Noname manuscript No.
(will be inserted by the editor)

Abstracting Strings for Model Checking of C Programs

Agostino Cortesi · Henrich Lauko · Martina Olliaro · Petr Ročkai

Received: date / Accepted: date

Abstract Automatic abstraction is a powerful software verification technique. In this paper, we elaborate an abstract domain for `C` strings, that is, null-terminated arrays of characters. The domain we present (called M-String) is parametrized on an index (bound) domain and a character domain. Picking different constituent domains, i.e., both shape information on the array structure and value information on the contained characters, allows M-String to be tailored for specific verification tasks, balancing precision against complexity. We describe the concrete and the abstract semantics of basic string operations and prove their soundness formally. In addition to a selection of string functions from the standard `C` library, we provide semantics for character access and update, enabling automatic lifting of arbitrary string-manipulating code into the domain.

In addition to describing the domain theoretically, we also provide an executable implementation of the abstract operations. Using a tool which automatically lifts existing programs into the M-String domain along with an explicit-state model checker, we evaluate the accuracy of the proposed domain experimentally on real-case test programs.

Keywords String analysis · Model checking · Abstract interpretation · Abstract domain

Martina Olliaro is the corresponding author.

Agostino Cortesi · Martina Olliaro
Ca' Foscari University of Venice. Scientific Campus,
Via Torino 155, 30172 Mestre, Venice, Italy
E-mail: cortesi@unive.it
E-mail: martina.olliaro@unive.it

Henrich Lauko · Martina Olliaro · Petr Ročkai
Masaryk University. Faculty of Informatics,
Botanickà 68A, 60200 Brno, Czech Republic
E-mail: xlauko@mail.muni.cz
E-mail: xrockai@fi.muni.cz

1 Introduction[1]

The `C` programming language is still very relevant [4]: a large number of systems of critical importance are written in `C`, including server software and embedded systems. Unfortunately, due to the way `C` programs are laid out in memory, they often contain bugs that can be exploited by malicious parties to mount security attacks. Guaranteeing correctness of such software is of great concern. In particular, we are interested in ensuring correctness of `C` programs that manipulate strings. Incorrect string manipulation can cause a number of catastrophic events, ranging from crashes in critical software components to loss or exposure of sensitive data.

In the `C` programming language, strings are not a basic data type and operations on them are provided as library functions [3]. Indeed strings are represented as zero-terminated arrays of characters – due to the possible discrepancy between string size and array (buffer) size, `C` programs which manipulate strings can suffer from buffer overflows and related issues. A buffer overflow is a bug that affects `C` code which incorrectly tries to access a buffer outside its bounds – an out-of-bounds write (a related bug – an out-of-bounds read – is also a problem, even though not as immediately dangerous as a buffer overflow). Moreover, buffer overflows are usually exploitable and often can easily lead to arbitrary code execution [26]. In the light of these facts, it is clearly important to investigate methods to automatically reason about correctness of string manipulation code in `C` programs. Automated code analysis tools can identify existing bugs, reduce the risk of introducing new bugs and therefore help prevent costly security incidents.

[1] This paper is a revised and extended version of [6] and [8].

1.1 Paper Contribution

In this paper, we present a sound approach for conducting string analysis in C programs. We define the M-String abstract domain which approximates sets of C character arrays. This domain allows the abstraction of both shape information on the array structure and value information on the contained characters, and it highlights the presence of well-formed strings in the approximated character arrays. M-String is based on a refinement of the segmentation approach to array representation proposed in [13].

The goal of M-String is to infer the presence of common string manipulation errors that may result in buffer overflows or, more generally, that may lead to undefined behaviour. Additionally, keeping track of the content of the character array after the first null character allows us to reduce false positives: in particular, rewriting the first null character in the string is not always a bug, since further null characters may follow.

Finally, M-String, like the array segmentation-based representation defined in [13], is parametric with respect to the abstraction of the array elements value, and the representation of array indices.

Additionally, we have extended LART [22], a tool which can perform automatic abstraction on programs, with support for more complicated (non-scalar) domains, which allowed us to also integrate the M-String domain. By using the extended version of LART along with DIVINE 4 [2], an explicit state model checker based on LLVM, we can automatically verify correctness of string operations in C programs. We demonstrate this capability by analyzing a number of 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 main contribution of this paper is in demonstrating the actual impact of an ad-hoc segmentation-based abstract domain on model checking of C programs.

1.2 Paper Structure

Section 2 presents related works. Section 3 gives basics in abstract interpretation and introduces the array segmentation abstract domain [13] on which M-String is based. Section 5 defines the concrete domain and semantics. Section 6 presents the M-String abstract domain for C character arrays and its semantics, whose soundness is formally proved. Section 10 concludes.

2 Related Work

Static methods with the ability to automatically detect buffer overflows have been widely studied in the literature and many different inference techniques were proposed and implemented: constraint solvers for various theories (including string theories) and techniques based on them (e.g., symbolic execution), tainted data-flow analysis, string pattern matching analysis or annotation analysis [30]. Additionally, a large number of bug hunting tools based on static analysis and the above mentioned techniques have been implemented [33, 32, 14, 15, 16, 24].

For instance, in [18] authors introduced a performant backward compatible method of bounds checking of C programs, i.e., the representation of pointers is left unchanged (thus differentiating the proposed schema from previously existing techniques), allowing inter-operation between checked and unchecked code, with recompilation confined to the modules where problems might occur. In [14], a static verifier of C strings has been presented, namely CSSV. 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 [34], is a tool that automatically generates test inputs, symbolically reasoning about lengths of input buffers.

Briefly, static code analysis attempts to quickly approximate possible behaviours of a program, without examining its actual executions. This way, static analysis reasons about many of the possible runs of a program and provides a degree of assurance that the property of interest holds (or that it is violated). However, with static analysis, neither positive nor negative results are guaranteed to be correct [27].

Various researchers have shown how the framework of abstract interpretation [10] can be used 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 – this is a specific instance of the general (bounded) set domain. Other are the *character inclusion* domain (which keeps track of which characters appear in a string, but not in what order or how many times), the *prefix-suffix* domain (which keeps track of the first and the last letter) and their various products. Another general-purpose string domain is the *string hash* domain proposed in [23], based on a distributive hash function. A more complete review of general-purpose string domains is readily available in the literature, e.g. [9, 1].

Such general-purpose domains focus on the generic aspects of strings, without accounting for the specifics of string handling in different programming languages.

It is, however, often beneficial to consider such specific aspects of string representation when designing abstract domains for program analysis: indeed, M-String is a domain tailored specifically for the representation of strings used in C programs. With regards to the C programming language, [19] proposed an abstract domain for C strings which tracks both their length and the buffer allocated size into which their are contained. Then, the latter domain, together with the cell abstraction [25], describes relations between length of variables and offsets of pointers. A number of abstract string domains (and their combinations) for analysis of JavaScript programs have been evaluated in [1]. Another domain that was conceived for JavaScript analysis is the simplified regular expression domain defined in [28]. 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 deserves attention: the M-String domain, tailored for modeling zero-terminated strings stored in character buffers in C programs has first been described in [6]. In addition to theoretical work, a number of tools based on the above mentioned abstract domains and their combinations have been designed and implemented [31, 17, 20, 28].

3 Prerequisites

We assume the reader is familiar with order theory.

