SPIN 2019, Beijing, China, July 15-16, 2019, Proceedings. Ed. by F. Biondi, T. G. Wilson, and A. Legay. Vol. 11636. Lecture Notes in Computer Science. Springer, 2019, pp. 74–93. DOI: 10.1007/978-3-030-30923-7_5.

[49] G. Costantini, P. Ferrara, and A. Cortesi. "A Suite of Abstract Domains for Static Analysis of String Values". In: Softw., Pract. Exper. 45.2 (2015), pp. 245–287. DOI: 10.1002/spe.2218.

[50] P. Cousot. "Abstract Interpretation Based Formal Methods and Future Challenges". In: Informatics - 10 Years Back. 10 Years Ahead. 2001, pp. 138–156. DOI: 10.1007/3-540-44577-3_10.

[51] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[52] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". In: J. Log. Comput. 2.4 (1992), pp. 511–547. DOI: 10.1093/logcom/2.4.511.

[53] P. Cousot and R. Cousot. "An Abstract Interpretation-based Framework for Software Watermarking". In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004. 2004, pp. 173–185. DOI: 10.1145/964001.964016.

[54] P. Cousot and R. Cousot. "Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation". In: Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings. 1992, pp. 269–295. DOI: 10.1007/3-540-55844-6_142.

[55] P. Cousot and R. Cousot. "Constructive Versions of Tarski's Fixed Point Theorems". In: Pacific Journal of Mathematics 81.1 (1979), pp. 43–57.

[56] P. Cousot and R. Cousot. "Systematic Design of Program Analysis Frameworks". In: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979. 1979, pp. 269–282. DOI: 10.1145/567752.567778.

[57] P. Cousot, R. Cousot, and F. Logozzo. "A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis". In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,

POPL 2011, Austin, TX, USA, January 26-28, 2011. 2011, pp. 105–118. DOI: 10.1145/1926385.1926399.

[58] P. Cousot and N. Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. 1978, pp. 84–96. DOI: 10.1145/512760.512770.

[59] P. Cousot et al. "The ASTREÉ Analyzer". In: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Ed. by S. Sagiv. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 21–30. DOI: 10.1007/978-3-540-31987-0_3.

[60] I. Cox et al. "Digital Watermarking and Steganography". M. Kaufmann, 2007.

[61] A. Dey, S. Bhattacharya, and N. Chaki. "Software Watermarking: Progress and Challenges". In: (2018). Exported from https://app.dimensions.ai on 2019/03/16, pp. 1–11. DOI: 10.1007/s41403-018-0058-8.

[62] N. Dor, M. Rodeh, and S. Sagiv. "CSSV: towards a realistic tool for statically detecting all buffer overflows in C". In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003. 2003, pp. 155–167. DOI: 10.1145/781131.781149.

[63] D. Evans and D. Larochelle. "Improving Security Using Extensible Lightweight Static Analysis". In: IEEE Software 19.1 (2002), pp. 42–51. DOI: 10.1109/52.976940.

[64] M. Fähndrich and F. Logozzo. "Static Contract Checking with Abstract Interpretation". In: Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers. Ed. by B. Beckert and C. Marché. Vol. 6528. Lecture Notes in Computer Science. Springer, 2010, pp. 10–30. DOI: 10.1007/978-3-642-18070-5_2.

[65] M. E. Farfoura et al. "A Blind Reversible Method for Watermarking Relational Databases Based on a Time-stamping Protocol". In: Expert Syst. Appl. 39.3 (2012), pp. 3185–3196. DOI: 10.1016/j.eswa.2011.09.005.

[66] P. Ferrara, F. Logozzo, and M. Fähndrich. "Safer unsafe code for .NET". In: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA. Ed. by G. E. Harris. ACM, 2008, pp. 329–346. DOI: 10.1145/1449764.1449791.

[67] D. Filaretti and S.Maffeis. "An Executable Formal Semantics of PHP". In: ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings. 2014, pp. 567–592. DOI: 10.1007/978-3-662-44202-9_23.

