


Article

Abstracting Strings for Model Checking of C Programs

Henrich  Lauko ^{1,*}, Martina Olliaro ^{1,2,*}, Agostino Cortesi ² and Petr Ročkal ¹¹ Faculty of Informatics, Masaryk University, Botanická 68A, 60200 Brno, Czech Republic; xrockai@fi.muni.cz² Scientific Campus, Ca' Foscari University of Venice, Via Torino 155, Mestre, 30172 Venice, Italy; cortesi@unive.it

* Correspondence: xlauko@mail.muni.cz (H.L.); martina.olliaro@unive.it (M.O.)

Received: 30 September 2020; Accepted: 2 November 2020; Published: 5 November 2020



Abstract: Data type abstraction plays a crucial role in software verification. In this paper, we introduce a domain for abstracting strings in the C programming language, where strings are managed as null-terminated arrays of characters. The new domain M-String is parametrized on an index (bound) domain and a character domain. By means of these different constituent domains, M-Strings captures shape information on the array structure as well as value information on the characters occurring in the string. By tuning these two parameters, M-String can be easily tailored for specific verification tasks, balancing precision against complexity. The concrete and the abstract semantics of basic operations on strings are carefully formalized, and soundness proofs are fully detailed. Moreover, for a selection of functions contained in the standard C library, we provide the semantics for character access and update, enabling an automatic lifting of arbitrary string-manipulating code into our new domain. An implementation of abstract operations is provided within a tool that automatically lifts existing programs into the M-String domain along with an explicit-state model checker. The accuracy of the proposed domain is experimentally evaluated on real-case test programs, showing that M-String can efficiently detect real-world bugs as well as to prove that program does not contain them after they are fixed.

Keywords: string analysis; model checking; abstract interpretation; abstract domain

1. Introduction

C is still one of the mainly used programming languages [1], and a large portion of systems of critical relevance are written in this language, such as server-side software and 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 [2]. 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 [3].

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.

1.1. Paper Contribution

This paper is a revised and extended version of [4,5]. 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 [6]. 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 [6], 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 [7], 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 [8], 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.

1.2. Paper Structure

In the following Section 2 we give basics in abstract interpretation and we introduce the array segmentation abstract domain [6] on which M-String is based. Furthermore, Section 3 introduces the syntax of some operations of interest. Section 4 defines the concrete domain and semantics. Section 5 presents the M-String abstract domain for C character arrays and its semantics, whose soundness is formally proved. In the Section 6, we present a general approach to abstraction as a program transformation and extend it to abstraction of program strings. Sections 7 and 8 present implementation and evaluation details of M-String abstraction. In Section 9 we discuss related work. Finally, Section 10 concludes.

2. Prerequisites

We assume the reader is familiar with order theory.

2.1. Abstract Interpretation

Abstract Interpretation [9,10] is a theory about sound approximation or *abstraction* of semantics of computer programs, focusing on some run-time properties of interest. Formally, the concrete semantics is based on a concrete domain \mathbf{D} . Likewise, the abstract semantics is based on an abstract domain $\overline{\mathbf{D}}$. Both the concrete and the abstract domains form a complete lattice, such that: $(\mathbf{D}, \leq_{\mathbf{D}}, \perp_{\mathbf{D}}, \top_{\mathbf{D}}, \sqcup_{\mathbf{D}}, \sqcap_{\mathbf{D}})$ and $(\overline{\mathbf{D}}, \leq_{\overline{\mathbf{D}}}, \perp_{\overline{\mathbf{D}}}, \top_{\overline{\mathbf{D}}}, \sqcup_{\overline{\mathbf{D}}}, \sqcap_{\overline{\mathbf{D}}})$. Please note that we use the same notation interchangeably to denote a domain and its set of elements. The concrete and the abstract domains are related by a pair of monotonic

functions: the concretization $\gamma_{\bar{\mathbf{D}}} : \bar{\mathbf{D}} \rightarrow \mathbf{D}$ and the abstraction $\alpha_{\bar{\mathbf{D}}} : \mathbf{D} \rightarrow \bar{\mathbf{D}}$ functions. In order to obtain a sound analysis, $\alpha_{\bar{\mathbf{D}}}$ and $\gamma_{\bar{\mathbf{D}}}$ have to form a Galois Connection (GC) [11]. $(\alpha_{\bar{\mathbf{D}}}, \gamma_{\bar{\mathbf{D}}})$ is a GC if and only if for every $\mathbf{d} \in \mathbf{D}$ and $\bar{\mathbf{d}} \in \bar{\mathbf{D}}$ we have that $\mathbf{d} \leq_{\mathbf{D}} \gamma_{\bar{\mathbf{D}}}(\bar{\mathbf{d}}) \Leftrightarrow \alpha_{\bar{\mathbf{D}}}(\mathbf{d}) \leq_{\bar{\mathbf{D}}} \bar{\mathbf{d}}$. Notice that, one function univocally identifies the other. Consequently, we can infer a Galois Connection by proving that $\gamma_{\bar{\mathbf{D}}}$ is a complete meet morphism (resp. $\alpha_{\bar{\mathbf{D}}}$ is a complete join morphism) (Proposition 7 of [12]). Please note that these conditions can be relaxed, performing abstract interpretation over non-lattice abstract domains [12]. Abstract domains that do not respect the Ascending Chain Condition (ACC) need to be equipped with a widening $\nabla_{\bar{\mathbf{D}}}$ and a narrowing $\blacksquare_{\bar{\mathbf{D}}}$ operator, in order to get fast convergence and to improve the accuracy of the resulting analysis, respectively [13]. An abstract domain functor $\bar{\mathbf{D}}$ is a function from the parameter abstract domains $\bar{\mathbf{D}}_1, \bar{\mathbf{D}}_2, \dots, \bar{\mathbf{D}}_n$ to a new abstract domain $\bar{\mathbf{D}}(\bar{\mathbf{D}}_1, \bar{\mathbf{D}}_2, \dots, \bar{\mathbf{D}}_n)$. The abstract domain functor $\bar{\mathbf{D}}(\bar{\mathbf{D}}_1, \bar{\mathbf{D}}_2, \dots, \bar{\mathbf{D}}_n)$ composes abstract domain properties of the parameter abstract domains to build a new class of abstract properties and operations [6].

2.2. Fun Array

In the following we recall the array segmentation analysis presented in [6]. Notice that we slightly modified the notation to be consistent with the whole work. For more details, we invite the reader to refer directly to the original paper.

2.2.1. Array Concrete Semantics

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

- $\theta(a) = (\rho, \text{low}_a, \text{high}_a, A_a)$ and,
- $\theta(a) \in \mathbf{A} = R_v \times \mathbb{E} \times \mathbb{E} \times (\mathbb{Z} \rightarrow (\mathbb{Z} \times V))$

where

1. R_v is the set of concrete variable environments. A concrete variable environment $\rho \in R_v$ maps variables (of basic types) $x \in \mathbb{X}$ to their values $\rho(x) \in V$.
2. \mathbb{E} is the set of program expressions made up of constants, variables, mathematical unary and binary operators. In the following, for simplicity, expressions are evaluated to integers. $\text{low}_a, \text{high}_a \in \mathbb{E}$ are expressions whose value, given by $\llbracket \text{low}_a \rrbracket \rho$ and $\llbracket \text{high}_a \rrbracket \rho$, respectively represents the lower bound and the upper bound of an array a , i.e., the lower and the upper bound of its indexes range. According to the denotational semantics approach, in [6] the value of an arithmetic expression e is denoted by $\llbracket e \rrbracket \rho$, 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, $\llbracket x \rrbracket \rho$ is equivalent to $\rho(x)$, with $x \in \mathbb{X}$, and $\llbracket n \rrbracket \rho$, where n is a constant, is equivalent to n itself. Thus, for example, if e is the expression $x - 1$, its semantics $\llbracket x - 1 \rrbracket \rho$ is defined as $\llbracket x \rrbracket \rho - \llbracket 1 \rrbracket \rho$, which corresponds to $\rho(x) - 1$. 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 V is the set of values. Let I_a be the set of indexes i of an array a , i.e., $I_a = \{i \mid i \in [\llbracket \text{low}_a \rrbracket \rho, \llbracket \text{high}_a \rrbracket \rho]\} \subseteq \mathbb{Z}$ and, let P_a be the set of pairs (i, v) such that v is the value of the element indexed by i in an array a , i.e., $P_a = \{(i, v) \mid i \in I_a \wedge \llbracket a[i] \rrbracket \rho = v \in V\} \subseteq \mathbb{Z} \times V$. Thus, $A_a : I_a \rightarrow P_a$ is a function mapping the indexes of an array a to their corresponding pairs (index, indexed array value).

Example 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 $\theta(a) = (\rho, 0, 5, A_a)$, where the value of the lower and the upper bound of a are clear from the context and the codomain of the function A_a is the set $P_a = \{(0, 5), (1, 7), (2, 9), (3, 11), (4, 13)\}$. Moreover, let b

denote the sub-array of \mathbf{a} from position 2 to 3 included, its concrete value is given by $\theta(\mathbf{b}) = (\rho, 2, 4, A_{\mathbf{b}})$ such that $P_{\mathbf{b}} = \{(2, 9), (3, 11)\}$.

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

2.2.2. Array Segmentation Abstract Domain Functor

According to [6], the FunArray abstract domain $\bar{\mathbf{S}}$ (shortcut for $\bar{\mathbf{S}}(\bar{\mathbf{B}}, \bar{\mathbf{A}}, \bar{\mathbf{R}})$) 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 2. Consider the integer array $\mathbf{a}[5] = \{5, 7, 9, 10, 12\}$. As an abstraction of \mathbf{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 $\bar{\mathbf{S}} = \{(\bar{\mathbf{B}} \times \bar{\mathbf{A}}) \times (\bar{\mathbf{B}} \times \bar{\mathbf{A}} \times \{_,?\})^k \times (\bar{\mathbf{B}} \times \{_,?\}) \mid k \geq 0\} \cup \{\perp_{\bar{\mathbf{S}}}\}$, which have the form $\bar{\mathbf{b}}_1 \bar{\mathbf{p}}_1 \bar{\mathbf{b}}_2[?_2] \bar{\mathbf{p}}_2 \dots \bar{\mathbf{p}}_{n-1} \bar{\mathbf{b}}_n[?_n]$ where

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

We denote by $\bar{\mathbf{E}}$ the set of expressions of canonical form $\bar{x} + k$, where $\bar{x} \in \bar{\mathbf{X}}$ and $k \in \mathbb{Z}$. The segment bounds $\bar{\mathbf{b}}_i$ are sets of expressions $\{\bar{e}_i^1, \dots, \bar{e}_i^m\}$, such that $\bar{e}_i^j \in \bar{\mathbf{E}}$. The variable abstract domain $\bar{\mathbf{X}}$ encodes program variables, i.e., $\bar{\mathbf{X}} = \mathbb{X} \cup \{v_0\}$, where v_0 is a special variable whose value is assumed to be zero. Moreover, $\bar{\mathbf{b}}_i = \emptyset$ denotes unreachability; if $\bar{\mathbf{b}}_i \neq \emptyset$, 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 \bar{e}_i^j is a constant). Thus, $\bar{\mathbf{B}}$ depends on the expression abstract domain $\bar{\mathbf{E}}$ which, in turn, depends on the variable abstract domain $\bar{\mathbf{X}}$.

2. $\bar{\mathbf{A}}$ is the array element abstract domain, with abstract properties $\bar{\mathbf{p}}_i \in \bar{\mathbf{A}}$. It denotes possible values of pairs (index, indexed array element) in a segment, for relational abstractions, array elements otherwise.
3. $\bar{\mathbf{R}}$ is the variable environment abstract domain, which depends on the variable abstract domain $\bar{\mathbf{X}}$, with abstract properties $\bar{\rho} \in \bar{\mathbf{R}}$.
4. the question mark, if present, expresses the possibility that the segment that precedes it may be empty. The question mark can never precede $\bar{\mathbf{b}}_1$. The space symbol $_$ in $\{_,?\}$ represents a non-empty segment.