3.1 Abstract Interpretation

Abstract Interpretation [10, 11] 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 belongs to a concrete domain $\mathbf{D}$. Likewise, the abstract semantics belongs to an abstract domain $\overline{\mathbf{D}}$. Both the concrete and the abstract domains form a complete lattice, such that: $(\mathcal{D}, \leq_{\mathbf{D}}, \bot_{\mathbf{D}}, \top_{\mathbf{D}}, \sqcup_{\mathbf{D}}, \sqcap_{\mathbf{D}})$ and $(\overline{\mathcal{D}}, \leq_{\overline{\mathbf{D}}} \ \bot_{\overline{\mathbf{D}}}, \top_{\overline{\mathbf{D}}}, \sqcup_{\overline{\mathbf{D}}}, \sqcap_{\overline{\mathbf{D}}})$. The concrete and the abstract domains are related by a pair of monotonic functions: the concretization $\gamma_{\overline{\mathbf{D}}} : \overline{\mathbf{D}} \to \mathbf{D}$ and the abstraction $\alpha_{\overline{\mathbf{D}}} : \mathbf{D} \to \overline{\mathbf{D}}$ functions. In order to obtain a sound analysis, $\alpha_{\overline{\mathbf{D}}}$ and $\gamma_{\overline{\mathbf{D}}}$ have to form a Galois connection [9]. $(\alpha_{\overline{\mathbf{D}}}, \gamma_{\overline{\mathbf{D}}})$ is a Galois connection if and only if for every $d \in \mathcal{D}$ and $\overline{d} \in \overline{\mathcal{D}}$ we have that $d \leq_{\mathbf{D}} \gamma_{\overline{\mathbf{D}}}(\overline{d}) \Leftrightarrow \alpha_{\overline{\mathbf{D}}}(d) \leq_{\overline{\mathbf{D}}} \overline{d}$. Notice that, one function univocally identifies the other. Consequently, we can infer a Galois connection by proving that $\gamma_{\overline{\mathbf{D}}}$ is a complete meet morphism (resp. $\alpha_{\overline{\mathbf{D}}}$ is a complete join morphism) (Proposition 7 of [12]). Abstract domains suffering from infinite heights need to be equipped with a widening $\nabla_{\overline{\mathbf{D}}}$ and a narrowing $\Delta_{\overline{\mathbf{D}}}$ operator, in order to get fast convergence and to improve the accuracy of the resulting analysis, respectively [7]. An abstract domain functor $\overline{\mathbf{D}}$ is a function from the parameter abstract domains $\overline{\mathbf{D}}_1, \overline{\mathbf{D}}_2, ..., \overline{\mathbf{D}}_n$ to a new abstract domain $\overline{\mathbf{D}}(\overline{\mathbf{D}}_1, \overline{\mathbf{D}}_2, ..., \overline{\mathbf{D}}_n)$. The abstract domain functor $\overline{\mathbf{D}}(\overline{\mathbf{D}}_1, \overline{\mathbf{D}}_2, ..., \overline{\mathbf{D}}_n)$ composes abstract domain properties of the parameter abstract domains to build a new class of abstract properties and operations [13].

3.2 FunArray

In this section we recall the array segmentation analysis introduced in [13]. 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.

3.2.1 Array Concrete Semantics

The concrete value of an array variable $\mathtt{a} \in \mathbb{A}$ can be represented as a quadruple

$$\theta(\mathtt{a}) \in \mathcal{A} \triangleq \mathcal{R}_v \times \mathbb{E} \times \mathbb{E} \times (\mathbb{Z} \to (\mathbb{Z} \times \mathcal{V}))$$

$$\theta(\mathtt{a}) = (\rho, \ell ow_{\mathtt{a}}, high_{\mathtt{a}}, A_{\mathtt{a}})$$

where,

1) $\mathcal{R}_a$ is the set of concrete array environments. A concrete array environment $\theta \in \mathcal{R}_a$ maps array variables $\mathtt{a} \in \mathbb{A}$ to their values $\theta(\mathtt{a}) \in \mathcal{A}$.
2) Let $\mathcal{R}_v$ be the set of concrete variable environments. The function $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \to \mathcal{V}$ is a variable environment which maps variables (of basic types) $\mathtt{x} \in \mathbb{X}$ to their values $\rho(\mathtt{x}) \in \mathcal{V}$.
3) $\ell ow_{\mathtt{a}}, high_{\mathtt{a}} \in \mathbb{E}$ are integer expressions whose value, given by $[\![\ell ow_{\mathtt{a}}]\!]\rho$ and $[\![high_{\mathtt{a}}]\!]\rho$, respectively represents the lower bound and the upper bound of the array $\mathtt{a}$.
4) $A_{\mathtt{a}}$ is a function which maps indexes $i \in [[\![\ell ow_{\mathtt{a}}]\!]\rho, [\![high_{\mathtt{a}}]\!]\rho)$ to the pair $A_{\mathtt{a}}(i) = (i, v)$ such that v is the value of the element indexed by i, i.e., $A_{\mathtt{a}} : \mathrm{I}_{\mathtt{a}} \to \mathrm{P}_{\mathtt{a}}$.

   $$\mathrm{I}_{\mathtt{a}} = \{i : i \in [[\![\ell ow_{\mathtt{a}}]\!]\rho, [\![high_{\mathtt{a}}]\!]\rho)\}$$

   $$\mathrm{P}_{\mathtt{a}} = \{(i, v) : i \in \mathrm{I}_{\mathtt{a}} \wedge [\![\mathtt{a}[i]]\!]\rho = v \in \mathcal{V}\}$$

Example 1 Let $\mathtt{a}$ be a C integer array initialized as follows:

```
int a[5] = {5, 7, 9, 11, 13};
```

The concrete value of $\mathtt{a}$ is given by the tuple

$$\theta(\mathtt{a}) = (\rho, 0, 5, A_{\mathtt{a}}),$$

where the value of the lower and the upper bound of $\mathtt{a}$ are clear from the context and the codomain of the function $A_{\mathtt{a}}$ is the set

$$\mathrm{P}_{\mathtt{a}} = \{(\mathtt{0,5}), (\mathtt{1,7}), (\mathtt{2,9}), (\mathtt{3,\ 11}), (\mathtt{4,13})\}.$$

Let `b` denote the sub-array of `a` from position 2 to 3 included, its concrete value is given by

$$\theta(\mathtt{b}) = (\rho, \mathtt{2}, \mathtt{4}, A_{\mathtt{b}}) \text{ such that } \mathrm{P}_{\mathtt{b}} = \{(\mathtt{2,9}), (\mathtt{3,11})\}.$$

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

3.2.2 Array Segmentation Abstract Domain Functor

The FunArray abstract domain $\overline{\mathbf{S}}$ allows to represent a sequence of consecutive, non-overlapping and possibly empty segments that over-approximate a set of concrete array values in $\mathcal{P}(\mathcal{A})$. 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. The elements of FunArray belong to the set $\overline{\mathcal{S}} \triangleq \{(\overline{\mathcal{B}} \times \overline{\mathcal{A}}) \times (\overline{\mathcal{B}} \times \overline{\mathcal{A}} \times \{\text{␣}, ?\})^k \times (\overline{\mathcal{B}} \times \{\text{␣}, ?\}) \mid k \geqslant 0\} \cup \{\bot_{\overline{\mathbf{S}}}\}$, which have the form

$$\overline{b}_1 \overline{p}_1 \overline{b}_2 [?^2] \overline{p}_2 ... \overline{p}_{n-1} \overline{b}_n [?^n]$$

where,

1) $\overline{\mathbf{B}}$ is the segment bound abstract domain, approximating array indexes, with abstract properties $\overline{b}_i \in \overline{\mathcal{B}}$ such that $i \in [1, n]$ and $n > 1$.
2) $\overline{\mathbf{A}}$ is the array element abstract domain, with abstract properties $\overline{p}_i \in \overline{\mathcal{A}}$. It denotes possible values of pairs (index, indexed array element) in a segment, for relational abstractions, array elements otherwise.
3) $\overline{\mathbf{R}}$ is the variable environment abstract domain, which depends on the variable abstract domain $\overline{\mathbf{X}}$, with abstract properties $\overline{\rho} \in \overline{\mathcal{R}}$.
4) the question mark, if present, expresses the possibility that the segment that precedes it may be empty and can never precede $\overline{b}_1$ (the question mark opposite symbol is represented by the text visible space symbol ␣, but in general not empty segments are not marked).