[68] G. Filé, R. Giacobazzi, and F. Ranzato. "A Unifying View of Abstract Domain Design". In: ACM Comput. Surv. 28.2 (1996), pp. 333–336. DOI: 10.1145/234528.234742.

[69] J. Franco-Contreras and G. Coatrieux. "Robust Watermarking of Relational Databases With Ontology-Guided Distortion Control". In: IEEE Trans. Inf. Forensics Secur. 10.9 (2015), pp. 1939–1952. DOI: 10.1109/TIFS.2015.2439962.

[70] J. Franco-Contreras et al. "Ontology-guided Distortion Control for Robust-lossless Database Watermarking: Application to inpatient hospital stay records". In: 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC 2014, Chicago, IL, USA, August 26-30, 2014. IEEE, 2014, pp. 4491–4494. DOI: 10.1109/EMBC.2014.6944621.

[71] X. Fu et al. "A Static Analysis Framework For Detecting SQL Injection Vulnerabilities". In: 31st Annual International Computer Software and Applications Conference, COMPSAC 2007, Beijing, China, July 24-27, 2007. Volume 1. 2007, pp. 87–96. DOI: 10.1109/COMPSAC.2007.43.

[72] J. Fulara et al. "Relational Abstract Domain of Weighted Hexagons". In: Electron. Notes Theor. Comput. Sci. 267.1 (2010), pp. 59–72. DOI: 10.1016/j.entcs.2010.09.006.

[73] G. Gange et al. "Abstract Interpretation over Non-lattice Abstract Domains". In: Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 6–24. DOI: 10.1007/978-3-642-38856-9_3.

[74] H. Gericke. "Alfred Tarski. A Lattice-theoretical Fixpoint Theorem and its Applications". In: Journal of Symbolic Logic. Pacific journal of mathematics, Bd. 5 (1955) 22.4 (1957), 285—309. DOI: 10.2307/2963937.

[75] R. Giacobazzi and I. Mastroeni. "Making Abstract Models Complete". In: Math. Struct. Comput. Sci. 26.4 (2016), pp. 658–701. DOI: 10.1017/S0960129514000358.

[76] R. Giacobazzi and I. Mastroeni. "Transforming Abstract Interpretations by Abstract Interpretation". In: Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings. Ed. by M. Alpuente and G. Vidal. Vol. 5079. Lecture Notes in Computer Science. Springer, 2008, pp. 1–17. DOI: 10.1007/978-3-540-69166-2_1.

[77] R. Giacobazzi and E. Quintarelli. "Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking". In: Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings. Ed. by P. Cousot. Vol. 2126. Lecture Notes in Computer Science. Springer, 2001, pp. 356–373. DOI: 10.1007/3-540-47764-0_20.

[78] R. Giacobazzi and F. Ranzato. "Functional Dependencies and Moore-Set Completions of Abstract Interpretations and Semantics". In: Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon, USA, December 4-7, 1995. Ed. by J. W. Lloyd. MIT Press, 1995, pp. 321–335.

[79] R. Giacobazzi and F. Ranzato. "Refining and Compressing Abstract Domains". In: Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings. 1997, pp. 771–781. DOI: 10.1007/3-540-63165-8_230.

[80] R. Giacobazzi, F. Ranzato, and F. Scozzari. "Making Abstract Interpretations Complete". In: J. ACM 47.2 (2000), pp. 361–416. DOI: 10.1145/333979.333989.

[81] M. L. P. Gort, E. A. Díaz, and C. F. Uribe. "A Highly-Reliable Virtual Primary Key Scheme for Relational Database Watermarking Techniques". In: 2017 International Conference on Computational Science and Computational Intelligence (CSCI). IEEE. 2017, pp. 55–60. DOI: 10.1109/CSCI.2017.10..