Example 3. Let $\bar{\mathbf{A}}$ 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.

Two segmentations, $\bar{\mathbf{b}}_1^1 \dots \bar{\mathbf{b}}_n^1[?_n^1]$ and $\bar{\mathbf{b}}_1^2 \dots \bar{\mathbf{b}}_n^2[?_n^2]$, are compatible if $\bar{\mathbf{b}}_1^1 \cap \bar{\mathbf{b}}_1^2 \neq \emptyset$ and $\bar{\mathbf{b}}_n^1 \cap \bar{\mathbf{b}}_n^2 \neq \emptyset$. The unification algorithm, in [6], modifies two compatible segmentations 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 terminate and it is deterministic. The partial order $\sqsubseteq_{\bar{\mathbf{S}}}$ over $\bar{\mathbf{S}}$ is defined over unified segmentations as well as the join $\sqcup_{\bar{\mathbf{S}}}$ and the meet $\sqcap_{\bar{\mathbf{S}}}$ operators. Please note that $\bar{\mathbf{S}}$ is not necessarily a lattice [14]. Moreover, $\bar{\mathbf{S}}$ does not respect the Ascending Chain

Condition, therefore, in order to ensure the convergence of the analysis, it is equipped with a widening operator ∇_{ξ} . A narrowing operator which improves the precision of the widening result, is also defined. Widening and narrowing operators are applied 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. Syntax

Strings in the programming language C are arrays of characters, whose length is determined by 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 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.

Table 1. String functions syntax in C.

<code>char *strcat(char *str1, const char *str2)</code>
<code>char *strchr(char *str, int c)</code>
<code>int strcmp(const char *str1, const char *str2)</code>
<code>char *strcpy(char *str1, const char *str2)</code>
<code>size_t strlen(const char *str)</code>

- `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 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 [15]. 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.

4. Concrete Domain and Semantics

Our aim is to capture the presence of well-formed strings in \mathbb{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 is also given.

4.1. Character Array Concrete Semantics

Let \mathbb{C} be a finite set of characters representable by the character encoding in use equipped with a top element $\top_{\mathbb{C}}$ representing an unknown value and let \mathbb{M} be the set of character array variables, such that $\mathbb{M} \subseteq \mathbb{A}$ (with \mathbb{A} being the set of array variables - of any type - presented in Section 2.2). Then, the operational semantics of character array variables are concrete array environments $\mu \in R_m$ mapping character arrays $m \in \mathbb{M}$ to their values $\mu(m)$. Precisely:

- $\mu(m) = (\rho, \text{low}_m, \text{high}_m, \mathcal{M}_m, N_m)$ and,
- $\mu(m) \in \mathbf{M} = R_v \times \mathbb{E} \times \mathbb{E} \times (\mathbb{Z} \rightarrow (\mathbb{Z} \times \mathbb{C})) \times \mathbb{Z}$

so that R_m is a map from \mathbb{M} to \mathbf{M} , where R_v and \mathbb{E} are the concrete variable environment and the expression domain defined in Section 2.2 respectively, \mathbb{Z} is the integer domain and \mathbb{C} is the character set introduced above. Notice that with respect to the concrete array environment θ introduced in Section 2.2, the function μ returns as a last component the set of indexes which map to the string terminating characters $N_m = \{i \mid i \in I_m \wedge \mathcal{M}_m(i) = (i, '\0')\}$, with I_m being the domain of the function \mathcal{M}_m . On the other hand, \mathcal{M}_m behaves exactly as \mathcal{A}_a in $\theta(a)$, 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} (c.f., Section 2.2) by adding a parameter that takes into account the presence of null characters in a character array. For well-formed strings, N_m cannot be empty. Moreover, character array elements which have not been initialized are mapped to the top value $\top_{\mathbb{C}}$ as they may be values already present in the memory assigned to the locations array itself.

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

4.1.1. 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 1 (string of interest). Let $m \in \mathbb{M}$ be an array of characters with concrete value $\mu(m) = (\rho, \text{low}_m, \text{high}_m, \mathcal{M}_m, N_m)$ and let z be the minimum element of N_m (if it is non-empty). The string of interest of the character array described by $\mu(m)$ is defined as follows:

$$\text{string}(\mu(m)) = \begin{cases} \langle v_i : i \in [\llbracket \text{low}_m \rrbracket \rho, z] \wedge \mathcal{M}_m(i) = (i, v) \rangle & \text{if } N_m \neq \emptyset \\ \text{undef} & \text{otherwise} \end{cases}$$

with v_i denoting the character value which occurs in the pair (i, v) .

Example 5. Consider the concrete character array value of Example 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).

4.2. Concrete Domain

As a concrete domain for array of characters we refer to the complete lattice $\mathcal{P}(\mathbf{M})$ defined as $(\mathcal{P}(\mathbf{M}), \subseteq_{\mathcal{P}(\mathbf{M})}, \perp_{\mathcal{P}(\mathbf{M})}, \top_{\mathcal{P}(\mathbf{M})}, \cup_{\mathcal{P}(\mathbf{M})}, \cap_{\mathcal{P}(\mathbf{M})})$ 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 $\perp_{\mathcal{P}(\mathbf{M})}$ 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.

4.3. Concrete Semantics

To formalize the concrete semantics of the C standard library functions from `string.h` introduced in Section 3, the following auxiliary functions *embedding*, *extraction*, *comparison* and *substitution* over single concrete character array values need to be introduced.

Definition 2 (embedding). Let $\mu(m_1), \mu(m_2) \in \mathbf{M}$ be two concrete character array values and $[l_1, u_1] \subseteq [[\text{low}_{m_1}]]\rho, [[\text{high}_{m_1}]]\rho, [l_2, u_2] \subseteq [[\text{low}_{m_2}]]\rho, [[\text{high}_{m_2}]]\rho$ be two indexes ranges of the same length. The function $\text{embedding}(\mu(m_1), [l_1, u_1], \mu(m_2), [l_2, u_2])$ embeds the sequence of characters of $\mu(m_2)$ which occurs from the index l_2 to the index u_2 into $\mu(m_1)$ from the index l_1 to the index u_1 . Formally, $\text{embedding}(\mu(m_1), [l_1, u_1], \mu(m_2), [l_2, u_2]) = \mu(m_1)'$ such that:

- $[[\text{low}_{m_1'}]]\rho = [[\text{low}_{m_1}]]\rho$ and $[[\text{high}_{m_1'}]]\rho = [[\text{high}_{m_1}]]\rho$
- $\mathcal{M}_{m_1'}$:
 - * $\forall i \in [[\text{low}_{m_1'}]]\rho, l_1): \mathcal{M}_{m_1'}(i) = (i, v)$ s.t. $k = i$ and $\mathcal{M}_{m_1}(k) = (k, v)$
 - * $\forall i \in [l_1, u_1]: \mathcal{M}_{m_1'}(i) = (i, v)$ s.t. $k = l_2 + (i - l_1)$ and $\mathcal{M}_{m_2}(k) = (k, v)$
 - * $\forall i \in (l_1, [[\text{high}_{m_1'}]]\rho): \mathcal{M}_{m_1'}(i) = (i, v)$ s.t. $k = i$ and $\mathcal{M}_{m_1}(k) = (k, v)$
- $N_{m_1'} = (N_{m_1} \setminus \{i \mid i \in [l_1, u_1]\}) \cup \{i \mid i \in [l_1, u_1] \wedge k = l_2 + (i - l_1) \wedge \mathcal{M}_{m_2}(k) = (k, '\backslash 0')\}$

Example 6. Let $\mu(m_1) = (\rho, 0, 7, \mathcal{M}_{m_1}, N_{m_1})$ and $\mu(m_2) = (\rho, 0, 6, \mathcal{M}_{m_2}, N_{m_2})$ be two concrete character array values such that:

- $P_{m_1} = \{(0, 'a'), (1, 'a'), (2, 'a'), (3, '\backslash 0'), (4, 'a'), (5, 'a'), (6, 'a')\}$
- $P_{m_2} = \{(0, 'b'), (1, 'b'), (2, 'b'), (3, 'b'), (4, 'b'), (5, '\backslash 0')\}$

Moreover, consider the intervals of equal length:

- $[2, 4]_{m_1} \subseteq [[\text{low}_{m_1}]]\rho, [[\text{high}_{m_1}]]\rho$
- $[3, 5]_{m_2} \subseteq [[\text{low}_{m_2}]]\rho, [[\text{high}_{m_2}]]\rho$

The function $\text{embedding}(\mu(m_1), [2, 4]_{m_1}, \mu(m_2), [3, 5]_{m_2}) = \mu(m_1)'$ where:

- $[[\text{low}_{m_1'}]]\rho = [[\text{low}_{m_1}]]\rho = 0$ and $[[\text{high}_{m_1'}]]\rho = [[\text{high}_{m_1}]]\rho = 7$

Algorithm 1 Lexicographic comparison of concrete character array values.

Function: $comparison(\mu(m_1), \mu(m_2))$

Input: two concrete character array values $\mu(m_1), \mu(m_2) \in \mathbf{M}$ such that:

- both N_{m_1} and N_{m_2} are different from the emptyset and,
- for $i_1 \in [\llbracket low_{m_1} \rrbracket \rho, \min(N_{m_1})], i_2 \in [\llbracket low_{m_2} \rrbracket \rho, \min(N_{m_2})]$: $\mathcal{M}(i_1) \neq (i_1, \top_c)$ and $\mathcal{M}(i_2) \neq (i_2, \top_c)$

Output: an integer value n .

```

1:  $n = 0, i_1 = \llbracket low_{m_1} \rrbracket \rho, i_2 = \llbracket low_{m_2} \rrbracket \rho$ 
2: while  $i_1 \in [\llbracket low_{m_1} \rrbracket \rho, \min(N_{m_1})]$  and  $i_2 \in [\llbracket low_{m_2} \rrbracket \rho, \min(N_{m_2})]$  do
3:    $n = v_{i_1} \top_c v_{i_2}$ 
4:   if  $n \neq 0$  then
5:     return  $n$ 
6:   else
7:      $i_1 = i_1 + 1$ 
8:      $i_2 = i_2 + 1$ 
9: return  $n$ 

```

- $P_{m'_1} = \{(0, 'a'), (1, 'a'), (2, 'b'), (3, 'b'), (4, '\0'), (5, 'a'), (6, 'a')\}$
- $N_{m'_1} = \{4\}$

Definition 3 (extraction). Let $\mu(m) \in \mathbf{M}$ be a concrete character array value and $[l, u] \subseteq [\llbracket low_m \rrbracket \rho, \llbracket high_m \rrbracket \rho]$ be an indexes range. The function $extraction(\mu(m), [l, u])$ extracts the sequence of characters which occurs in $\mu(m)$ from the index l to the index u . Formally, $extraction(\mu(m), [l, u]) = \mu(m)'$ such that:

- $\llbracket low_{m'} \rrbracket \rho = l$ and $\llbracket high_{m'} \rrbracket \rho = u + 1$
- $\mathcal{M}_{m'} : \forall i \in [\llbracket low_{m'} \rrbracket \rho, \llbracket high_{m'} \rrbracket \rho]: \mathcal{M}_{m'}(i) = (i, v)$ s.t. $k = i$ and $\mathcal{M}_m(k) = (k, v)$
- $N_{m'} = N_m \setminus \{i \mid i \notin [l, u]\}$

Example 7. Let $\mu(m_1)$ be the character array concrete value of Example 6 and $[1, 3]_{m_1} \subseteq [\llbracket low_{m_1} \rrbracket \rho, \llbracket high_{m_1} \rrbracket \rho]$ be an indexes range of $\mu(m_1)$. The function $extraction(\mu(m_1), [1, 3]_{m_1}) = \mu(m_1)'$ such that:

- $\llbracket low_{m'_1} \rrbracket \rho = 1$ and $\llbracket high_{m'_1} \rrbracket \rho = 4$
- $P_{m'_1} = \{(1, 'a'), (2, 'a'), (3, '\0')\}$
- $N_{m'_1} = \{3\}$

Definition 4 (comparison). Let $\mu(m_1), \mu(m_2) \in \mathbf{M}$ be two concrete character array values which contain a fully initialized well-formed string of interest, i.e., no \top_c occurs. The function $comparison(\mu(m_1), \mu(m_2))$ (c.f. Algorithm 1) lexicographically compares the strings of interest of $\mu(m_1)$ and $\mu(m_2)$ and it returns an integer value n which denotes the lexicographic distance between them.