Example 2 Let $\overline{\mathbf{B}}$ be an interval abstraction of indexes and let $\overline{\mathbf{A}}$ be a sign abstraction of numerical values and $\overline{\mathbf{R}}$ be the arithmetic expression evaluation environment. The segmentation abstract predicate below

$$[0,2] \ + \ [3,3]? \ - \ [5,7]$$

represents arrays like:

7	9	10	-11	-9	-8	-3
0	1	2	3	4	5	6

6	8	5	-4	-2
0	1	2	3	4

-2	-6	-3	-1	-6	-8
0	1	2	3	4	5

Notice that, in the last case, the lack of positive values is justified by the presence of the question mark that says that the first segment is optional.

The paper [13] defined the *unification* algorithm that modifies two compatible segmentations[2] so that they coincides (cf. [13], Section 11).

The array segmentation abstract domain is a complete lattice where the lattice operators are *unification*-based. Finally, the lattice is equipped with a widening and a narrowing operators.

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 sophisticated enough to detect common string manipulation errors.

4 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, in [5]:

- `strcat` appends the null-terminated string pointed to by `str1` to the null-terminated string pointed to by `str2`. The first character of `str2` overwrites the null-terminator of `str1` and, `str2` should not overlap `str1`. The string concatenation returns the pointer `str1`.

[2] Two segmentations, $\overline{b}^1_1 ... \overline{b}^1_n [?^1_n]$ and $\overline{b}^2_1 ... \overline{b}^2_n [?^2_n]$, are compatible if they have same lower and upper bounds, i.e., $\overline{b}^1_1 = \overline{b}^2_1$ and $\overline{b}^1_n = \overline{b}^2_n$.

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

Table 1: String functions syntax in `C`

- `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 [29]. 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.

5 Concrete Domain and Semantics

Our aim is to capture the presence of well-formed string in `C` character arrays, to avoid undesired execution behaviours that may be security relevant. Thus, we define the concrete value of a character array stigmatizing the occurrence of null characters in it and we present the notion of the so-called *string of interest* of an array of chars. The concrete semantics relative to the operations presented in Section 4 is also given.

5.1 Character Array Concrete Semantics

Let $\mathbb{C} = \mathsf{C} \cup \{\top_{\mathsf{c}}\}$ be a finite set of characters C representable by the character encoding in use equipped with a top value $\top_{\mathsf{c}}$ representing an unknown value, and let $\mathbb{M} = \mathbb{C}^*$ be the set of all the possible character array variables. Then, the operational semantics of character array variables ($\mathtt{m} \in \mathbb{M} \subseteq \mathbb{A}$) are concrete array environments $\mu \in \mathcal{R}_m$ mapping character arrays $\mathtt{m} \in \mathbb{M}$ to their values $\mu(\mathtt{m})$. Precisely,

$$\mu(\mathtt{m}) \in \mathcal{M} \triangleq \mathcal{R}_v \times \mathbb{E} \times \mathbb{E} \times (\mathbb{Z} \to (\mathbb{Z} \times \mathbb{C})) \times \mathbb{Z}$$

$$\mu(\mathtt{m}) = (\rho, \ell ow_{\mathtt{m}}, high_{\mathtt{m}}, M_{\mathtt{m}}, \mathrm{N}_{\mathtt{m}})$$

so that $\mathcal{R}_m \triangleq \mathbb{M} \to \mathcal{M}$, where $\mathcal{R}_v$ and $\mathbb{E}$ are the variable environment and the expression domain defined in Section 3.2 respectively, $\mathbb{Z}$ is the integer domain and, $\mathbb{C}$ is the execution character set.

Notice that, with respect to the concrete array environment θ defined in Section 3.2, the function μ returns as a last component the set of indexes which map to the string terminating characters.

$$\mathrm{N}_{\mathtt{m}} = \{i : i \in [[\![\ell ow_{\mathtt{m}}]\!]\rho, [\![high_{\mathtt{m}}]\!]\rho) \wedge M_{\mathtt{m}}(i) = (i, \text{'\textbackslash 0'})\}$$

Thus, $\mathcal{M}$ extends $\mathcal{A}$ (c.f., Section 3.2) by adding a parameter that takes into account the presence of null characters in a character array. Notice that for well-formed strings, $\mathrm{N}_{\mathtt{m}}$ cannot be empty. Moreover, character array elements which have not been initialized are mapped to the top value $\top_{\mathsf{c}}$ as they may be values already present in the memory assigned to the locations array itself.

Example 3 Consider the simple program below which concatenates two well-formed strings.

```c
1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5  char x[10] = "aaa";
6  char y[4] = "bbb";
7  strcat(x,y);
8 }
```

In this program, the string concatenation behaves correctly as the size of the destination array `x` is big

enough to store all the characters from the source array `y`. That said, the concrete value of `x` at program point `13` is given by $\mu(\mathtt{x}) = (\rho, \mathtt{0}, \mathtt{10}, M_{\mathtt{x},\mathtt{13}}, \mathrm{N}_{\mathtt{x},\mathtt{13}})$ where, the lower and the upper bounds of the character array `x` are clear from the context, $\mathrm{P}_{\mathtt{x},\mathtt{13}}$, the codomain of $M_{\mathtt{x},\mathtt{13}}$ (cf. Section 3.2), is the set $\{(\mathtt{0},\text{'a'}), (\mathtt{1},\text{'a'}), (\mathtt{2},\text{'a'}), (\mathtt{3},\text{'b'}), (\mathtt{4},\text{'b'}), (\mathtt{5},\text{'b'}), (\mathtt{6},\text{'\textbackslash 0'}), (\mathtt{7}, \top_c), (\mathtt{8}, \top_c), (\mathtt{9}, \top_c)\}$ and, $\mathrm{N}_{\mathtt{x},\mathtt{13}}$ is the singleton $\{\mathtt{6}\}$, being the array cell of index `6` the only one containing a null character.

5.1.1 *String of Interest*

We formally define the string of interest of a character array as the sequence of characters up to the termination character (included).

Definition 1 ***(string of interest)*** Let $\mathtt{m} \in \mathbb{M}$ be an array of characters with concrete value given by

$m = \mu(\mathtt{m}) = (\rho, \ell ow_{\mathtt{m}}, high_{\mathtt{m}}, M_{\mathtt{m}}, \mathrm{N}_{\mathtt{m}})$

and let k be the minimum element of $\mathrm{N}_{\mathtt{m}}$ (if it is non-empty). The string of interest of m is defined as follows,

$$s(m) = \begin{cases} \langle v_i : i \in [[\![\ell ow_{\mathtt{m}}]\!]\rho, k] \wedge M_{\mathtt{m}}(i) = (i, v) \rangle & \text{if } \mathrm{N}_{\mathtt{m}} \neq \emptyset \\ \mathtt{undef} & \text{otherwise} \end{cases}$$

Our definition of string of interest of character arrays allows us to distinguish well-formed strings and avoid bad usage of arrays of characters. Notice that, in the case where the null character occurs 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 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`). Notice that, the `strtok()`[3] function in `string.h` library can be used as a "remedy", as it substitutes delimiters occurring in a character array with null characters; however, if the character array given as input is not a well-formed string and none of the defined separators belongs to the considered array, then the function results in an undefined behaviour too.