[82] M. L. P. Gort, C. F. Uribe, and I. Nummenmaa. "A Minimum Distortion: High Capacity Watermarking Technique for Relational Data". In: Proceedings of the 5th ACM Workshop on Information Hiding and Multimedia Security. ACM. 2017, pp. 111–121. DOI: 10.1145/3082031.3083241.

[83] M. L. P. Gort et al. "HQR-Scheme: A High Quality and Resilient Virtual Primary Key Generation Approach for Watermarking Relational Data". In: Expert Systems with Applications 138 (2019), p. 112770. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2019.06.058.

[84] M. L. P. Gort et al. "Preventing Additive Attacks to Relational Database Watermarking". In: Research and Practical Issues of Enterprise Information Systems - 13th IFIP WG 8.9 International Conference, CONFENIS 2019, Prague, Czech Republic, December 16-17, 2019, Proceedings. Ed. by P. Doucek et al. Vol. 375. Lecture Notes in Business Information Processing. Springer, 2019, pp. 131–140. DOI: 10.1007/978-3-030-37632-1_12.

[85] M. L. P. Gort et al. "Semantic-driven Watermarking of Relational Textual Databases". In: Expert Systems with Applications (2020), p. 114013. DOI: 10.1016/j.eswa.2020.114013.

[86] C. Gould, Z. Su, and P. Devanbu. "JDBC Checker: a Static Analysis Tool for SQL/JDBC Applications". In: Proceedings. 26th International Conference on Software Engineering. IEEE, 2004. DOI: 10.1109/ICSE.2004.1317494.

[87] P. Granger. "Improving the Results of Static Analyses of Programs by Local Decreasing Iterations". In: Shyamasundar R. (eds) Foundations of Software Technology and Theoretical Computer Science. FSTTCS 1992. Lecture Notes in Computer Science, vol 652. Springer, Berlin, Heidelberg. 1992. DOI: 0.1007/3-540-56287-7_95.

[88] S. Gulwani and A. Tiwari. "Combining Abstract Interpreters". In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2006, 376–386. DOI: 10.1145/1133981.1134026.

[89] J. Guo. "Fragile Watermarking Scheme for Tamper Detection of Relational Database". In: 2011 International Conference on Computer and Management (CAMAN). IEEE. 2011, pp. 1–4. DOI: 10.1109/CAMAN.2011.5778907.

[90] M. K. Gupta, M. C. Govil, and G. Singh. "Static Analysis Approaches to Detect SQL Injection and Cross Site Scripting Vulnerabilities in Web Applications: a Survey". In: International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014). 2014, pp. 1–5. DOI: 10.1109/ICRAIE.2014.6909173.

[91] S. Gupta and B. B. Gupta. "Cross-Site Scripting (XSS) Attacks and Defense Mechanisms: Classification and State-of-the-Art". In: Int. J. Systems Assurance Engineering and Management 8.1 (2017), pp. 512–530. DOI: 10.1007/s13198-015-0376-0.

[92] R. Halder, S. Pal, and A. Cortesi. "Watermarking Techniques for Relational Databases: Survey, Classification and Comparison". In: J. Univers. Comput. Sci. 16.21 (2010), pp. 3164–3190. DOI: 10.3217/jucs-016-21-3164.

[93] R. Halder, S. Pal, and A. Cortesi. "Watermarking Techniques for Relational Databases: Survey, Classification and Comparison". In: J. UCS 16.21 (2010), pp. 3164–3190. DOI: 10.3217/jucs-016-21-3164.

[94] W. G. J. Halfond and A. Orso. "AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks". In: 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA. 2005, pp. 174–183. DOI: 10.1145/1101908.1101935.

[95] W. G. J. Halfond, J. Viegas, and A. Orso. "A Classification of SQL Injection Attacks and Countermeasures". 2006.

[96] A. Hliaoutakis et al. "Information Retrieval by Semantic Similarity". In: Int. J. Semantic Web Inf. Syst. 2.3 (2006), pp. 55–73. DOI: 10.4018/jswis.2006070104.