Notice that n will be strictly smaller than zero if $string(\mu(m_1))$ precedes $string(\mu(m_2))$ in lexicographic order, equal to zero if $string(\mu(m_1))$ and $string(\mu(m_2))$ are lexicographically equivalent, and strictly greater than zero if $string(\mu(m_1))$ follows $string(\mu(m_2))$ in lexicographic order.

Example 8. Let $\mu(m_1)$ and $\mu(m_2)$ be the character array concrete values of Example 6. Both of them contain a fully initialized well-formed string of interest and the function $comparison(\mu(m_1), \mu(m_2))$ computes the lexicographic distance between them. Precisely, the procedure stops after the first iteration of the for loop (c.f. Algorithm 1) and, assuming ASCII as the character encoding set, it returns the value -1 , i.e., $n = 97 - 98$, which means that $string(\mu(m_1))$ lexicographically precedes $string(\mu(m_2))$.

Definition 5 (substitution). Let $\mu(m) \in \mathbf{M}$ be a concrete character array value, $z \in [\llbracket \text{low}_m \rrbracket \rho, \llbracket \text{high}_m \rrbracket \rho]$ be an index and $c \in \mathbb{C}$ be a character. The function $\text{substitution}(\mu(m), z, c)$ substitutes the character which appears in $\mu(m)$ at the index z with the character c . Formally, $\text{substitution}(\mu(m), z, c) = \mu(m)'$ such that:

- $\llbracket \text{low}_{m'} \rrbracket \rho = \llbracket \text{low}_m \rrbracket \rho$ and $\llbracket \text{high}_{m'} \rrbracket \rho = \llbracket \text{high}_m \rrbracket \rho$
- $\mathcal{M}_{m'}$:
 - * $\forall i \in [\llbracket \text{low}_{m'} \rrbracket \rho, z): \mathcal{M}_{m'}(i) = (i, v)$ s.t. $k = i$ and $\mathcal{M}_m(k) = (k, v)$
 - * for $i = z: \mathcal{M}_{m'}(z) = (z, c)$
 - * $\forall i \in (z, \llbracket \text{high}_{m'} \rrbracket \rho): \mathcal{M}_{m'}(i) = (i, v)$ s.t. $k = i$ and $\mathcal{M}_m(k) = (k, v)$
- $N_{m'} = \begin{cases} N_m & \text{if } (z \in N_m \wedge c \text{ is null}) \vee (z \notin N_m \wedge c \text{ is not null}) \\ N_m \setminus \{z\} & \text{if } z \in N_m \wedge c \text{ is not null} \\ N_m \cup \{z\} & \text{otherwise} \end{cases}$

Example 9. Let $\mu(m_1)$ be the character array concrete value of Example 6, the index z be equal to 4 and the character c be the null termination '\0'. The function $\text{sub}(\mu(m_1), 4, '\0') = \mu(m_1)'$ such that:

- $\llbracket \text{low}_{m'_1} \rrbracket \rho = 0$ and $\llbracket \text{high}_{m'_1} \rrbracket \rho = 7$
- $P_{m'_1} = \{(0, 'a'), (1, 'a'), (2, 'a'), (3, '\0'), (4, '\0'), (5, 'a'), (6, 'a')\}$
- $N_{m'_1} = \{3, 4\}$

4.3.1. Array Access

The semantics operator \mathfrak{A} , given the statement access_j and a set of concrete character array values M in $\mathcal{P}(\mathbf{M})$ as parameter, returns a value in \mathbb{C} . In particular, $\text{access}_j(M)$ returns the character v which occurs at position j if all the character array values in M 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 M ; otherwise it returns $\top_{\mathbb{C}}$. Formally,

$$\mathfrak{A}[\llbracket \text{access}_j \rrbracket](M) = \begin{cases} v & \text{if } \forall \mu(m) \in M : j \in [\llbracket \text{low}_m \rrbracket \rho, \llbracket \text{high}_m \rrbracket \rho] \text{ and } \mathcal{M}_m(j) = (j, v) \\ \top_{\mathbb{C}} & \text{otherwise} \end{cases}$$

4.3.2. String Concatenation

The semantics 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 $\text{strcat}(M_1, M_2)$, it returns all the possible embeddings in M_1 of a string of interest taken from M_2 if all the character array values (which belong to both M_1 and M_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,

$$\mathfrak{M}[\llbracket \text{strcat} \rrbracket](M_1, M_2) = \begin{cases} M'_1 & \text{if } \forall \mu(m_1) \in M_1 : \forall \mu(m_2) \in M_2 : \text{string}(\mu(m_1)) \neq \text{undef} \neq \text{string}(\mu(m_2)) \\ & \wedge \text{size.condition is true} \\ \top_{\mathcal{P}(\mathbf{M})} & \text{otherwise} \end{cases}$$

The `size.condition is true` if:

$$(\llbracket \text{high}_{m_1} \rrbracket \rho - \llbracket \text{low}_{m_1} \rrbracket \rho) \geq [(\min(N_{m_1}) - \llbracket \text{low}_{m_1} \rrbracket \rho - 1) + (\min(N_{m_2}) - \llbracket \text{low}_{m_2} \rrbracket \rho)]$$

Moreover, M'_1 is the set of *embedding*($\mu(m_1), [l_1, u_1], \mu(m_2), [l_2, u_2]$) (c.f. Definition 2), such that:

- $\mu(m_1) \in M_1, l_1 = \min(N_{m_1})$ and $u_1 = l_1 + (\min(N_{m_2}) - \llbracket \text{low}_{m_2} \rrbracket \rho)$
- $\mu(m_2) \in M_2, l_2 = \llbracket \text{low}_{m_2} \rrbracket \rho$ and $u_2 = \min(N_{m_2})$

4.3.3. String Character

The semantics operator \mathfrak{M} , when applied to $\text{strchr}_v(M)$, returns the set of string of interest suffixes in M from the index corresponding to the first occurrence of the character v if all the character array values in M contain a well-formed string containing v . Otherwise, if all the character array values in M contain a well-formed string in which does not occur the character v , it returns the emptyset (denoted by $\perp_{\mathcal{P}(M)}$); otherwise it returns $\top_{\mathcal{P}(M)}$. Formally,

$$\mathfrak{M}[\llbracket \text{strchr}_v \rrbracket](M) = \begin{cases} S & \text{if } \forall \mu(m) \in M : \text{string}(\mu(m)) \neq \text{undef} \text{ and } v \in \text{string}(\mu(m)) \\ \perp_{\mathcal{P}(M)} & \text{if } \forall \mu(m) \in M : \text{string}(\mu(m)) \neq \text{undef} \text{ and } v \notin \text{string}(\mu(m)) \\ \top_{\mathcal{P}(M)} & \text{otherwise} \end{cases}$$

In particular, S is the set of $\text{extraction}(\mu(m), [l, u])$ (c.f. Definition 3), such that:

- $\mu(m) \in M, l = \min(\{i : i \in [\llbracket \text{low}_m \rrbracket \rho, \min(N_m)] \wedge \mathcal{M}_m(i) = (i, v)\})$ and $u = \min(N_m)$

4.3.4. String Compare

The semantics operator \mathfrak{P} , given the statement strcmp and two sets of concrete character array values M_1, M_2 in $\mathcal{P}(M)$ 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, $\text{strcmp}(M_1, M_2)$ returns an integer value n which denotes the lexicographic distance between strings of interest in M_1 and M_2 if for all $\mu(m_1) \in M_1$ and $\mu(m_2) \in M_2$ the procedure $\text{comparison}(\mu(m_1), \mu(m_2))$ (c.f. Definition 4) returns n ; otherwise it returns $\top_{\mathbb{Z}}$. Formally,

$$\mathfrak{P}[\llbracket \text{strcmp} \rrbracket](M_1, M_2) = \begin{cases} n & \text{if } \forall \mu(m_1) \in M_1 : \forall \mu(m_2) \in M_2 : \text{comparison}(\mu(m_1), \mu(m_2)) = n \\ \top_{\mathbb{Z}} & \text{otherwise} \end{cases}$$

4.3.5. String Copy

The semantics operator \mathfrak{M} , when applied to $\text{strcpy}(M_1, M_2)$, behaves similarly to the string concatenation function above. Formally,

$$\mathfrak{M}[\llbracket \text{strcpy} \rrbracket](M_1, M_2) = \begin{cases} M'_1 & \text{if } \forall \mu(m_1) \in M_1 : \forall \mu(m_2) \in M_2 : \text{string}(\mu(m_1)) \neq \text{undef} \neq \text{string}(\mu(m_2)) \\ & \wedge \text{size.condition is true} \\ \top_{\mathcal{P}(M)} & \text{otherwise} \end{cases}$$

The `size.condition` is true if:

$$(\llbracket \text{high}_{m_1} \rrbracket \rho - \llbracket \text{low}_{m_1} \rrbracket \rho) \geq (\min(N_{m_2}) - \llbracket \text{low}_{m_2} \rrbracket \rho)$$

Moreover, M'_1 is the set of $\text{embedding}(\mu(m_1), [l_1, u_1], \mu(m_2), [l_2, u_2])$, such that:

- $\mu(m_1) \in M_1, l_1 = \llbracket \text{low}_{m_1} \rrbracket \rho$ and $u_1 = l_1 + (\min(N_{m_2}) - \llbracket \text{low}_{m_2} \rrbracket \rho)$
- $\mu(m_2) \in M_2, l_2 = \llbracket \text{low}_{m_2} \rrbracket \rho$ and $u_2 = \min(N_{m_2})$

4.3.6. String Length

The semantics operator \mathfrak{L} , given the statement strlen and a set of concrete character array values M in $\mathcal{P}(M)$ 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, $\text{strlen}(M)$ 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 M if all the character array values in M contain a well-formed string of the same length; otherwise it returns $\top_{\mathbb{Z}}$. Formally,

$$\mathcal{L}[\text{strlen}](M) = \begin{cases} n & \text{if } \forall \mu(m) \in M : \text{string}(\mu(m)) \neq \text{undef} \wedge (\text{min}(N_m) - \llbracket \text{low}_m \rrbracket \rho) = n \\ \perp_Z & \text{otherwise} \end{cases}$$

4.3.7. Array Update

The semantics operator \mathfrak{M} , when applied to $\text{update}_{j,v}(M)$, returns the set of character array values in M 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 M ; otherwise it returns $\perp_{\mathcal{P}(M)}$. Formally,

$$\mathfrak{M}[\text{update}_{j,v}](M) = \begin{cases} M' & \text{if } \forall \mu(m) \in M : j \in [\llbracket \text{low}_m \rrbracket \rho, \llbracket \text{high}_m \rrbracket \rho) \\ \perp_{\mathcal{P}(M)} & \text{otherwise} \end{cases}$$

In particular, M' is the set of $\text{substitution}(\mu(m), j, v)$ (c.f. Definition 2).

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 $\mathcal{P}(M)$, 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 $\mathcal{P}(M)$, and its semantics for which soundness is proved.