[3] The `C` library function `char *strtok(char *str, const char *sep)` breaks string `str` into a series of tokens using the delimiter `sep`. This function returns a pointer to the last token found in the string. A null pointer is returned if there are no tokens left to retrieve [cite].

5.2 Concrete Domain

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

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

5.3 Concrete Semantics

In order to formalise the concrete semantics of the `C` standard library functions from `string.h` introduced in Section 4, the following auxiliaries functions *emb*, *cut* and *sub* need to be introduced.

Definition 2 ***(emb)*** Let $\mathtt{m}_1, \mathtt{m}_2 \in \mathbb{M}$ be two arrays of characters with concrete value given by

$m_1 = \mu(\mathtt{m}_1) = (\rho, \ell ow_{\mathtt{m}_1}, high_{\mathtt{m}_1}, M_{\mathtt{m}_1}, \mathrm{N}_{\mathtt{m}_1})$ and

$m_2 = \mu(\mathtt{m}_2) = (\rho, \ell ow_{\mathtt{m}_2}, high_{\mathtt{m}_2}, M_{\mathtt{m}_2}, \mathrm{N}_{\mathtt{m}_2})$

Moreover, let

$n_1, e_1 \in [[\![\ell ow_{\mathtt{m}_1}]\!]\rho, [\![high_{\mathtt{m}_1}]\!]\rho)$ and

$n_2, e_2 \in [[\![\ell ow_{\mathtt{m}_2}]\!]\rho, [\![high_{\mathtt{m}_2}]\!]\rho)$

be indexes which properly range in the arrays bounds with $n_1 \leqslant e_1$ and $n_2 \leqslant e_2$. Moreover, e_1 has to be equal to $n_1 + (e_2 - n_2)$. The embedding of the character array element values which occur from n_2 to e_2 in $\mathtt{m}_2$ into $\mathtt{m}_1$ from n_1 to e_1 is given by $emb(m_1, n_1, e_1, m_2, n_2, e_2) = m_1^{'}$ such that,

$$\begin{aligned}
&[\![\ell ow_{\mathtt{m}_1^{'}}]\!]\rho = [\![\ell ow_{\mathtt{m}_1}]\!]\rho \\
&[\![high_{\mathtt{m}_1^{'}}]\!]\rho = [\![high_{\mathtt{m}_1}]\!]\rho \\
&M_{\mathtt{m}_1^{'}} : \forall i \in [[\![\ell ow_{\mathtt{m}_1^{'}}]\!]\rho, n_1) \\
&\qquad M_{\mathtt{m}_1^{'}}(i) = (i, v) \text{ s.t. let } k = i \\
&\qquad\qquad M_{\mathtt{m}_1}(k) = (k, v) \\
&\quad \forall i \in [n_1, e_1] \\
&\qquad M_{\mathtt{m}_1^{'}}(i) = (i, v) \text{ s.t.} \\
&\qquad\quad \text{let } k = n_2 + (i - n_1) \\
&\qquad\qquad M_{\mathtt{m}_2}(k) = (k, v)
\end{aligned}$$

$$\forall i \in (e_1, [\![high_{\mathtt{m}_1'}]\!]\rho) \quad M_{\mathtt{m}_1'}(i) = (i, v) \text{ s.t. let } k = i \quad M_{\mathtt{m}_1}(k) = (k, v)$$

$$\mathrm{N}_{\mathtt{m}_1'} = (\mathrm{N}_{\mathtt{m}_1} - \{i : i \in [n_1, e_1]\}) \cup \{i : i \in [n_2, e_2] \wedge M_{\mathtt{m}_2}(i) = (i, \text{'\textbackslash 0'})\}$$

Definition 3 (*cut*) Let $\mathtt{m} \in \mathbb{M}$ be an array of characters with concrete value given by

$m = \mu(\mathtt{m}) = (\rho, \ell ow_{\mathtt{m}}, high_{\mathtt{m}}, M_{\mathtt{m}}, \mathrm{N}_{\mathtt{m}})$

and let

$n, e \in [[\![\ell ow_{\mathtt{m}}]\!]\rho, [\![high_{\mathtt{m}}]\!]\rho)$

be two indexes which properly range in the array bounds with $n \leqslant e$. The character array sub-value from n to e of $\mathtt{m}$ is given by $cut(m, n, e) = m'$ such that,

$$[\![\ell ow_{\mathtt{m}'}]\!]\rho = n$$

$$[\![high_{\mathtt{m}'}]\!]\rho = e + 1$$

$$M_{\mathtt{m}'} : \forall i \in [[\![\ell ow_{\mathtt{m}'}]\!]\rho, [\![high_{\mathtt{m}'}]\!]\rho) \quad M_{\mathtt{m}'}(i) = (i, v) \text{ s.t. let } k = i \quad M_{\mathtt{m}}(k) = (k, v)$$

$$\mathrm{N}_{\mathtt{m}'} = \mathrm{N}_{\mathtt{m}} - \{i : i \notin [[\![\ell ow_{\mathtt{m}'}]\!]\rho, [\![high_{\mathtt{m}'}]\!]\rho)\}$$

Definition 4 (*sub*) Let $\mathtt{m} \in \mathbb{M}$ be an array of characters with concrete value given by

$m = \mu(\mathtt{m}) = (\rho, \ell ow_{\mathtt{m}}, high_{\mathtt{m}}, M_{\mathtt{m}}, \mathrm{N}_{\mathtt{m}})$

and let

$z \in [[\![\ell ow_{\mathtt{m}}]\!]\rho, [\![high_{\mathtt{m}}]\!]\rho)$

be an index which properly ranges in the array bounds and c be a character in the execution set. The substitution of the array element occurring at position z with c is given by $sub(m, z, c) = m'$ such that,

$$[\![\ell ow_{\mathtt{m}'}]\!]\rho = [\![\ell ow_{\mathtt{m}}]\!]\rho$$

$$[\![high_{\mathtt{m}'}]\!]\rho = [\![high_{\mathtt{m}}]\!]\rho$$

$$M_{\mathtt{m}'} : \forall i \in [[\![\ell ow_{\mathtt{m}'}]\!]\rho, z) \quad M_{\mathtt{m}'}(i) = (i, v) \text{ s.t. let } k = i \quad M_{\mathtt{m}}(k) = (k, v)$$

$$i = z \quad M_{\mathtt{m}'}(z) = (z, c)$$

$$\forall i \in (z, [\![high_{\mathtt{m}'}]\!]\rho) \quad M_{\mathtt{m}'}(i) = (i, v) \text{ s.t. let } k = i \quad M_{\mathtt{m}}(k) = (k, v)$$

$$\mathrm{N}_{\mathtt{m}'} = \begin{cases} \mathrm{N}_{\mathtt{m}} & \text{if } (z \in N_{\mathtt{m}} \wedge c \text{ is null}) \vee (z \notin N_{\mathtt{m}} \wedge c \text{ is not null}) \\ \mathrm{N}_{\mathtt{m}} - \{z\} & \text{if } z \in N_{\mathtt{m}} \wedge c \text{ is not null} \\ \mathrm{N}_{\mathtt{m}} \cup \{z\} & \text{otherwise} \end{cases}$$