[97] G. J. Holzmann. "UNO: Static Source Code Checking for UserDefined Properties". In: 6th World Conf. on Integrated Design and Process Technology, IDPT '02. 2002.

[98] J. M. Howe and A. King. "Logahedra: A New Weakly Relational Domain". In: Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings. 2009, pp. 306–320. DOI: 10.1007/978-3-642-04761-9_23.

[99] H.Seidl, R. Wilhelm, and S. Hack. "Compiler Design - Analysis and Transformation". Springer, 2012. DOI: 10.1007/978-3-642-17548-0.

[100] Y. W. Huang et al. "Securing Web Application Code by Static Analysis and Runtime Protection". In: Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004. 2004, pp. 40–52. DOI: 10.1145/988672.988679.

[101] IEEE. "IEEE Standard for Software Verification and Validation Plans". IEEE Std 1012-1986. 1986.

[102] H. Illous, M. Lemerre, and X. Rival. "A Relational Shape Abstract Domain". In: NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings. 2017, pp. 212–229. DOI: 10.1007/978-3-319-57288-8_15.

[103] J. R. Ingram. "Digital Piracy". In: The Encyclopedia of Criminology and Criminal Justice (2014). DOI: 10.1002/9781118517383.wbeccj116.

[104] Z. Jalil and A. M. Mirza. "A Review of Digital Watermarking Techniques for Text Documents". In: 2009 International Conference on Information and Multimedia Technology. Dec. 2009, pp. 230–234. DOI: 10.1109/ICIMT.2009.11.

[105] S. H. Jensen, A. Møller, and P. Thiemann. "Type Analysis for JavaScript". In: Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings. 2009, pp. 238–255. DOI: 10.1007/978-3-642-03237-0_17.

[106] C. Jiang, X. Chen, and Z. Li. "Watermarking Relational Databases for Ownership Protection Based on DWT". In: Proceedings of the Fifth International Conference on Information Assurance and Security, IAS 2009, Xi'An, China, 18-20 August 2009. IEEE Computer Society, 2009, pp. 305–308. DOI: 10.1109/IAS.2009.220.

[107] R. W. M. Jones and P. H. J. Kelly. "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs". In: Proceedings of the Third International Workshop on Automated Debugging, AADEBUG 1997, Linköping, Sweden, May 26-27, 1997. Vol. 2. Linköping Electronic Articles in Computer and Information Science 009. Linköping University Electronic Press, 1997, pp. 13–26.

[108] M. Journault, A. Miné, and A. Ouadjaout. "Modular Static Analysis of String Manipulations in C Programs". In: Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings. 2018, pp. 243–262. DOI: 10.1007/978-3-319-99725-4_16.

[109] N. Jovanovic, C. Kruegel, and E. Kirda. "Pixy: a Static Analysis Tool for Detecting Web Application Vulnerabilities". In: 2006 IEEE Symposium on Security and Privacy (S P'06). 2006, 6 pp.–263. DOI: 10.1109/SP.2006.29.

[110] N. S. Kamaruddin et al. "A Review of Text Watermarking: Theory, Methods, and Applications". In: IEEE Access 6 (2018), pp. 8011–8028. URL: https://doi.org/10.1109/ACCESS.2018.2796585.

[111] M. Kamran and M. Farooq. "A Formal Usability Constraints Model for Watermarking of Outsourced Datasets". In: IEEE Trans. Inf. Forensics Secur. 8.6 (2013), pp. 1061–1072. DOI: 10.1109/TIFS.2013.2259234.

[112] V. Kashyap et al. "JSAI: a Static Analysis Platform for JavaScript". In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. 2014, pp. 121–132. DOI: 10.1145/2635868.2635904.

[113] G. A. Kildall. "A Unified Approach to Global Program Optimization". In: Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA. 1973, pp. 194–206. DOI: 10.1145/512927.512945.