5.1. Character Array Abstract Domain

The M-String (\overline{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 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 $\overline{M}(\overline{B}, \overline{C}, \overline{R})$ where

1. \overline{B} denotes the abstraction of segment bounds, equipped with the addition ($+\overline{b}$) and subtraction ($-\overline{b}$) operations.
2. \overline{C} is the abstraction of the character array elements, it is signed, it contains the value 0, and it is equipped with is_null , a special monotonic function lifting abstract elements in \overline{C} to a value in the set $\{\text{true}, \text{false}, \text{maybe}\}$ and with subtraction ($-\overline{c}$).
3. \overline{R} denotes the abstraction of scalar variable environments (cf. Section 2.2). Namely, the constant propagation domain on the set of variables \mathbb{X} .

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

- \overline{M}_s corresponds to $\{(\overline{B} \times \overline{C}) \times \{\overline{B} \times \overline{C} \times \{_, ?\}\}^k \times \{\overline{B} \times \{_, ?\}\} \mid k \geq 0\} \cup \{\overline{B}\} \cup \{\emptyset\}$ and it represents the segmentation of the strings of interest of a set of character arrays.
- \overline{M}_{ns} corresponds to $\{(\overline{B} \times \overline{C}) \times \{\overline{B} \times \overline{C} \times \{_, ?\}\}^k \times \{\overline{B} \times \{_, ?\}\} \mid k \geq 0\} \cup \{\emptyset\}$ 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.
- $\perp_{\overline{M}}, \top_{\overline{M}}$ are special elements denoting the bottom/top element of \overline{M} .

The elements in \overline{M} are split segmentation abstract predicates of the form $\overline{m} = (s, ns)$. For instance, when \overline{m} is equal to $(\overline{b}_1, \emptyset)$, it abstracts concrete character array values of length 1 and containing a null string of interest (c.f. Section 4.1.1). On the other hand, when \overline{m} is equal to $(\overline{b}_1, \overline{b}_2 \overline{p}_2 \overline{b}_3 [?_3] \dots \overline{b}_n [?_n])$, it approximates concrete character array values of length greater than or equal to 1 containing a null string of interest. In particular:

1. $\bar{b}_i \in \bar{\mathbf{B}}$ denotes the segment bounds, chosen in abstract domain $\bar{\mathbf{B}}$, such that $i \in [1, n]$ and $n > 1$. 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 $\bar{\mathbf{B}}$ is let free.
 For the sake of readability, we apply arithmetic operators on \bar{b}_i directly. For instance, $\bar{b}_{+_{\bar{\mathbf{B}}}} 1$ should be read as $\alpha_{\bar{\mathbf{B}}}(\{i + 1 \mid i \in \gamma_{\bar{\mathbf{B}}}(\bar{b}_i)\})$ or $\bar{b}_1 +_{\bar{\mathbf{B}}} \bar{b}_2$ as $\alpha_{\bar{\mathbf{B}}}(\{i_1 + i_2 \mid i_1 \in \gamma_{\bar{\mathbf{B}}}(\bar{b}_1) \wedge i_2 \in \gamma_{\bar{\mathbf{B}}}(\bar{b}_2)\})$, where $\alpha_{\bar{\mathbf{B}}}$ and $\gamma_{\bar{\mathbf{B}}}$ are respectively the abstraction and concretization functions over the bounds abstract domain.
 Please note that \bar{b}_1 and \bar{b}_n respectively represent the segmentation lower and upper bound and in the case in which $\bar{\mathbf{m}}$ corresponds to the split segmentation $(\bar{b}_1 \bar{p}_1 \bar{b}_2 [?_2] \bar{p}_2 \bar{b}_3 [?_3] \dots \bar{b}_{n-1} [?_{n-1}], \emptyset)$ the segmentation upper bound is hidden, due to a representative choice, and equal to $\bar{b}_{n-1} +_{\bar{\mathbf{B}}} 1$. Moreover, in a segmentation $\dots \bar{b}_i [?_i] \bar{p}_{i+1} \bar{b}_{i+1} [?_{i+1}] \dots$ we always assume that $\min(\gamma_{\bar{\mathbf{B}}}(\bar{b}_{i+1})) > \max(\gamma_{\bar{\mathbf{B}}}(\bar{b}_i))$.
2. $\bar{p}_i \in \bar{\mathbf{C}}$ are abstract predicates, chosen in an abstract domain $\bar{\mathbf{C}}$, 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 [6], non-empty segments are not marked.

Example 10. Consider the split segmentation abstract predicate $\bar{\mathbf{m}} = ([0, 0] \text{ 'a' } [2, 5], \emptyset)$ where $\bar{\mathbf{C}}$ is the constant propagation domain for characters and $\bar{\mathbf{B}}$ the interval domain. $\bar{\mathbf{m}}$ 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 "aaaa\0".

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

M-String, like FunArray, is equipped with join $\sqcup_{\bar{\mathbf{M}}}$, meet $\sqcap_{\bar{\mathbf{M}}}$, widening $\nabla_{\bar{\mathbf{M}}}$ and narrowing $\blacksquare_{\bar{\mathbf{M}}}$ operators (c.f. Section 2.2.2). We highlight the fact that the choice of $\bar{\mathbf{B}}$ is let free, so the segmentation unification algorithm presented in [6] needs to be modified accordingly, while preserving its original requirements. The unify procedure behaves as follows: given $\bar{\mathbf{m}}_1, \bar{\mathbf{m}}_2 \in \bar{\mathbf{M}}$, $\text{unify}(\bar{\mathbf{m}}_1, \bar{\mathbf{m}}_2)$ results into the pair $\text{unify}(\bar{\mathbf{m}}_1.s, \bar{\mathbf{m}}_2.s)$ and $\text{unify}(\bar{\mathbf{m}}_1.ns, \bar{\mathbf{m}}_2.ns)$, where $\bar{\mathbf{m}}_1.s$ and $\bar{\mathbf{m}}_2.s$ (resp. $\bar{\mathbf{m}}_1.ns$ and $\bar{\mathbf{m}}_2.ns$) are compatible, leading to two abstract predicates $(\bar{\mathbf{m}}'_1.s, \bar{\mathbf{m}}'_1.ns)$ and $(\bar{\mathbf{m}}'_2.s, \bar{\mathbf{m}}'_2.ns)$, respectively. Given two split segmentations $\bar{\mathbf{m}}_1$ and $\bar{\mathbf{m}}_2$, let $\text{low}_{\bar{\mathbf{m}}_1.s}$ and $\text{high}_{\bar{\mathbf{m}}_1.s}$ (resp. $\text{low}_{\bar{\mathbf{m}}_2.s}$ and $\text{high}_{\bar{\mathbf{m}}_2.s}$) denote the lower and upper bounds of $\bar{\mathbf{m}}_1.s$ (resp. $\bar{\mathbf{m}}_2.s$). $\bar{\mathbf{m}}_1.s$ and $\bar{\mathbf{m}}_2.s$ are compatible if $\text{low}_{\bar{\mathbf{m}}_1.s} \sqcap_{\bar{\mathbf{B}}} \text{low}_{\bar{\mathbf{m}}_2.s} \neq \perp_{\bar{\mathbf{B}}}$ and $\text{high}_{\bar{\mathbf{m}}_1.s} \sqcap_{\bar{\mathbf{B}}} \text{high}_{\bar{\mathbf{m}}_2.s} \neq \perp_{\bar{\mathbf{B}}}$. The same apply to $\bar{\mathbf{m}}_1.ns$ and $\bar{\mathbf{m}}_2.ns$. Definitions 6 and 7 present how the join and the meet operators over $\bar{\mathbf{M}}$ are computed. The widening and narrowing can be easily derived.

Example 11. Consider the following split segmentations: $\bar{\mathbf{m}}_1 = ([0, 0] \text{ odd } [2, 4] \text{ even } [7, 7], \emptyset)$ and $\bar{\mathbf{m}}_2 = ([0, 0] \text{ odd } [1, 2] \top_{\text{parity}} [3, 6] \text{ even } [7, 7], \emptyset)$. Their unification leads to the abstract elements $\bar{\mathbf{m}}'_1 = ([0, 0] \text{ odd } [2, 4] \text{ even } [7, 7], \emptyset)$ and $\bar{\mathbf{m}}'_2 = ([0, 0] \text{ odd } [2, 2] \top_{\text{parity}} [7, 7], \emptyset)$. Observe that the unify yields to a pair of segmentations with the same number of segments and that is not always optimal.

Definition 6 (M-String join). $\sqcup_{\bar{\mathbf{M}}}$ represents the join operator that defines a minimal upper bound between two abstract elements. Let $\text{unify}(\bar{\mathbf{m}}_1, \bar{\mathbf{m}}_2) = \bar{\mathbf{m}}'_1, \bar{\mathbf{m}}'_2$, then $\bar{\mathbf{m}}'_1 \sqcup_{\bar{\mathbf{M}}} \bar{\mathbf{m}}'_2 = (\bar{\mathbf{m}}'_1.s \sqcup_{\bar{\mathbf{M}}} \bar{\mathbf{m}}'_2.s, \bar{\mathbf{m}}'_1.ns \sqcup_{\bar{\mathbf{M}}} \bar{\mathbf{m}}'_2.ns)$ such that:

- $\bar{\mathbf{m}}'_1.s \sqcup_{\bar{\mathbf{M}}} \bar{\mathbf{m}}'_2.s = \bar{b}_1^{-1} \sqcup_{\bar{\mathbf{B}}} \bar{b}_1^{-2} \bar{p}_1^{-1} \sqcup_{\bar{\mathbf{C}}} \bar{p}_1^{-2} \bar{b}_2^{-1} \sqcup_{\bar{\mathbf{B}}} \bar{b}_2^{-2} [?_2^1] \vee [?_2^2] \dots \bar{b}_k^{-1} \sqcup_{\bar{\mathbf{B}}} \bar{b}_k^{-2} [?_k^1] \vee [?_k^2]$
- $\bar{\mathbf{m}}'_1.ns \sqcup_{\bar{\mathbf{M}}} \bar{\mathbf{m}}'_2.ns = \bar{b}_{k+1}^{-1} \sqcup_{\bar{\mathbf{B}}} \bar{b}_{k+1}^{-2} \bar{p}_{k+1}^{-1} \sqcup_{\bar{\mathbf{C}}} \bar{p}_{k+1}^{-2} \bar{b}_{k+2}^{-1} \sqcup_{\bar{\mathbf{B}}} \bar{b}_{k+2}^{-2} [?_{k+2}^1] \vee [?_{k+2}^2] \dots \bar{b}_n^{-1} \sqcup_{\bar{\mathbf{B}}} \bar{b}_n^{-2} [?_n^1] \vee [?_n^2]$

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

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

Definition 7 (M-String meet). $\sqcap_{\overline{\mathbf{M}}}$ represents the meet operator that defines a maximal lower bound between abstract elements.

Let $\text{unify}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2) = \overline{\mathbf{m}}'_1, \overline{\mathbf{m}}'_2$, then $\overline{\mathbf{m}}'_1 \sqcap_{\overline{\mathbf{M}}} \overline{\mathbf{m}}'_2 = (\overline{\mathbf{m}}'_1.s \sqcap_{\overline{\mathbf{M}}} \overline{\mathbf{m}}'_2.s, \overline{\mathbf{m}}'_1.ns \sqcap_{\overline{\mathbf{M}}} \overline{\mathbf{m}}'_2.ns)$ such that:

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

if $\overline{\mathbf{m}}'_1.s$ and $\overline{\mathbf{m}}'_2.s$ (resp. $\overline{\mathbf{m}}'_1.ns$ and $\overline{\mathbf{m}}'_2.ns$) are compatible; $\perp_{\overline{\mathbf{M}}}$ 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 $\{_,?\}$, respectively. In particular, $\sqcup \curlywedge \sqcup = \sqcup \curlywedge ? = ? \curlywedge \sqcup = \sqcup$ and $? \curlywedge ? = ?$.