5.3.1 Array Access

The semantics operator $\mathfrak{C}$, given the statement $\mathtt{access}_j$ and a set of concrete character array values $\mathtt{M}$ in $\mathcal{P}(\mathcal{M})$ as parameter, returns a value in $\mathbb{C}$ (cf. Section 5.1). In particular, $\mathtt{access}_j(\mathtt{M})$ returns the character v which occurs at position j if all the character array values in $\mathtt{M}$ 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 $\mathtt{M}$; otherwise it returns $\top_{\mathtt{C}}$. Formally,

$$\mathfrak{C}[\![\mathtt{access}_j]\!](\mathtt{M}) = \begin{cases} v & \text{if } \forall m \in \mathtt{M} : j \in [[\![\ell ow_{\mathtt{m}}]\!]\rho, [\![high_{\mathtt{m}}]\!]\rho) \text{ and } M_{\mathtt{m}}(j) = (j, v) \\ \top_{\mathtt{C}} & \text{otherwise} \end{cases}$$

5.3.2 String Concatenation

The semantics operator $\mathfrak{M}$, given a statement and some sets of concrete character array values in $\mathcal{P}(\mathcal{M})$ as parameters, returns a set of concrete character array values. When applied to $\mathtt{strcat}(\mathtt{M}_1, \mathtt{M}_2)$, it returns all the possible embeddings in $\mathtt{M}_1$ of a string of interest taken from $\mathtt{M}_2$ if all the character array values (which belong to both $\mathtt{M}_1$ and $\mathtt{M}_2$) contain a well-formed string and the condition on the size of the destination character array values is fulfilled; otherwise it returns $\top_{\mathtt{M}}$. Formally,

$$\mathfrak{M}[\![\mathtt{strcat}]\!](\mathtt{M}_1, \mathtt{M}_2) = \begin{cases} \mathtt{M}_1' & \text{if } \forall m_1 \in \mathtt{M}_1 : \forall m_2 \in \mathtt{M}_2 : s(m_1) \neq \mathtt{undef} \neq s(m_2) \text{ and } \mathtt{size.condition} \text{ is } \mathtt{true} \\ \top_{\mathtt{M}} & \text{otherwise} \end{cases}$$

where $\mathtt{M}_1'$ is the set of $emb(m_1, n_1, e_1, m_2, n_2, e_2)$ such that,

$m_1 \in \mathtt{M}_1$

$n_1 = min_{\mathrm{N}_{\mathtt{m}_1}}, \ e_1 = n_1 + (min_{\mathrm{N}_{\mathtt{m}_2}} - [\![\ell ow_{\mathtt{m}_2}]\!]\rho)$

$m_2 \in \mathtt{M}_2$

$n_2 = [\![\ell ow_{\mathtt{m}_2}]\!]\rho, \ e_2 = min_{\mathrm{N}_{\mathtt{m}_2}}$

and the $\mathtt{size.condition}$ is $\mathtt{true}$ if

$$([\![high_{\mathtt{m}_1}]\!]\rho - [\![\ell ow_{\mathtt{m}_1}]\!]\rho) \geqslant [(min_{\mathrm{N}_{\mathtt{m}_1}} - [\![\ell ow_{\mathtt{m}_1}]\!]\rho - 1) + (min_{\mathrm{N}_{\mathtt{m}_2}} - [\![\ell ow_{\mathtt{m}_2}]\!]\rho)]$$

5.3.3 String Character

The semantics operator $\mathfrak{M}$, when applied to $\mathtt{strchr}_v(\mathtt{M})$, returns the set of string of interest suffixes in $\mathtt{M}$ from the index corresponding to the first occurrence of the character v if all the character array values in $\mathtt{M}$ contain a well-formed string containing v. Otherwise, if all the character array values in $\mathtt{M}$ contain a well-formed string

in which does not occur the character v, it returns the emptyset (i.e., $\bot_{\mathbf{M}}$); otherwise it returns $\top_{\mathbf{M}}$. Formally,

$$\mathfrak{M}[\![\mathtt{strchr}_v]\!](\mathtt{M}) = \begin{cases} \mathtt{S} & \text{if } \forall m \in \mathtt{M} : s(m) \neq \mathtt{undef} \text{ and } v \in s(m) \\ \bot_{\mathbf{M}} & \text{if } \forall m \in \mathtt{M} : s(m) \neq \mathtt{undef} \text{ and } v \notin s(m) \\ \top_{\mathbf{M}} & \text{otherwise} \end{cases}$$

In particular, $\mathtt{S}$ is the set of $cut(m, n, e)$ such that,

$m \in \mathtt{M}$

$n = min\{i : i \in [[\![\ell ow_{\mathtt{m}}]\!]\rho, min_{\mathtt{N_m}}] \wedge M_{\mathtt{m}}(i) = (i, v)\}$

$e = min_{\mathtt{N_m}}$

5.3.4 String Compare

The semantics operator $\mathfrak{P}$, given the statement `strcmp` and two sets of concrete character array values $\mathtt{M}_1, \mathtt{M}_2$ in $\mathcal{P}(\mathcal{M})$ as parameters, returns a value in the set of integers equipped with a top element, i.e., $\mathbb{Z} \cup \top_{\mathsf{Z}}$. In particular, $\mathtt{strcmp}(\mathtt{M}_1, \mathtt{M}_2)$ returns an integer value n which denotes the lexicographic order between strings of interest in $\mathtt{M}_1$ and $\mathtt{M}_2$ if all the character array values (which belong to both $\mathtt{M}_1$ and $\mathtt{M}_2$) contain a well-formed string whose elements are fully initialized; otherwise it returns$\top_{\mathsf{Z}}$.

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

$$\mathfrak{P}[\![\mathtt{strcmp}]\!](\mathtt{M}_1, \mathtt{M}_2) = \begin{cases} n & \text{if } \forall m_1 \in \mathtt{M}_1 : \forall m_2 \in \mathtt{M}_2 : \\ & \quad s(m_1) \neq \mathtt{undef} \neq s(m_2) \text{ and} \\ & \quad\quad \top_{\mathsf{C}} \notin s(m_1), s(m_2) \\ \top_{\mathsf{Z}} & \text{otherwise} \end{cases}$$

In particular, n is obtained as defined in Algorithm 1.

Algorithm 1 Computing n

```
1: i1 = [[ℓow_m1]]ρ
2: i2 = [[ℓow_m2]]ρ
3: for i1 ∈ [[[ℓow_m1]]ρ, min_Nm1] ∧ i2 ∈ [[[ℓow_m2]]ρ, min_Nm2] do
4:     n = v_i1 − v_i2
5:     if n ≠ 0 then
6:         return n
7:     else
8:         i1 = i1 + 1
9:         i2 = i2 + 1
10: return n
```

Notice that v_i denotes the character array element value v which occurs at position i, i.e., $M_{\mathtt{m}}(i) = (i, v)$.