[114] H. Kim, K. Doh, and D. A. Schmidt. "Static Validation of Dynamically Generated HTML Documents Based on Abstract Parsing and Semantic Processing". In: Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Ed. by F. Logozzo and M. Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 194–214. DOI: 10.1007/978-3-642-38856-9_12.

[115] S. Kim et al. "Inferring Grammatical Summaries of String Values". In: Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings. 2014, pp. 372–391. DOI: 10.1007/978-3-319-12736-1_20.

[116] D. A. Kindy and A. K. Pathan. "A Survey on SQL Injection: Vulnerabilities, Attacks, and Prevention Techniques". In: 2011 IEEE 15th International Symposium on Consumer Electronics (ISCE). 2011, pp. 468–471. DOI: 10.1109/ISCE.2011.5973873.

[117] J. C. King. "Symbolic Execution and Program Testing". In: Commun. ACM 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[118] E. Kneuss, P. Suter, and V. Kuncak. "Phantm: PHP analyzer for type mismatch". In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010. 2010, pp. 373–374. DOI: 10.1145/1882291.1882355.

[119] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California, Mar. 2004. DOI: 10.1109/CGO.2004.1281665.

[120] H. Lauko, P. Rockai, and J. Barnat. "Symbolic Computation via Program Transformation". In: Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings. 2018, pp. 313–332. DOI: 10.1007/978-3-030-02508-3_17.

[121] H. Lauko et al. "Abstracting Strings for Model Checking of C Programs". In: Appl. Sci., 10(21), 7853; (2020). DOI: 10.3390/app10217853.

[122] H. Lee et al. "SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript". In: Proceedings of the 19th International Workshop on Foundations of Object-Oriented Languages (FOOL'12). 2012.

[123] T. Liang et al. "An Efficient SMT Solver for String Constraints". In: Formal Methods in System Design 48.3 (2016), pp. 206–234. DOI: 10.1007/s10703-016-0247-6.

[124] V. B. Livshits and M. S. Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis". In: Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005. Ed. by P. D. McDaniel. USENIX Association, 2005.

[125] F. Logozzo. "Towards a Quantitative Estimation of Abstract Interpretations". In: Workshop on Quantitative Analysis of Software. Microsoft, June 2009.

[126] F. Logozzo and M. Fähndrich. "Pentagons: a Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses". In: Sci. Comput. Program. 75.9 (2010), pp. 796–807. DOI: 10.1016/j.scico.2009.04.004.

[127] M. Madsen and E. Andreasen. "String Analysis for Dynamic Field Access". In: Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. 2014, pp. 197–217. DOI: 10.1007/978-3-642-54807-9_12.

[128] S. Maffeis, J. C. Mitchell, and A. Taly. "An Operational Semantics for JavaScript". In: Programming Languages and Systems, 6th Asian Symposium, AP-LAS 2008, Bangalore, India, December 9-11, 2008. Proceedings. 2008, pp. 307–325. DOI: 10.1007/978-3-540-89330-1_22.

[129] MathWorks. "Polyspace". http://www.mathworks.com/products/polyspace.html. Last check: November 23rd, 2020.

[130] B. B. Mehta and H. D. Aswar. "Watermarking for Security in Database: a Review". In: 2014 Conference on IT in Business, Industry and Government (CSIBIG) (2014), pp. 1–6. DOI: 10.1109/CSIBIG.2014.7056938.

[131] S. Melkundi and C. Chandankhede. "A Robust Technique for Relational Database Watermarking and Verification". In: Communication, Information & Computing Technology (ICCICT), 2015 International Conference on. IEEE. 2015, pp. 1–7. DOI: 10.1109/ICCICT.2015.7045676.

[132] G. A. Miller. "WordNet: An Electronic Lexical Database". MIT press, 1998.

[133] Y. Minamide. "Static Approximation of Dynamically Generated Web Pages". In: Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005. Ed. by A. Ellis and T. Hagino. ACM, 2005, pp. 432–441. DOI: 10.1145/1060745.1060809.