5.1.1. Abstraction

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

5.1.2. 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: $\gamma_{\overline{\mathbf{M}}}(\perp_{\overline{\mathbf{M}}}) = \emptyset$, 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 [6], which corresponds to the set of character array values whose elements in the segment $[\overline{\mathbf{b}}, \overline{\mathbf{b}}'[?])$ satisfy the predicate $\overline{\mathbf{p}}$.

$$\gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{b}}\overline{\mathbf{p}}\overline{\mathbf{b}}'[?])\overline{\rho} \triangleq \{(\rho, \text{low}, \text{high}, \mathcal{M}, \mathbf{N}) \mid \rho \in \gamma_{\overline{\mathbf{R}}}(\overline{\rho}) \wedge \forall \mathbf{b}, \mathbf{b}' : \mathbf{b} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}), \mathbf{b}' \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}') \wedge \llbracket \text{low} \rrbracket \rho \leq \mathbf{b} \leq \mathbf{b}' \leq \llbracket \text{high} \rrbracket \rho \wedge \forall i \in [\mathbf{b}, \mathbf{b}'] : \mathcal{M}(i) \in \gamma_{\overline{\mathbf{C}}}(\overline{\mathbf{p}}) \wedge \mathbf{N} = \{i \mid \mathcal{M}(i) = (i, '\setminus 0')\}\}$$

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

We remind that the upper bound of $\overline{\mathbf{m}}.s$ is not followed by a segment abstract predicate. Let $\overline{\mathbf{b}}$ be the upper bound of $\overline{\mathbf{m}}.s$ (which may coincide with the lower bound of $\overline{\mathbf{m}}.s$ 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 $\overline{\mathbf{b}}' = \overline{\mathbf{b}} +_{\overline{\mathbf{B}}} 1$ 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 $\overline{\mathbf{m}}.s$ and $\overline{\mathbf{m}}.ns$ belonging to a character array abstract predicate $\overline{\mathbf{m}}$, i.e., $\gamma_{\overline{\mathbf{M}}}^* \in \overline{\mathbf{M}} \rightarrow \overline{\mathbf{R}} \rightarrow \mathcal{P}(\mathbf{M})$. Let $+_{\mathbf{M}}$ denote the concatenation of several concrete values.

$$\begin{aligned} &\gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{m}}.s)\overline{\rho} \\ &\triangleq \{(\rho, \text{low}, \text{high}, \mathcal{M}, \mathbf{N}) \in \bigoplus_{i=1}^k \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{b}}_i\overline{\mathbf{p}}_i\overline{\mathbf{b}}_{i+1}[?_{i+1}])\overline{\rho} \mid \forall \mathbf{b}_1, \mathbf{b}_k : \mathbf{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1), \\ &\hspace{15em} \mathbf{b}_k \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_k) \wedge \mathbf{b}_1 = \llbracket \text{low} \rrbracket \rho \wedge \mathbf{b}_k + 1 \leq \llbracket \text{high} \rrbracket \rho\} \\ &\text{if } \overline{\mathbf{m}}.s = \overline{\mathbf{b}}_1\overline{\mathbf{p}}_1\overline{\mathbf{b}}_2[?_2] \dots \overline{\mathbf{b}}_{k-1}[?_{k-1}]\overline{\mathbf{p}}_{k-1}\overline{\mathbf{b}}_k[?_k] \\ &\triangleq \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{b}}_1)\overline{\rho} \\ &\text{if } \overline{\mathbf{m}}.s = \overline{\mathbf{b}}_1 \\ &\triangleq \emptyset \\ &\text{otherwise} \\ &\gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{m}}.ns)\overline{\rho} \end{aligned}$$

$$\begin{aligned}
 &\triangleq \{(\rho, \text{low}, \text{high}, \mathcal{M}, \mathcal{N}) \in \bigoplus_{i=1}^{n-1} \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{b}}_i \overline{\mathbf{p}}_i \overline{\mathbf{b}}_{i+1} [?_{i+1}]) \overline{\rho} \mid \forall \mathbf{b}_1, \mathbf{b}_n : \mathbf{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1), \mathbf{b}_n \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_n) \wedge \\
 &\hspace{15em} \mathbf{b}_1 = \llbracket \text{low} \rrbracket \rho \wedge \mathbf{b}_n = \llbracket \text{high} \rrbracket \rho\} \\
 &\text{if } \overline{\mathbf{m}}.ns = \overline{\mathbf{b}}_1 \overline{\mathbf{p}}_1 \overline{\mathbf{b}}_2 [?_2] \dots \overline{\mathbf{b}}_n [?_n] \\
 &\triangleq \{(\rho, \text{low}, \text{high}, \mathcal{M}, \mathcal{N}) \in \bigoplus_{i=k+1}^{n-1} \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{b}}_i \overline{\mathbf{p}}_i \overline{\mathbf{b}}_{i+1} [?_{i+1}]) \overline{\rho} \mid \forall \mathbf{b}_{k+1}, \mathbf{b}_n : \mathbf{b}_{k+1} \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_{k+1}), \\
 &\hspace{15em} \mathbf{b}_n \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_n) \wedge \llbracket \text{low} \rrbracket \rho < \mathbf{b}_{k+1} \wedge \mathbf{b}_n = \llbracket \text{high} \rrbracket \rho\} \\
 &\text{if } \overline{\mathbf{m}}.ns = \overline{\mathbf{b}}_{k+1} \overline{\mathbf{p}}_{k+1} \overline{\mathbf{b}}_{k+2} [?_{k+2}] \dots \overline{\mathbf{b}}_n [?_n] \\
 &\triangleq \emptyset \\
 &\text{otherwise}
 \end{aligned}$$

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

$$\gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}) \overline{\rho} \triangleq \{(\rho, \text{low}, \text{high}, \mathcal{M}, \mathcal{N}) \in \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{m}}.s) \overline{\rho} +_{\mathbf{M}} \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{m}}.ns) \overline{\rho} \mid \forall \mathbf{b}_1, \mathbf{b}_n : \mathbf{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1), \\
 \mathbf{b}_n \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_n) \wedge \mathbf{b}_1 = \llbracket \text{low} \rrbracket \rho \wedge \mathbf{b}_n = \llbracket \text{high} \rrbracket \rho\}$$

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

Definition 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 $\perp_{\overline{\mathbf{C}}}$ and its upper bound $\overline{\mathbf{b}}'$ is not followed by a question mark.*

Theorem 1. *Let $\overline{\mathbf{X}} \subseteq \overline{\mathbf{M}}$ such that all elements in $\overline{\mathbf{X}}$ are compatible and their meet does not result in split segmentation abstract predicates which contain invalid abstract elements. The following inference chain holds:*

$$\begin{aligned}
 &\gamma_{\overline{\mathbf{M}}} \left(\prod_{\overline{\mathbf{m}} \in \overline{\mathbf{X}}} \overline{\mathbf{m}} \right) \\
 &= \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}') \quad \text{where } \overline{\mathbf{m}}' \text{ is the result of the meet operation over } \overline{\mathbf{X}} \text{ as defined in Definition 7} \\
 &= \{(\rho, \text{low}, \text{high}, \mathcal{M}, \mathcal{N}) \in \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{m}}'.s) \overline{\rho} +_{\mathbf{M}} \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{m}}'.ns) \overline{\rho} \mid \forall \mathbf{b}_1, \mathbf{b}_n : \mathbf{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1), \mathbf{b}_n \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_n) \wedge \\
 &\hspace{10em} \mathbf{b}_1 = \llbracket \text{low} \rrbracket \rho \wedge \mathbf{b}_n = \llbracket \text{high} \rrbracket \rho\} \quad \text{by definition of } \gamma_{\overline{\mathbf{M}}} \\
 &= \bigcap_{\overline{\mathbf{m}} \in \overline{\mathbf{X}}} \{(\rho, \text{low}, \text{high}, \mathcal{M}, \mathcal{N}) \in \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{m}}.s) \overline{\rho} +_{\mathbf{M}} \gamma_{\overline{\mathbf{M}}}^*(\overline{\mathbf{m}}.ns) \overline{\rho} \mid \forall \mathbf{b}_1, \mathbf{b}_n : \mathbf{b}_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_1), \mathbf{b}_n \in \gamma_{\overline{\mathbf{B}}}(\overline{\mathbf{b}}_n) \wedge \\
 &\hspace{10em} \mathbf{b}_1 = \llbracket \text{low} \rrbracket \rho \wedge \mathbf{b}_n = \llbracket \text{high} \rrbracket \rho\} \\
 &= \bigcap_{\overline{\mathbf{m}} \in \overline{\mathbf{X}}} \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}) \quad \text{by definition of } \gamma_{\overline{\mathbf{M}}}
 \end{aligned}$$

Observe that if the hypotheses of Theorem 1 are not satisfied, i.e., if either the abstract predicates in $\overline{\mathbf{X}}$ are not compatible or their meet leads to invalid segmentations, then $\gamma_{\overline{\mathbf{M}}} \left(\prod_{\overline{\mathbf{m}} \in \overline{\mathbf{X}}} \overline{\mathbf{m}} \right) = \gamma_{\overline{\mathbf{M}}}(\perp_{\overline{\mathbf{M}}}) = \emptyset$, and $\bigcap_{\overline{\mathbf{m}} \in \overline{\mathbf{X}}} \gamma_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}) = \emptyset$.

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

Definition 9 (lift). *Let $\mathbf{M} \subseteq \mathcal{P}(\mathbf{M})$ be a set of concrete character array values. Given the abstraction function $\alpha_{\overline{\mathbf{M}}}$ on M-String, we define the lift operation of \mathbf{M} as follows:*

$$\text{lift}(\mathbf{M}) = \alpha_{\overline{\mathbf{M}}}(\mathbf{M}).$$

Definition 10 (lower). *Let $\overline{\mathbf{m}}$ denote $\text{lift}(\mathbf{M})$ (c.f. Definition 9). Given the concretization function $\gamma_{\overline{\mathbf{M}}}$ on M-String, we define the lower operation of $\overline{\mathbf{m}}$ as follows:*

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

5.2. Abstract Semantics

Let us now formalize the abstract semantics of the concrete operations defined in Section 4.3, over the M-String domain. In doing so, we will take advantage of the auxiliary function *minlen* which computes the minimum length of an element $\bar{m} \in \bar{\mathbf{M}}$, as the upper bound of a split segmentation is possibly followed by a question mark.

Definition 11 (\bar{m} minimum length). Let $\bar{m} \in \bar{\mathbf{M}}$ different from $\perp_{\bar{\mathbf{M}}}$ and let $\text{low}_{\bar{m}}, \text{high}_{\bar{m}} \in \bar{\mathbf{B}}$ denote the lower and the upper bound of \bar{m} , respectively. We define the minimum length of a split segmentation abstract predicate \bar{m} , denoted by $\text{minlen}(\bar{m})$, as follows:

$$\text{minlen}(\bar{m}) = \begin{cases} \bar{b}_k \neg_{\bar{b}} \text{low}_{\bar{m}} & \text{if } \bar{m}.ns \neq \emptyset \wedge \text{high}_{\bar{m}} \text{ is followed by } ? \wedge \\ & \exists k \in \bar{m}.ns : k = \max\{i \in \bar{m}.ns \mid \bar{b}_i \text{ is not followed by } ?\} \\ \text{high}_{\bar{m}} \neg_{\bar{b}} \text{low}_{\bar{m}} & \text{otherwise} \end{cases}$$

Please note that in the second case of Definition 11, the minimum length of a split segmentation corresponds to its length, denoted by $\text{len}(\bar{m})$. The *len* operation can be also applied over the parameters of \bar{m} themselves, when they are different from the emptyset and their upper bound is not question marked, which is always the case with $\bar{m}.s$.