5.3.5 String Copy

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

$$\mathfrak{M}[\![\mathtt{strcpy}]\!](\mathtt{M}_1, \mathtt{M}_2) = \begin{cases} \mathtt{M}_1^{'} & \text{if } \forall m_1 \in \mathtt{M}_1 : \forall m_2 \in \mathtt{M}_2 : \\ & \quad s(m_1) \neq \mathtt{undef} \neq s(m_2) \text{ and} \\ & \quad\quad \mathtt{size.condition} \text{ is } \mathtt{true} \\ \top_{\mathbf{M}} & \text{otherwise} \end{cases}$$

where $\mathtt{M}_1^{'}$ is the set of $emb(m_1, n_1, e_1, m_2, n_2, e_2)$ such that,

$m_1 \in \mathtt{M}_1$

$n_1 = [\![\ell ow_{\mathtt{m}_1}]\!]\rho$, $e_1 = n_1 + (min_{\mathtt{N_{m_2}}} - [\![\ell ow_{\mathtt{m}_2}]\!]\rho)$

$m_2 \in \mathtt{M}_2$

$n_2 = [\![\ell ow_{\mathtt{m}_2}]\!]\rho$, $e_2 = min_{\mathtt{N_{m_2}}}$

and the `size.condition` is `true` if

$([\![high_{\mathtt{m}_1}]\!]\rho - [\![\ell ow_{\mathtt{m}_1}]\!]\rho) \geqslant (min_{\mathtt{N_{m_2}}} - [\![\ell ow_{\mathtt{m}_2}]\!]\rho)$

5.3.6 String Length

The semantics operator $\mathfrak{Z}$, given the statement `strlen` and a set of concrete character array values $\mathtt{M}$ in $\mathcal{P}(\mathcal{M})$ as parameter, returns a value in the set of integers equipped with a top element, i.e., $\mathbb{Z} \cup \top_{\mathsf{Z}}$. In particular, $\mathtt{strlen}(\mathtt{M})$ 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 $\mathtt{M}$ if all the character array values in $\mathtt{M}$ contain a well-formed string of the same length; otherwise it returns $\top_{\mathsf{Z}}$. Formally,

$$\mathfrak{Z}[\![\mathtt{strlen}]\!](\mathtt{M}) = \begin{cases} n & \text{if } \forall m \in \mathtt{M} : s(m) \neq \mathtt{undef} \text{ and} \\ & \quad min_{\mathtt{N_m}} - [\![\ell ow_{\mathtt{m}}]\!]\rho = n \\ \top_{\mathsf{Z}} & \text{otherwise} \end{cases}$$

5.3.7 Array update

The semantics operator $\mathfrak{M}$, when applied to $\mathtt{update}_{j,v}$ $(\mathtt{M})$, returns the set of character array values in $\mathtt{M}$ 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 $\mathtt{M}$; otherwise it returns $\top_{\mathbf{M}}$.

Formally,

$$\mathfrak{M}[\![\texttt{update}_{j,v}]\!](\texttt{M}) = \begin{cases} \texttt{M}' & \text{if } \forall m \in \texttt{M} : j \in [[\![\ell ow_{\texttt{m}}]\!]\rho, [\![high_{\texttt{m}})]\!]\rho) \\ \top_{\texttt{M}} & \text{otherwise} \end{cases}$$

In particular, $\texttt{M}'$ is the set of $sub(m, j, v)$.

6 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 $\mathbf{M}$, made of sets of character array values, and its concrete semantics of some operations of interest. In the following we formalize the M-String abstract domain, which approximates elements in $\mathbf{M}$, and its semantics for which soundness is proved.

6.1 Character Array Abstract Domain

the M-String ($\overline{\mathbf{M}}$) abstract domain approximates sets of character arrays with a pair of segmentations that highlights the nature of their *strings of interest*. the elements of the domain are split segmentation abstract predicates. As for FunArray (recalled in Section 3.2), segments capture sequences of identical abstract values, and are delimited by the so-called segment bounds. More precisely, the M-String abstract domain is a functor given by $\overline{\mathbf{M}}(\overline{\mathbf{B}}, \overline{\mathbf{C}}, \overline{\mathbf{R}})$.

1) $\overline{\mathbf{B}}$ denotes the abstraction of segment bounds, equipped with the following operations: least upper bound between subsequent segment bounds ($\sqcup_{\overline{\mathbf{B}}}[\overline{b}_i, \overline{b}_{i+1})$), addition ($+_{\overline{\mathbf{B}}}$), and subtraction ($-_{\overline{\mathbf{B}}}$).
2) $\overline{\mathbf{C}}$ is the abstraction of the character array elements, it is signed, it contains the value 0, and it is equipped with `is_null`, a special monotonic function lifting abstract elements in $\overline{\mathcal{C}}$ to a value in the set {`true`, `false`, `maybe`}, and with subtraction ($-_{\overline{\mathbf{C}}}$).
3) $\overline{\mathbf{R}}$ denotes the abstraction of scalar variable environments (cf. Section 3.2). Namely, the constant propagation domain on the set of variables $\mathbb{X}$.

M-String is a complete lattice

$$(\overline{\mathcal{M}}, \leqslant_{\overline{\mathbf{M}}}, \bot_{\overline{\mathbf{M}}}, \top_{\overline{\mathbf{M}}}, \sqcap_{\overline{\mathbf{M}}}, \sqcup_{\overline{\mathbf{M}}})$$

where,

- $\overline{\mathcal{M}} \triangleq (\overline{\mathcal{M}}_s, \overline{\mathcal{M}}_{ns}) \cup \{\bot_{\overline{\mathbf{M}}}, \top_{\overline{\mathbf{M}}}\}$
  - $\overline{\mathcal{M}}_s$ corresponds to
    $\{\overline{\mathcal{S}}_{sb} \times \overline{\mathcal{S}}^k_{sm} \times \overline{\mathcal{S}}_{se} \mid k \geqslant 0\} \cup \overline{\mathcal{S}}_{se} \cup \{\emptyset\}$
    and it represents the segmentation of the strings of interest of a set of character arrays. Here,
    $\overline{\mathcal{S}}_{sb} = \{\overline{\mathcal{B}} \times \overline{\mathcal{C}}\}$
    $\overline{\mathcal{S}}_{sm} = \{\overline{\mathcal{B}} \times \overline{\mathcal{C}} \times \{␣, ?\}\}$
    $\overline{\mathcal{S}}_{se} = \{\overline{\mathcal{B}} \times \{␣\}\}$
  - $\overline{\mathcal{M}}_{ns}$ corresponds to
    $\{\overline{\mathcal{S}}_{nsb} \times \overline{\mathcal{S}}^k_{nsm} \times \overline{\mathcal{S}}_{nse} \mid k \geqslant 0\} \cup \{\emptyset\}$
    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. Here,
    $\overline{\mathcal{S}}_{nsb} = \overline{\mathcal{S}}_{sb}$
    $\overline{\mathcal{S}}_{nsm} = \overline{\mathcal{S}}_{sm}$
    $\overline{\mathcal{S}}_{nse} = \{\overline{\mathcal{B}} \times \{␣, ?\}\}$

In particular (c.f. Section 3.2):

1) $\overline{b}_i \in \overline{\mathcal{B}}$ denotes the segment bounds, chosen in abstract domain $\overline{\mathbf{B}}$, such that $i \in [1, n]$ and $n > 1$ ($\overline{b}_1$ and $\overline{b}_n$ respectively represent the segmentation lower and upper bound). Notice that, a segment bound approximates a set of indexes (i.e., positive integers $\mathbb{Z}^+$). For the sake of readability, we apply arithmetic operators on $\overline{b}_i$ directly. For instance, $\overline{b} +_{\overline{\mathbf{B}}} 1$ should be read as $\alpha_{\overline{\mathbf{B}}}(\{i+1 : i \in \gamma_{\overline{\mathbf{B}}}(\overline{b})\})$.
2) $\overline{p}_i \in \overline{\mathcal{C}}$ are abstract predicates, chosen in an abstract domain $\overline{\mathbf{C}}$, denoting possible values of pairs (index, character array element value) in a segment, for relational abstraction, character array elements otherwise.
3) the question mark ?, if present, indicates that the preceding segment might be empty, while ␣ indicates a non-empty segment. Non-empty segments are not marked.
   Notice that, in M-String, the upper bound of the left hand side parameter of a split segmentation, if different from the emptyset, will be always followed by a text visible space ␣. This means that all the character arrays it approximates surely contain a well-formed string.

the elements in $\overline{\mathcal{M}}$ are split segmentation abstract predicates of the form $\overline{m} = (s, ns)$. Let $\texttt{m} \in \mathbb{M}$ be an array of characters with concrete value denoted by m. For instance, if the string of interest of m is `null` (see Section 5.1.1) then, $\overline{m}$ is equal to $(\overline{b}_1␣, \emptyset)$ if the size of m is equal to 1, $(\overline{b}_1␣, \overline{b}_2\overline{p}_2\overline{b}_3[?_3]\overline{b}_n[?_n])$ otherwise.