[134] A. Miné. "Field-sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics". In: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Ontario, Canada, June 14-16, 2006. 2006, pp. 54–63. DOI: 10.1145/1134650.1134659.

[135] A. Miné. "The Octagon Abstract Domain". In: High. Order Symb. Comput. 19.1 (2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1.

[136] A. Møller and M. I. Schwartzbach. "Static Program Analysis". Department of Computer Science, Aarhus University, 2015.

[137] F. Nielson. "Tensor Products Generalize the Relational Data Flow Analysis Method". In: Fourth Hungarian Computer Science Conference. 1985, pp. 211–225.

[138] F. Nielson and H. R. Nielson. "Type and Effect Systems". In: Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel). Ed. by E. R. Olderog and B. Steffen. Vol. 1710. Lecture Notes in Computer Science. Springer, 1999, pp. 114–136. DOI: 10.1007/3-540-48092-7_6.

[139] F. Nielson, H. R. Nielson, and C. Hankin. "Principles of program analysis". Springer, 1999. DOI: 10.1007/978-3-662-03811-6.

[140] Aleph One. "Smashing The Stack For Fun And Profit". Phrack Magazine. 1996.

[141] OWASP. "OWASP Top Ten". https://owasp.org/www-project-top-ten/. Last check: November 25th, 2020.

[142] C. Park, H. Im, and S. Ryu. "Precise and Scalable Static Analysis of jQuery Using a Regular Expression Domain". In: Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016. 2016, pp. 25–36. DOI: 10.1145/2989225.2989228.

[143] E. G. M. Petrakis et al. "X-Similarity: Computing Semantic Similarity between Concepts from Different Ontologies". In: J. Digit. Inf. Manag. 4.4 (2006), pp. 233–237.

[144] A. Di Pierro and H. Wiklicky. "Measuring the Precision of Abstract Interpretations". In: Logic Based Program Synthesis and Transformation, 10th International Workshop, LOPSTR 2000 London, UK, July 24-28, 2000, Selected Papers. 2000, pp. 147–164. DOI: 10.1007/3-540-45142-0_9.

[145] M. Pradel and K. Sen. "The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript". In: 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic. Ed. by J. T. Boyland. Vol. 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 519–541. DOI: 10.4230/LIPIcs.ECOOP.2015.519.

[146] M. Dalla Preda and M. Pasqua. "Software Watermarking: A Semantics-based Approach". In: Electr. Notes Theor. Comput. Sci. 331 (2017), pp. 71–85. DOI: 10.1016/j.entcs.2017.02.005.

[147] Princeton-University. "About WordNet. WordNet. Princeton University". 2010. URL: http://wordnet.princeton.edu.

[148] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: Trans. Amer. Math. Soc. 74 (1953), pp. 358 –366.

[149] X. Rival and L. Mauborgne. "The Trace Partitioning Abstract Domain". In: ACM Trans. Program. Lang. Syst. 29.5 (2007), p. 26. DOI: 10.1145/1275497.1275501.

[150] H. Samimi et al. "Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving". In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. 2012, pp. 277–287. DOI: 10.1109/ICSE.2012.6227186.

[151] H. M. Sardroudi and S. Ibrahim. "A New Approach for Relational Database Watermarking Using Image". In: 5th International Conference on Computer Sciences and Convergence Information Technology. IEEE. 2010, pp. 606–610. DOI: 10.1109/ICCIT.2010.5711126.

[152] P. Saxena et al. "A Symbolic Execution Framework for JavaScript". In: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. 2010, pp. 513–528. URL: https://doi.org/10.1109/SP.2010.38.

[153] D. A. Schmidt. "Denotational Semantics: A Methodology for Language Development". USA: William C. Brown Publishers, 1986. ISBN: 0697068492.

[154] R. C. Seacord. "Secure Coding in C and C++". 2nd. Addison-Wesley Professional, 2013. ISBN: 0321822137.