Example 12. Consider the split segmentation abstract predicate $\bar{m} = ([0, 0] \text{ 'a' } [2, 5], [3, 6] \text{ 'b' } [7, 7] \text{ 'c' } [8, 8]?)$ where $\bar{\mathbf{C}}$ is the constant propagation domain for characters and $\bar{\mathbf{B}}$ the interval domain. The minimum length of \bar{m} is given by $\text{minlen}(\bar{m}) = [7, 7] \neg_{\bar{b}} [0, 0] = [7, 7]$ as its upper bound is followed by a question mark. Logically speaking, the maximum length of \bar{m} is $[8, 8] \neg_{\bar{b}} [0, 0] = [8, 8]$. The length of $\bar{m}.s$ is given by $\text{len}(\bar{m}.s) = [2, 5] \neg_{\bar{b}} [0, 0] = [2, 5]$.

5.2.1. Abstract Array Access

The semantics operator $\mathfrak{A}_{\bar{\mathbf{M}}}$ is the abstract counterpart of \mathfrak{A} (c.f. Section 4.3.1). In particular, $\text{access}_{\bar{j}}(\bar{m})$ returns, if \bar{j} is valid for \bar{m} (i.e., there exist, and it is unique, a segment bounds interval $[\bar{b}_i[?_i], \bar{b}_{i+1})$ in \bar{m} to which \bar{j} belongs), the segment abstract predicate \bar{p}_i ; otherwise it returns $\top_{\bar{\mathbf{C}}}$. Formally,

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

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

5.2.2. Abstract String Concatenation

The semantics operator $\mathfrak{M}_{\bar{\mathbf{M}}}$ is the abstract counterpart of \mathfrak{M} . When applied to $\text{strcat}(\bar{m}_1, \bar{m}_2)$, it returns \bar{m}'_1 that is \bar{m}_1 into which $\bar{m}_2.s$ has been embedded starting from the upper bound of $\bar{m}_1.s$, 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 $\top_{\bar{\mathbf{M}}}$. Formally,

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

The *size.condition* is true if $\text{minlen}(\bar{m}_1) \geq_{\bar{b}} (\text{len}(\bar{m}_1.s) +_{\bar{b}} \text{len}(\bar{m}_2.s) +_{\bar{b}} 1)$. Let:

- $\bar{m}_1 = (\bar{b}_1 \bar{p}_1^1 \bar{b}_2[?_2^1] \dots \bar{p}_{k-1}^1 \bar{b}_k[?_k^1], \bar{b}_{k+1}^1 \bar{p}_{k+1}^1 \bar{b}_{k+2}[?_{k+2}^1] \dots \bar{b}_n[?_n^1])$
- $\bar{m}_2 = (\bar{b}_1 \bar{p}_1^2 \bar{b}_2[?_2^2] \dots \bar{p}_{k-1}^2 \bar{b}_k[?_k^2], ns)$

Then, $\overline{\mathbf{m}}'_1.s = \overline{\mathbf{b}}_1 \overline{\mathbf{p}}_1 \overline{\mathbf{b}}_2 [?^1] \dots \overline{\mathbf{p}}_{k-1} \overline{\mathbf{b}}_k [?^1] \overline{\mathbf{p}}_1^2 (\overline{\mathbf{b}}_k +_{\mathbb{B}} (\overline{\mathbf{b}}_2 -_{\mathbb{B}} \overline{\mathbf{b}}_1)) [?^2] \dots \overline{\mathbf{p}}_{k-1}^2 (\overline{\mathbf{b}}' +_{\mathbb{B}} (\overline{\mathbf{b}}_k -_{\mathbb{B}} \overline{\mathbf{b}}_{k-1})) [?^2]$ 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 13. Let $([0, 0] \mathbf{a}^* [5, 7], [6, 8] \mathbf{br}^* [13, 14])$ and $([0, 0] \mathbf{a}^* [3, 3], \emptyset)$ 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, $([0, 0] \mathbf{a}^* [5, 7], [6, 8] \mathbf{br}^* [13, 14])$ 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, $([0, 0] \mathbf{a}^* [3, 3], \emptyset)$ 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,

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

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:

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

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

5.2.3. Abstract String Character

The semantics operator $\mathfrak{M}_{\overline{\mathbf{M}}}$, when applied to $\text{strchr}_{\overline{\mathbf{v}}}(\overline{\mathbf{m}})$, returns a split segmentation abstract predicate $\overline{\mathbf{s}}$ with the left hand side parameter equal to the suffix segmentation of the input $\overline{\mathbf{m}}.s$ from the first segment in which $\overline{\mathbf{v}}$ 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 $\overline{\mathbf{v}}$ 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 $\overline{\mathbf{v}}$ does not occur in $\overline{\mathbf{m}}.s$, it returns $\perp_{\overline{\mathbf{M}}}$; otherwise it returns $\top_{\overline{\mathbf{M}}}$. Formally,

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

where $\overline{\mathbf{s}} = (\overline{\mathbf{b}}_j \overline{\mathbf{p}}_j \overline{\mathbf{b}}_{j+1} \dots \overline{\mathbf{b}}_k [?^k], \emptyset)$ such that $j = \min\{i \in \overline{\mathbf{m}}.s \mid \overline{\mathbf{p}}_i = \overline{\mathbf{v}} \wedge \overline{\mathbf{b}}_i, \overline{\mathbf{b}}_{i+1} \text{ are not question marked}\}$.

5.2.4. Abstract String Compare

The semantics $\mathfrak{P}_{\overline{\mathbf{M}}}$ is the abstract counterpart of \mathfrak{P} . In particular, $\text{strcmp}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2)$ returns a value $\overline{\mathbf{n}}$ denoting the lexicographic distance between $\overline{\mathbf{m}}_1.s$ and $\overline{\mathbf{m}}_2.s$ 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 $\overline{\mathbf{n}}$ 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 $\overline{\mathbf{n}}$ 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 $\overline{\mathbf{n}}$ is equal to zero they are lexicographically equal. Formally,

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

where $\overline{\mathbf{n}} = \text{comparison}_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2)$ (c.f. Algorithm 2).

Algorithm 2 Lexicographic comparison of split segmentation abstract predicates.**Function:** $comparison_{\overline{\mathbf{M}}}(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2)$ **Input:** two compatible split segmentation abstract predicates $\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2 \in \overline{\mathbf{M}}$.**Output:** an integer value \overline{n} .

```

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

```

5.2.5. Abstract String Copy

The semantics $\mathfrak{M}_{\overline{\mathbf{M}}}$, when applied to $strcopy(\overline{\mathbf{m}}_1, \overline{\mathbf{m}}_2)$, 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 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,

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

The `size.condition` is true if $minlen(\overline{\mathbf{m}}_1) \geq_{\overline{\mathbf{b}}} len(\overline{\mathbf{m}}_2.s) +_{\overline{\mathbf{b}}} 1$. Let:

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

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

5.2.6. Abstract String Length

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

$$\mathfrak{L}_{\overline{\mathbf{M}}}[strlen](\overline{\mathbf{m}}) = \begin{cases} \overline{n} & \text{if } \overline{\mathbf{m}}.s = \overline{\mathbf{b}}_1 \overline{\mathbf{p}}_1 \overline{\mathbf{b}}_2 [?_2] \dots \overline{\mathbf{b}}_k \wedge \nexists i \in \overline{\mathbf{m}}.s : is_null(\overline{\mathbf{p}}_i) = \text{maybe} \\ \top_{\overline{\mathbf{b}}} & \text{otherwise} \end{cases}$$

where $\bar{n} = \bar{b}_k - \bar{b}_1$.

5.2.7. Abstract Array Update

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

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

5.3. Soundness

Theorem 2. $\mathfrak{A}_{\bar{M}}$ is a sound over-approximation of \mathfrak{A} . Formally,

$$\gamma_{\bar{C}}(\mathfrak{A}_{\bar{M}}[\text{stm}])(\bar{m}) \supseteq \{\mathfrak{A}[\text{stm}](\mu(m)) : \mu(m) \in \gamma_{\bar{M}}(\bar{m})\}$$

Proof. Consider the unary operator access_j and let \bar{m} be a split segmentation abstract predicate. We have to prove that:

$$\gamma_{\bar{C}}(\mathfrak{A}_{\bar{M}}[\text{access}_j])(\bar{m}) \supseteq \{\mathfrak{A}[\text{access}_j](\mu(m)) : \mu(m) \in \gamma_{\bar{M}}(\bar{m})\}$$

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

Theorem 3. $\mathfrak{M}_{\bar{M}}$ is a sound over-approximation of \mathfrak{M} . Formally,

$$\gamma_{\bar{M}}(\mathfrak{M}_{\bar{M}}[\text{stm}])(\bar{m}) \supseteq \{\mathfrak{M}[\text{stm}](\mu(m)) : \mu(m) \in \gamma_{\bar{M}}(\bar{m})\}$$

Proof.

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

$$\gamma_{\bar{M}}(\mathfrak{M}_{\bar{M}}[\text{strcat}])(\bar{m}_1, \bar{m}_2) \supseteq \{\mathfrak{M}[\text{strcat}](\mu(m_1), \mu(m_2)) : \mu(m_1) \in \gamma_{\bar{M}}(\bar{m}_1) \wedge \mu(m_2) \in \gamma_{\bar{M}}(\bar{m}_2)\}$$

strcat of $\mu(m_1)$ and $\mu(m_2)$ returns, by definition of \mathfrak{M} , $\mu(m_1)'$ where the first null-terminating memory block of $\mu(m_2)$ (including the null terminator), i.e., its string of interest, is embedded into $\mu(m_1)$ starting from the index to which occurs the first null character in $\mu(m_1)$, if both $\mu(m_1)$ and $\mu(m_2)$ contain a well-formed string and the size condition on the destination character array value is fulfilled; $\top_{\bar{M}}$ otherwise. Then, $\mu(m_1)'$ belongs to $\gamma_{\bar{M}}(\mathfrak{M}_{\bar{M}}[\text{strcat}])(\bar{m}_1, \bar{m}_2)$ because strcat of \bar{m}_1 and \bar{m}_2 , by definition of $\mathfrak{M}_{\bar{M}}$, is equal to \bar{m}_1' that is \bar{m}_1 into which \bar{m}_2 's has been embedded starting from the upper bound of \bar{m}_1 's, if both \bar{m}_1 and \bar{m}_2 approximate character arrays which contain a well-formed string and the size condition on the destination segmentation abstract predicate is fulfilled; $\top_{\bar{M}}$ otherwise.

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

$$\gamma_{\bar{M}}(\mathfrak{M}_{\bar{M}}[\text{strchr}_{\bar{v}}])(\bar{m}) \supseteq \{\mathfrak{M}[\text{strchr}_{\bar{v}}](\mu(m)) : \mu(m) \in \gamma_{\bar{M}}(\bar{m})\}$$

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

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

$$\gamma_{\bar{M}}(\mathfrak{M}_{\bar{M}}[\text{strcpy}](\bar{m}_1, \bar{m}_2)) \supseteq \{\mathfrak{M}[\text{strcpy}](\mu(m_1), \mu(m_2)) : \mu(m_1) \in \gamma_{\bar{M}}(\bar{m}_1) \wedge \mu(m_2) \in \gamma_{\bar{M}}(\bar{m}_2)\}$$

strcpy of $\mu(m_1)$ and $\mu(m_2)$ returns, by definition of \mathfrak{M} , $\mu(m_1)'$ where the first null-terminating memory block of $\mu(m_2)$ (including the null terminator), i.e., its string of interest, is embedded into $\mu(m_1)$ starting from the lower bound of $\mu(m_1)$, if both $\mu(m_1)$ and $\mu(m_2)$ contain a well-formed string and the size condition on the destination character array value is fulfilled, \top_M otherwise. Then, $\mu(m_1)'$ belongs to $\gamma_{\bar{M}}(\mathfrak{M}_{\bar{M}}[\text{strcpy}](\bar{m}_1, \bar{m}_2))$ because strcpy of \bar{m}_1 and \bar{m}_2 , by definition of $\mathfrak{M}_{\bar{M}}$, is equal to \bar{m}_1' that is \bar{m}_1 into which $\bar{m}_2.s$ has been embedded starting from the lower bound of \bar{m}_1 , if both \bar{m}_1 and \bar{m}_2 approximate character arrays which contain a well-formed string and the size condition on the destination segmentation abstract predicate is fulfilled; $\top_{\bar{M}}$ otherwise.