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

- Let $\overline{m}_1$ and $\overline{m}_2$ be two abstract values in the M-String domain. the partial ordering on $\overline{\mathbf{M}}$ is defined as follows. $\forall \overline{m} \in \overline{\mathcal{M}} : \bot_{\overline{\mathbf{M}}} \leqslant_{\overline{\mathbf{M}}} \overline{m}$. If $\overline{m}_1 \neq \bot_{\overline{\mathbf{M}}}$ and $\overline{m}_2 \neq \bot_{\overline{\mathbf{M}}}$, $\overline{m}_1$ and $\overline{m}_2$ are comparable only if they can be normalised through *unify*. If this is the case, let $unify(\overline{m}_1, \overline{m}_2) = \overline{m}_1^{'}, \overline{m}_2^{'}$, $\overline{m}_1^{'} \leqslant_{\overline{\mathbf{M}}} \overline{m}_2^{'}$ if and only if for each segment the relation $\leqslant_{\overline{\mathbf{C}}}$ holds.
  Here, *unify* refers to the *unification* algorithm originally defined in [13] (mentioned in Section 3.2) and tweaked in [6] to modify two split segmentations so that they coincide. Take $\overline{m}_1$ and $\overline{m}_2$ to be compatible if their parameters have common lower and upper bounds of s and ns. Then,
  $$\begin{aligned} unify(\overline{m}_1, \overline{m}_2) &= (unify(s_1, s_2), unify(ns_1, ns_2)) \\ &= (s_1^{'}, ns_1^{'}), (s_2^{'}, ns_2^{'}) \\ &= \overline{m}_1^{'}, \overline{m}_2^{'} \end{aligned}$$
- $\bot_{\overline{\mathbf{M}}}$, $\top_{\overline{\mathbf{M}}}$ are special elements denoting the bottom/-top element of the lattice.
- $\sqcap_{\overline{\mathbf{M}}}$ represents the meet operator, that defines the greatest lower bound between abstract elements.
  Let $unify(\overline{m}_1, \overline{m}_2) = \overline{m}_1^{'}, \overline{m}_2^{'}$, then $\overline{m}_1^{'} \sqcap_{\overline{\mathbf{M}}} \overline{m}_2^{'} = (\overline{b}_1^1 \sqcap_{\overline{\mathbf{B}}} \overline{b}_1^2 \overline{p}_1^1 \sqcap_{\overline{\mathbf{C}}} \overline{p}_1^2 \overline{b}_2^1 \sqcap_{\overline{\mathbf{B}}} \overline{b}_2^2 [?_2^1] \curlywedge [?_2^2] ... \overline{b}_k^1 \sqcap_{\overline{\mathbf{B}}} \overline{b}_k^2 \textvisiblespace \curlywedge \textvisiblespace, \overline{b}_{k+1}^1 \sqcap_{\overline{\mathbf{B}}} \overline{b}_{k+1}^2 \overline{p}_{k+1}^1 \sqcap_{\overline{\mathbf{C}}} \overline{p}_{k+1}^2 \overline{b}_{k+2}^1 \sqcap_{\overline{\mathbf{B}}} \overline{b}_{k+2}^2 [?_{k+2}^1] \curlywedge [?_{k+2}^2] ... \overline{b}_n^1 \sqcap_{\overline{\mathbf{B}}} \overline{b}_n^2 [?_n^1] \curlywedge [?_n^2])$. If $\overline{m}_1$ and $\overline{m}_2$ cannot be unified then the greatest lower bound between $\overline{m}_1$ and $\overline{m}_2$ is equal to $\bot_{\overline{\mathbf{M}}}$.
- $\sqcup_{\overline{\mathbf{M}}}$ represents the join operator, that defines the least upper bound between two abstract elements.
  Let $unify(\overline{m}_1, \overline{m}_2) = \overline{m}_1^{'}, \overline{m}_2^{'}$, then $\overline{m}_1^{'} \sqcup_{\overline{\mathbf{M}}} \overline{m}_2^{'} = (\overline{b}_1^1 \sqcup_{\overline{\mathbf{B}}} \overline{b}_1^2 \overline{p}_1^1 \sqcup_{\overline{\mathbf{C}}} \overline{p}_1^2 \overline{b}_2^1 \sqcup_{\overline{\mathbf{B}}} \overline{b}_2^2 [?_2^1] \curlyvee [?_2^2] ... \overline{b}_k^1 \sqcup_{\overline{\mathbf{B}}} \overline{b}_k^2 \textvisiblespace \curlyvee \textvisiblespace, \overline{b}_{k+1}^1 \sqcup_{\overline{\mathbf{B}}} \overline{b}_{k+1}^2 \overline{p}_{k+1}^1 \sqcup_{\overline{\mathbf{C}}} \overline{p}_{k+1}^2 \overline{b}_{k+2}^1 \sqcup_{\overline{\mathbf{B}}} \overline{b}_{k+2}^2 [?_{k+2}^1] \curlyvee [?_{k+2}^2] ... \overline{b}_n^1 \sqcup_{\overline{\mathbf{B}}} \overline{b}_n^2 [?_n^1] \curlyvee [?_n^2])$. If $\overline{m}_1$ and $\overline{m}_2$ cannot be unified then the least upper bound between $\overline{m}_1$ and $\overline{m}_2$ is equal to $\top_{\overline{\mathbf{M}}}$.

M-String is equipped with a widening $\nabla_{\overline{\mathbf{M}}}$ and a narrowing $\Delta_{\overline{\mathbf{M}}}$ operators, whose definitions follow those in Section 11.5 of [13].