[155] N. Seco, T. Veale, and J. Hayes. "An Intrinsic Information Content Metric for Semantic Similarity in WordNet". In: Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004. Ed. by R. L. De Mántaras and L. Saitta. IOS Press, 2004, pp. 1089–1090.

[156] H. Shahriar and M. Zulkernine. "Classification of Static Analysis-Based Buffer Overflow Detectors". In: Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, June 9-11, 2010 - Companion Volume. 2010, pp. 94–101. DOI: 10.1109/SSIRI-C.2010.28.

[157] J. Shanmugam and M. Ponnavaikko. "Cross Site Scripting-Latest Developments and Solutions: a Survey". In: Int. J. Open Problems Compt. Math 1 (2008).

[158] P. Singh and R. S. Chadha. "A Survey of Digital Watermarking Techniques, Applications and Attacks". In: 2013.

[159] T. Slimani. "Description and Evaluation of Semantic Similarity Measures Approaches". In: International Journal of Computer Applications 80.10 (Oct. 2013), 25–33.

[160] P. Sotin. "Quantifying the Precision of Numerical Abstract Domains". 2010.

[161] A. Sotirov. "Automatic Vulnerability Detection Using Static Source Code Analysis". 2005.

[162] F. Spoto. "The Julia Static Analyzer for Java". In: Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Ed. by X. Rival. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016, pp. 39–57. DOI: 10.1007/978-3-662-53413-7_3.

[163] SQLi. "SQL Injection". https://www.owasp.org/index.php/SQL_Injection. Last check: December 18th, 2018.

[164] Z. Su and G. Wassermann. "The Essence of Command Injection Attacks in Web Applications". In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006. 2006, pp. 372–382. DOI: 10.1145/1111037.1111070.

[165] J. Sun, Z. Cao, and Z. Hu. "Multiple Watermarking Relational Databases Using Image". In: MultiMedia and Information Technology, 2008. MMIT'08. International Conference on. IEEE. 2008, pp. 373–376. DOI: 10.1109/MMIT.2008.211.

[166] M. A. H. Taieb, M. B. Aouicha, and A. B. Hamadou. "A New Semantic Relatedness Measurement Using WordNet Features". In: Knowl. Inf. Syst. 41.2 (2014), pp. 467–497. DOI: 10.1007/s10115-013-0672-4.

[167] T. Tateishi, M. Pistoia, and O. Tripp. "Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic". In: ACM Trans. Softw. Eng. Methodol. 22.4 (2013), 33:1–33:33. DOI: 10.1145/2522920.2522926.

[168] TIOBE. "TIOBE Index for November 2020". https://www.tiobe.com/tiobe-index/. Last check: November 25th, 2020.

[169] U. Topkara, M. Topkara, and M. J. Atallah. "The Hiding Virtues of Ambiguity: Quantifiably Resilient Watermarking of Natural Language Text Through Synonym Substitutions". In: Proceedings of the 8th workshop on Multimedia & Security, MM&Sec 2006, Geneva, Switzerland, September 26-27, 2006. Ed. by S. Voloshynovskiy, J. Dittmann, and J. J. Fridrich. ACM, 2006, pp. 164–174. DOI: 10.1145/1161366.1161397.

[170] O. Tripp, P. Ferrara, and M. Pistoia. "Hybrid Security Analysis of Web JavaScript Code Via Dynamic Partial Evaluation". In: International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014. 2014, pp. 49–59. DOI: 10.1145/2610384.2610385.

[171] F. Vasilescu, P. Langlais, and G. Lapalme. "Evaluating Variants of the Lesk Approach for Disambiguating Words". In: Proceedings of the Fourth International Conference on Language Resources and Evaluation, LREC 2004, May 26-28, 2004, Lisbon, Portugal. European Language Resources Association, 2004.

[172] M. Veanes, P. De Halleux, and N. Tillmann. "Rex: Symbolic Regular Expression Explorer". In: Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010. 2010, pp. 498–507. DOI: 10.1109/ICST.2010.15.