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

$$\gamma_{\bar{M}}(\mathfrak{M}_{\bar{M}}[\text{update}_{j,\bar{v}}](\bar{m})) \supseteq \{\mathfrak{M}[\text{update}_{j,\bar{v}}](\mu(m)) : \mu(m) \in \gamma_{\bar{M}}(\bar{m})\}$$

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

□

Theorem 4. $\mathfrak{P}_{\bar{M}}$ is a sound over-approximation of \mathfrak{P} . Formally,

$$\gamma_{\bar{B}}(\mathfrak{P}_{\bar{M}}[\text{stm}](\bar{m})) \supseteq \{\mathfrak{P}[\text{stm}](\mu(m)) : \mu(m) \in \gamma_{\bar{M}}(\bar{m})\}$$

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

$$\gamma_{\bar{C}}(\mathfrak{P}_{\bar{M}}[\text{strcmp}](\bar{m}_1, \bar{m}_2)) \supseteq \{\mathfrak{P}[\text{strcmp}](\mu(m_1), \mu(m_2)) : \mu(m_1) \in \gamma_{\bar{M}}(\bar{m}_1) \wedge \mu(m_2) \in \gamma_{\bar{M}}(\bar{m}_2)\}$$

strcmp of $\mu(m_1)$ and $\mu(m_2)$ returns an integer value n , resulting from the difference between corresponding character array elements, denoting the lexicographic distance between the strings of interest of $\mu(m_1)$ and $\mu(m_2)$, if both contain a well-formed string, \top_Z otherwise, by definition of \mathfrak{P} . Then n belongs to $\gamma_{\bar{C}}(\mathfrak{P}_{\bar{M}}[\text{strcmp}](\bar{m}_1, \bar{m}_2))$ because strcmp of \bar{m}_1 and \bar{m}_2 , by definition of $\mathfrak{P}_{\bar{M}}$, is equal to \bar{n} that is the difference between corresponding segment abstract predicates, denoting the lexicographic

distance between $\bar{m}_1.s$ and $\bar{m}_2.s$, if \bar{m}_1 and \bar{m}_2 are comparable, both approximate character arrays which contain a well-formed string where $\top_{\bar{c}}$ does not occur; \top_z otherwise. \square

Theorem 5. $\mathcal{L}_{\bar{M}}$ is a sound over-approximation of \mathcal{L} . Formally,

$$\gamma_{\bar{B}}(\mathcal{L}_{\bar{M}}\llbracket\text{stm}\rrbracket(\bar{m})) \supseteq \{\mathcal{L}\llbracket\text{stm}\rrbracket(\mu(m)) : \mu(m) \in \gamma_{\bar{M}}(\bar{m})\}$$

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

$$\gamma_{\bar{B}}(\mathcal{L}_{\bar{M}}\llbracket\text{strlen}\rrbracket(\bar{m})) \supseteq \{\mathcal{L}\llbracket\text{strlen}\rrbracket(\mu(m)) : \mu(m) \in \gamma_{\bar{M}}(\bar{m})\}$$

`strlen` of $\mu(m)$ returns, by definition of \mathcal{L} , an integer value n which denotes the length of the sequence of character before the first null one in $\mu(m)$, if $\mu(m)$ contains a well-formed string; \top_z otherwise. Then n belongs to $\gamma_{\bar{B}}(\mathcal{L}_{\bar{M}}\llbracket\text{strlen}\rrbracket(\bar{m}))$ because `strlen` of (\bar{m}) , by definition of $\mathcal{L}_{\bar{M}}$ is equal to the difference between the lower and the upper bound of $\bar{m}.s$ if \bar{m} approximates character arrays which contain a well-formed string of interest; $\top_{\bar{B}}$ otherwise. \square

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 paper, 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 [7]. 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 6.3, we explore the implications of aggregate (as opposed to scalar) domains within this framework. Sections 6.4 and 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.

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 1 depicts a

comparison of the compilation-based approach with respect the interpretation-based approach adopted by more conventional abstract interpreters.

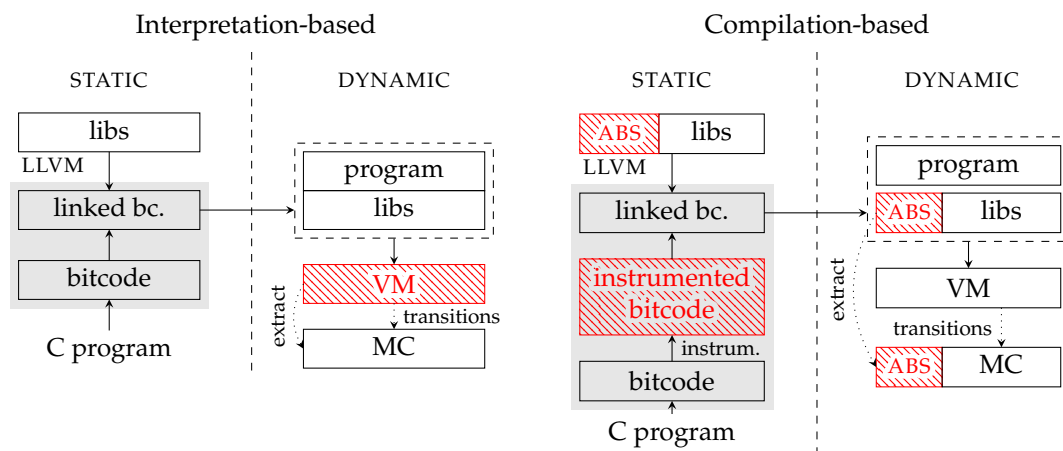


Figure 1. In the figure, we compare an abstract interpretation with a compilation-based approach. In the interpretation-based approach, the whole abstract interpretation is performed at runtime. The bitcode operations are interpreted abstractly by a virtual machine (VM) which maintains an abstract state. 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 [7].

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 [16]. Syntactic abstraction replaces some of the LLVM instructions that occur in the program with their abstract counterparts, as depicted in Figure 2.

```

1 char *a = input_string();
2 char *b = string();
3 char *c = strcat(a, b);
4 int l = strlen(c);

1 abstract a = abstract_string();
2 char *b = string();
3 abstract c = abstract_strcat(a, lift(b));
4 abstract l = abstract_strlen(c);

```

Figure 2. Syntactic abstraction.

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 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 $\Gamma(T)$ derived from a set of scalars T is defined as follows:

1. $T \subseteq \Gamma(T)$, meaning each scalar type is included in $\Gamma(T)$,
2. if $t_1, \dots, t_n \in \Gamma(T)$ then also the *product type* is in $\Gamma(T)$: $(t_1, \dots, t_n) \in \Gamma(T), n \in \mathbb{N}$,
3. if $t_1, \dots, t_n \in \Gamma(T)$ then also *disjoint union* is in $\Gamma(T)$: $t_1 \mid t_2 \mid \dots \mid t_n \in \Gamma(T), n \in \mathbb{N}$,
4. if $t \in \Gamma(T)$ then $t^* \in \Gamma(T)$, where t^* denotes pointer type.

In a concrete LLVM program, the set of admissible types comprise those derived from concrete scalars S , i.e., $\Gamma(S)$. 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 \bar{S} . Correspondence between abstract and concrete scalars is given by a bijective map $\Lambda : S \rightarrow \bar{S}$. Finally, each value, which exists in the abstracted program, has an assigned type of $\Gamma(S \cup \bar{S})$. 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.

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 4.3.1 for abstract semantics of access, respectively to the Section 4.3.7 for the abstract semantics of 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 $P^* \subseteq \Gamma(S)$. A set of abstract pointers types $\overline{P^*}$ 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 $\Lambda : P^* \rightarrow \overline{P^*}$. 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: $\Gamma(S \cup \overline{P^*})$.

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.

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 9 and 10): operations (instructions) converting values between their concrete and abstract representations. They realize a runtime implementation of domain functions: abstraction ($\alpha_{\overline{M}}$ in the case of M-String) and concretization ($\gamma_{\overline{M}}$).

The *lift* operation implements abstraction of concrete values by a single over-approximating abstract value. For example, in Figure 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 operator. 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 be implemented as random, by choosing one particular path. For further details of the program transformation performed by LART, we kindly refer the reader to [7].

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 $(t_1, \dots, t_n) \rightarrow t_r$ is substituted by an abstract operation of type $(\Lambda(t_1), \dots, \Lambda(t_n)) \rightarrow \Lambda(t_r)$. 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 $(m, m) \rightarrow s$ is replaced by `abstract_strcmp` of type $(\Lambda(m), \Lambda(m)) \rightarrow \Lambda(s)$ 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 [7].

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 losing a certain degree of accuracy in the abstraction, the exact amount depending on the single operation.

7. Instantiating M-String

As an aggregate domain, M-String is 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 5, provided a suitable scalar domain \bar{C} for characters and scalar domain \bar{B} to represent segment bounds. The instantiation demands that both scalar domains \bar{C} and \bar{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.

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 [7].

The domain of symbolic values (we call it a term domain) requires DIVINE to be augmented with an SMT solver from a suitable theory. For scalars in C programs, we use the bitvector theory. 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).

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 $\bar{\mathbf{C}}$ to be the concrete domain (i.e., we let the characters be represented by themselves). We let the domain of segment bounds $\bar{\mathbf{B}}$ 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.

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). The elements of these arrays are characters and bounds, whose type is derived from parametrization, i.e., from the scalar domains $\bar{\mathbf{C}}$ and $\bar{\mathbf{B}}$. 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 14.

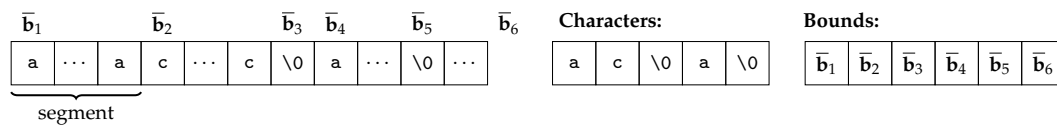


Figure 3. M-String value with symbolic bounds, where string of interest is from \bar{b}_1 to \bar{b}_3 .

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 $a^k b^l c^m \dots$ where relationships between k, l, m, \dots 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 8.