[173] J. Viega et al. "Token-based Scanning of Source Code for Security Problems". In: ACM Trans. Inf. Syst. Secur. 5.3 (2002), pp. 238–261. DOI: 10.1145/545186.545188.

[174] P. Vogt et al. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis". In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007. The Internet Society, 2007.

[175] D. A. Wagner et al. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities". In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2000, San Diego, California, USA. The Internet Society, 2000.

[176] G. Wassermann and Z. Su. "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities". In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007. 2007, pp. 32–41. DOI: 10.1145/1250734.1250739.

[177] G. Wassermann and Z. Su. "Static Detection of Cross-Site Scripting Vulnerabilities". In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008. 2008, pp. 171–180. DOI: 10.1145/1368088.1368112.

[178] G. Wassermann et al. "Static Checking of Dynamically Generated Queries in Database Applications". In: ACM Trans. Softw. Eng. Methodol. 16.4 (2007). DOI: 10.1145/1276933.1276935.

[179] R. Wilhelm, S. Sagiv, and T. W. Reps. "Shape Analysis". In: Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings. 2000, pp. 1–17. DOI: 10.1007/3-540-46423-9_1.

[180] K. Winstein. "Lexical Steganography through Adaptive Modulation of the Word Choice Hash". Secondary education at the Illinois Mathematics and Science Academy. 1999.

[181] Y. Xie, A. Chou, and D. R. Engler. "ARCHER: Using Symbolic, Path-sensitive Analysis to Detect Memory Access Errors". In: Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003. 2003, pp. 327–336. DOI: 10.1145/940071.940115.

[182] XSS. "Cross Site Scripting". https://www.owasp.org/index.php/Cross-site_Scripting_(XSS). Last check: November 23rd, 2020.

[183] R. Xu, P. Godefroid, and R. Majumdar. "Testing for Buffer Overflows with Length Abstraction". In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008. 2008, pp. 27–38. DOI: 10.1145/1390630.1390636.

[184] F. Yu, M. Alkhalaf, and T. Bultan. "Patching Vulnerabilities with Sanitization Synthesis". In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. 2011, pp. 251–260. DOI: 10.1145/1985793.1985828.

[185] F. Yu, T. Bultan, and B. Hardekopf. "String Abstractions for String Verification". In: Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings. 2011, pp. 20–37. DOI: 10.1007/978-3-642-22306-8_3.

[186] F. Yu, T. Bultan, and O. H. Ibarra. "Symbolic String Verification: Combining String Analysis and Size Analysis". In: Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 322–336. DOI: 10.1007/978-3-642-00768-2_28.

[187] F. Yu et al. "Automata-based Symbolic String Analysis for Vulnerability Detection". In: Formal Methods Syst. Des. 44.1 (2014), pp. 44–70. DOI: 10.1007/s10703-013-0189-1.

[188] F. Yu et al. "Symbolic String Verification: An Automata-Based Approach". In: Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings. 2008, pp. 306–324. DOI: 10.1007/978-3-540-85114-1_21.

[189] L. Zhang et al. "Relational databases watermarking for textual and numerical data". In: Aug. 2011, pp. 1633–1636. DOI: 10.1109/MEC.2011.6025791.

[190] Y. Zhang et al. "A Method of Verifying Relational Databases Ownership with Image Watermark". In: The 6th International Symposium on Test and Measurement, Dalian, PR China. 2005, pp. 6316–6319.

[191] Z. Zhang et al. "Watermarking Relational Database Using Image". In: Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No. 04EX826). Vol. 3. IEEE. 2004, pp. 1739–1744. DOI: 10.1109/ICMLC.2004.1382056.

[192] X. Zhou et al. "Security Theory and Attack Analysis for Text Watermarking". In: 2009 International Conference on E-Business and Information System Security. May 2009, pp. 1–6. DOI: 10.1109/EBISS.2009.5138072.