Example 14. Simple program analysis with symbolic bounds and concrete characters:

```

1 mstring str = abstract_string(x,  $\bar{b}_1$ , \0,  $\bar{b}_2$ , y,  $\bar{b}_3$ , \0,  $\bar{b}_4$ );
2 symbolic idx = abstract_int();
3 if (idx <  $\bar{b}_4$ ) {
4   str[idx] = 'y';
5   symbolic len = abstract_strlen(str);
6 }

```

Imagine we are given symbolic bounds $\bar{b}_1 < \bar{b}_2 < \bar{b}_3 < \bar{b}_4$, then the first line of the transformed program creates *mstring* value with characters $[x, \0, y, \0]$ and bounds $[0, \bar{b}_1, \bar{b}_2, \bar{b}_3, \bar{b}_4]$. 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{str}_x , as result we join all possibilities:

1. if $idx < \overline{b}_1$: idx falls to the first segment: $\overline{str}_1 = ([x, y, x, \backslash 0, y, \backslash 0], [0, idx, idx + 1, \overline{b}_1, \overline{b}_2, \overline{b}_3, \overline{b}_4])$ and creates a new segment between idx and $idx + 1$ containing character y . Notice that if $idx = 0$ the first segment is empty, similarly the third segment for $idx + 1 = \overline{b}_1$. The string of interest for \overline{str}_1 is of form $x^{idx}y^1x^{\overline{b}_1-idx-1}$.
2. if $\overline{b}_1 \leq idx < \overline{b}_2$: than $\overline{str}_2 = ([x, \backslash 0, y, \backslash 0, y, \backslash 0], [0, \overline{b}_1, idx, idx + 1, \overline{b}_2, \overline{b}_3, \overline{b}_4])$, with string of interest as join of following forms:
 - if the update is performed right after the first segment, i.e., $idx = \overline{b}_1$:
 - if and $|\overline{b}_1 - \overline{b}_2| > 1$, i.e., the segment of zeros contains more elements, then the string has form $x^{\overline{b}_1}y$,
 - otherwise the update overwrites the single zero character, hence extends the string of interest by segment of y characters: $x^{\overline{b}_1}y^{\overline{b}_3-\overline{b}_1}$.
 - otherwise between first segment and idx is a terminating zero, hence the string of interest remains unchanged: $x^{\overline{b}_1}$.
3. if $\overline{b}_2 \leq idx < \overline{b}_3$: than $\overline{str}_3 = \overline{str}$, because update stores the same character as is already present in the segment.
4. if $\overline{b}_3 \leq idx < \overline{b}_4$: than update creates a new segment inside of sequence of last zeros: $\overline{str}_4 = ([x, \backslash 0, y, \backslash 0, y, \backslash 0], [0, \overline{b}_1, \overline{b}_2, \overline{b}_3, idx, idx + 1, \overline{b}_4])$.

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., $\overline{b}_1 \cup \overline{b}_3$.

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{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.

7.4. Implementation

Finally, we implemented the M-String abstraction as a LART domain. The implementation, with examples and documentation of domain usage, can be found online on the supplementary page <https://divine.fi.muni.cz/2020/mstring>. The LART domain is a C++ library that implements abstract semantics of M-String operations presented in Section 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 7.2 and depicted in Figure 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 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 6.3), and the representation of the *offset* component must be compatible with the bound domain $\bar{\mathbf{B}}$.

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 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.

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

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 2) and the other variant (Table 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.

Table 2. Measurements of M-String operations on two types of inputs: *Word* and *Sequence* described in Section 8.1. 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 8.1.

	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. 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.

	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	–

8.2. C Standard Libraries

In the second set of benchmarks (see Table 4), 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 μ CLibc. 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.

Table 4. Verification results of functions from PDCLib with timeout of 1 h. Measurements show the size of state space and verification time for the parametric length of the input.

	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

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 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.

8.3. Veriabs Overflow Benchmarks

In this scenario (see Table 5), 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.

Table 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.

	Correct		Error Found		Timeout
	Tests	Time(s)	Tests	Time(s)	
apache	0	–	26	384.26	24
openser	43	234.13	45	105.93	6
wu-ftp	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

8.4. 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 6. As with the previous benchmark sets, the M-String inputs with a smaller number of segments outperformed other analyses. 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.

Table 6. Measurements of time and size of state space for analyses of automatically generated parsers.

	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	–
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

9. Related Work

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 [17]. Furthermore, the above mentioned techniques and a large number of bug hunting tools based on static analysis have been implemented [18–23].

For instance, in [24] 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 [20] 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 [25], is a tool that automatically generates test inputs, symbolically reasoning about lengths of input buffers.

Static code analysis aims at approximating possible behaviours of a program without examining all of its (possibly infinite) actual executions. By a proper abstraction of data and operations, static analysis results into an over-approximation of all the possible runs of a program, and its effectiveness heavily depends on degree of precision of such an abstraction. In particular, the framework of abstract interpretation [9] can be adopted also to approximate semantics of string operations. The basic, well-known domain is a *string set* domain, which simply keeps track of a set of strings and it is a specific instance of the general (bounded) set domain. Others are the *prefix-suffix* domain (which captures the first and the last letter of a string) and the *character inclusion* domain (which only tracks the characters that surely or maybe appear in a string). Another general-purpose string domain is the *string hash* domain proposed in [26], based on a distributive hash function. More complete reviews of general-purpose string domains can be found in [11,27].

Most general-purpose domains 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, [28] 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 [29], such domain is able to describe relations between length of variables and offsets of pointers. Amadini et al. [27] have evaluated several abstract string domains (and their combinations) for analysis of JavaScript programs. In [30] was defined the simplified regular expression domain for JavaScript analysis 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 [30–33]. 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.

10. Conclusions

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 codes 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.

As a future work, we plan to further enhance the effectiveness of the M-String domains by combining it by reduced product with other either numerical or symbolic domains.

Supplementary Materials: The supplementary materials at <https://divine.fi.muni.cz/2020/mstring> contain binary distribution and sources of extended model-checker DIVINE, which is open source software distributed under the ISC license.

Author Contributions: All authors contributed substantially to the work reported. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been partially supported by the Czech Science Foundation grant No. 18-02177S.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Cass, S. The Top Programming Languages 2019. *IEEE Spectr. Mag.* 2019. Available online: <https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019> (accessed on 10 February 2020).
2. Bultan, T.; Yu, F.; Alkhalaf, M.; Aydin, A. *String Analysis for Software Verification and Security*; Springer: Berlin/Heidelberg, Germany, 2017.
3. One, A. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 8 November 1996.
4. Cortesi, A.; Olliaro, M. M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs. In Proceedings of the Theoretical Aspects of Software Engineering—12th International Symposium (TASE 2018), Guangzhou, China, 29–31 August 2018. [[CrossRef](#)]
5. Cortesi, A.; Lauko, H.; Olliaro, M.; Rockai, P. String Abstraction for Model Checking of C Programs. In Proceedings of the Model Checking Software—26th International Symposium (SPIN 2019), Beijing, China, 15–16 July 2019; pp. 74–93. [[CrossRef](#)]
6. Cousot, P.; Cousot, R.; Logozzo, F. 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, 26–28 January 2011; pp. 105–118. [[CrossRef](#)]
7. Lauko, H.; Rockai, P.; Barnat, J. Symbolic Computation via Program Transformation. In Proceedings of the Theoretical Aspects of Computing—15th International Colloquium (ICTAC 2018), Stellenbosch, South Africa, 16–19 October 2018; pp. 313–332. [[CrossRef](#)]
8. Baranová, Z.; Barnat, J.; Kejstová, K.; Kucera, T.; Lauko, H.; Mrázek, J.; Rockai, P.; Still, V. Model Checking of C and C++ with DIVINE 4. In Proceedings of the Automated Technology for Verification and Analysis—15th International Symposium (ATVA 2017), Pune, India, 3–6 October 2017; pp. 201–207. [[CrossRef](#)]
9. Cousot, P.; Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Proceedings of the Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, CA, USA, 17–19 January 1977; pp. 238–252. [[CrossRef](#)]

10. Cousot, P.; Cousot, R. Systematic Design of Program Analysis Frameworks. In Proceedings of the Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, TX, USA, 29–31 January 1979; pp. 269–282. [[CrossRef](#)]
11. Costantini, G.; Ferrara, P.; Cortesi, A. A Suite of Abstract Domains for Static Analysis of String Values. *Softw. Pract. Exper.* **2015**, *45*, 245–287. [[CrossRef](#)]
12. Cousot, P.; Cousot, R. Abstract Interpretation Frameworks. *J. Log. Comput.* **1992**, *2*, 511–547, [[CrossRef](#)]
13. Cortesi, A.; Zanioli, M. Widening and Narrowing Operators for Abstract Interpretation. *Comput. Lang. Syst. Struct.* **2011**, *37*, 24–42. [[CrossRef](#)]
14. Gange, G.; Navas, J.A.; Schachte, P.; Søndergaard, H.; Stuckey, P.J. Abstract Interpretation over Non-lattice Abstract Domains. In Proceedings of the Static Analysis—20th International Symposium (SAS 2013), Seattle, WA, USA, 20–22 June 2013; Logozzo, F., Fähndrich, M., Eds; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7935, pp. 6–24. [[CrossRef](#)]
15. Seacord, R.C. *Secure Coding in C and C++*, 2nd ed.; Addison-Wesley Professional: Boston, MA, USA, 2013.
16. Lattner, C.; Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the International Symposium on Code Generation and Optimization (CGO'04), San Jose, CA, USA, USA, 20–24 March 2004. [[CrossRef](#)]
17. Shahriar, H.; Zulkernine, M. Classification of Static Analysis-Based Buffer Overflow Detectors. In Proceedings of the Fourth International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010), Singapore, 9–11 June 2010; pp. 94–101. [[CrossRef](#)]
18. Xie, Y.; Chou, A.; Engler, D.R. 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, Helsinki, Finland, 1–5 September 2003; pp. 327–336. [[CrossRef](#)]
19. Wagner, D.A.; Foster, J.S.; Brewer, E.A.; Aiken, A. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2020.
20. Dor, N.; Rodeh, M.; Sagiv, S. 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, CA, USA, 9–11 June 2003; pp. 155–167. [[CrossRef](#)]
21. Evans, D.; Larochelle, D. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.* **2002**, *19*, 42–51. [[CrossRef](#)]
22. Holzmann, G.J. UNO: Static Source Code Checking for UserDefined Properties. In Proceedings of the 6th World Conference on Integrated Design and Process Technology, IDPT'02, Pasadena, CA, USA, 23–28 June 2002.
23. MathWorks. Polyspace. 2001. Available online: <https://www.mathworks.com/products/polyspace.html> (accessed on 10 February 2020).
24. Jones, R.W.M.; Kelly, P.H.J. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. Presented at AADEBUG, Linköping, Sweden, 26–27 May 1997; pp. 13–26.
25. Xu, R.; Godefroid, P.; Majumdar, R. Testing for Buffer Overflows with Length Abstraction. In Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, Seattle, WA, USA, 20–24 July 2008; pp. 27–38. [[CrossRef](#)]
26. Madsen, M.; Andreasen, E. String Analysis for Dynamic Field Access. In Proceedings of the Compiler Construction—23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, Grenoble, France, 5–13 April 2014; pp. 197–217. [[CrossRef](#)]
27. Amadini, R.; Jordan, A.; Gange, G.; Gauthier, F.; Schachte, P.; Søndergaard, H.; Stuckey, P.J.; Zhang, C. Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In Proceedings of the European Joint Conferences on Theory and Practice of Software, Uppsala, Sweden, 22–29 April 2017; pp. 41–57. [[CrossRef](#)]
28. Journault, M.; Miné, A.; Ouadjaout, A. Modular Static Analysis of String Manipulations in C Programs. In Proceedings of the Static Analysis—25th International Symposium, Freiburg, Germany, 29–31 August 2018; pp. 243–262. [[CrossRef](#)]

29. Miné, A. 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, ON, Canada, 14–16 June 2006; pp. 54–63. [[CrossRef](#)]
30. Park, C.; Im, H.; Ryu, S. Precise and Scalable Static Analysis of jQuery Using a Regular Expression Domain. In Proceedings of the 12th Symposium on Dynamic Languages, Amsterdam, The Netherlands, 1 November 2016; pp. 25–36. [[CrossRef](#)]
31. Spoto, F. The Julia Static Analyzer for Java. In Proceedings of the Static Analysis—23rd International Symposium, Edinburgh, UK, 8–10 September 2016; pp. 39–57. [[CrossRef](#)]
32. Jensen, S.H.; Møller, A.; Thiemann, P. Type Analysis for JavaScript. In Proceedings of the 16th International Static Analysis Symposium, Los Angeles, CA, USA, 9–11 August 2009; pp. 238–255. [[CrossRef](#)]
33. Kashyap, V.; Dewey, K.; Kuefner, E.A.; Wagner, J.; Gibbons, K.; Sarracino, J.; Wiedermann, B.; Hardekopf, B. 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, 16–22 November 2014; pp. 121–132. [[CrossRef](#)]

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).