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

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

Formally, we firstly define the concretization function of a generic segment $(\overline{b}\overline{p}\overline{b}'[?])$ (regardless of what part of the split it is part of), following [13], which corresponds to the set of character array values whose elements in the segment $[\overline{b}, \overline{b}'[?])$ satisfy the predicate $\overline{p}$. Let l and h denote the character array lower and upper bound respectively, then:

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

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

We remind that only the upper bound of $\overline{m}.s$ is explicitly followed by a visible space (␣), preserving a special meaning. Precisely, $\overline{b}\textvisiblespace$ is equivalent to the segment $\overline{b}\textvisiblespace\overline{p}\overline{b}'$ such that $\overline{b}' = \overline{b} +_{\overline{\mathbf{B}}} 1$ and $\overline{p}$ is null.

an abstract element in the M-String domain is a pair of segmentations. Thus, we define the concretization function of the possible s and ns belonging to a character array abstract predicate $\overline{m}$, i.e., $\gamma_{\overline{\mathbf{M}}}^{\star} \in \overline{\mathcal{M}} \to \overline{\mathcal{R}} \to \mathcal{P}(\mathcal{M})$.

$$\begin{aligned} &\gamma_{\overline{\mathbf{M}}}^{\star}(\overline{m}.s)\overline{\rho} \\ &\triangleq \{(\rho, l, h, M, \mathrm{N}) \in \dotplus_{i=1}^{k}{}_{\mathbf{M}} \gamma_{\overline{\mathbf{M}}}^{*}(\overline{b}_i[\textvisiblespace]\overline{p}_i\overline{b}_{i+1}[?^{i+1}])\overline{\rho} \mid \\ &\quad \exists b_1, b_k : \ b_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_1), b_k \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_k) \wedge \\ &\qquad b_1 = [\![l]\!]\rho \wedge b_k + 1 \leqslant [\![h]\!]\rho\} \\ &\quad \text{if } \overline{m}.s = \overline{b}_1\overline{p}_1\overline{b}_2[?^2]...\overline{b}_k\textvisiblespace \\ &\triangleq \gamma_{\overline{\mathbf{M}}}^{*}(\overline{b}_1\textvisiblespace)\overline{\rho} \text{ if } \overline{m}.s = \overline{b}_1\textvisiblespace \\ &\triangleq \emptyset \text{ otherwise.} \end{aligned}$$

$$\begin{aligned} &\gamma_{\overline{\mathbf{M}}}^{\star}(\overline{m}.ns)\overline{\rho} \\ &\triangleq \{(\rho, l, h, M, \mathrm{N}) \in \dotplus_{i=1}^{n-1}{}_{\mathbf{M}} \gamma_{\overline{\mathbf{M}}}^{*}(\overline{b}_i\overline{p}_i\overline{b}_{i+1}[?^{i+1}])\overline{\rho} \mid \\ &\quad \exists b_1, b_n : \ b_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_1), b_n \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_n) \wedge \\ &\qquad b_1 = [\![l]\!]\rho \wedge b_n = [\![h]\!]\rho\} \\ &\quad \text{if } \overline{m}.ns = \overline{b}_1\overline{p}_1\overline{b}_2[?^2]...\overline{b}_n[?^n] \\ &\triangleq \{(\rho, l, h, M, \mathrm{N}) \in \dotplus_{i=k+1}^{n-1}{}_{\mathbf{M}} \gamma_{\overline{\mathbf{M}}}^{*}(\overline{b}_i\overline{p}_i\overline{b}_{i+1}[?^{i+1}])\overline{\rho} \mid \\ &\quad \exists b_{k+1}, b_n : \ b_{k+1} \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_{k+1}), b_n \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_n) \wedge \\ &\qquad [\![l]\!]\rho < b_{k+1} \wedge b_n = [\![h]\!]\rho\} \\ &\quad \text{if } \overline{m}.ns = \overline{b}_{k+1}\overline{p}_{k+1}\overline{b}_{k+2}[?^{k+2}]...\overline{b}_n[?^n] \\ &\triangleq \emptyset \text{ otherwise.} \end{aligned}$$

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

$$\gamma_{\overline{\mathbf{M}}}(\overline{m})\overline{\rho} \triangleq \{(\rho, l, h, M, \mathrm{N}) \in \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{m}.s)\overline{\rho} +_{\mathbf{M}} \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{m}.ns)\overline{\rho} \mid \exists b_1, b_n : \ b_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_1), b_n \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_n) \wedge b_1 = [\![l]\!]\rho \wedge b_n = [\![h]\!]\rho\}$$

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

6.1.1 Galois Connection

Let us show that $(\alpha_{\overline{\mathbf{M}}}, \gamma_{\overline{\mathbf{M}}})$ is a Galois connection.

Definition 5 ***(invalid segment)*** Given a generic segment $\overline{b}\overline{p}\overline{b}'[?]$, it is considered invalid if its segment abstract predicate $\overline{p}$ is equal to $\bot_{\overline{\mathbf{C}}}$ and its upper bound $\overline{b}'$ is not followed by a question mark.

Theorem 1 *Let the abstraction function $\alpha_{\overline{\mathbf{M}}}$ be defined by $\alpha_{\overline{\mathbf{M}}} = \lambda \mathsf{Y}. \sqcap_{\overline{\mathbf{M}}} \{\overline{m} : \gamma_{\overline{\mathbf{M}}}(\overline{m}) \subseteq \mathsf{Y}\}$. Then,*

$$\langle \mathcal{P}(\mathcal{M}), \subseteq_{\mathbf{M}} \rangle \underset{\alpha_{\overline{\mathbf{M}}}}{\overset{\gamma_{\overline{\mathbf{M}}}}{\leftrightarrows}} \langle \overline{\mathcal{M}}, \leqslant_{\overline{\mathbf{M}}} \rangle$$

Proof By Theorem 1.1 in [9], we only need to prove that $\gamma_{\overline{\mathbf{M}}}$ is a complete meet morphism. Formally, we have to prove that

$$\gamma_{\overline{\mathbf{M}}}\left(\bigsqcap_{\overline{m} \in \overline{\mathsf{X}}} {}_{\overline{\mathbf{M}}} \overline{m}\right) = \bigcap_{\overline{m} \in \overline{\mathsf{X}}} {}_{\mathbf{M}} \gamma_{\overline{\mathbf{M}}}(\overline{m})$$

where $\overline{\mathsf{X}}$ denotes a set of abstract elements.

By definition of $\sqcap_{\overline{\mathbf{M}}}$ we can have the two following cases:

1. For compatible abstract elements in $\overline{\mathsf{X}}$ whose meet does not result in an invalid abstract element, then we have the following inference chain:

   $$\gamma_{\overline{\mathbf{M}}}\left(\bigsqcap_{\overline{m} \in \overline{\mathsf{X}}} {}_{\overline{\mathbf{M}}} \overline{m}\right)$$

   by definition of $\sqcap_{\overline{\mathbf{M}}}$

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

   by definition of $\gamma_{\overline{\mathbf{M}}}$

   $$= \{(\rho, l, h, M\mathrm{N}) \in \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{m}'.s)\overline{\rho} +_{\mathbf{M}} \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{m}'.ns)\overline{\rho} \mid \exists b_1, b_n : b_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_1), b_n \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_n) \wedge b_1 = [\![l]\!]\rho \wedge b_n = [\![h]\!]\rho\}$$

   by Definition 5

   $$= \bigcap_{\overline{m} \in \overline{\mathsf{X}}} {}_{\mathbf{M}} \{(\rho, l, h, \mathrm{C}, \mathrm{N}) \in \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{m}.s)\overline{\rho} +_{\mathbf{M}} \gamma^{\star}_{\overline{\mathbf{M}}}(\overline{m}.ns)\overline{\rho} \mid \exists b_1, b_n : \ b_1 \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_1), b_n \in \gamma_{\overline{\mathbf{B}}}(\overline{b}_n) \wedge b_1 = [\![l]\!]\rho \wedge b_n = [\![h]\!]\rho\}$$

   by definition of $\gamma_{\overline{\mathbf{M}}}$

   $$= \bigcap_{\overline{m} \in \overline{\mathsf{X}}} {}_{\mathbf{M}} \gamma_{\overline{\mathbf{M}}}(\overline{m})$$

2. Otherwise, $\gamma_{\overline{\mathbf{M}}}\left(\bigsqcap_{\overline{m} \in \overline{\mathsf{X}}} {}_{\overline{\mathbf{M}}} \overline{m}\right) = \gamma_{\overline{\mathbf{M}}}(\bot_{\overline{\mathbf{M}}}) = \emptyset$.

   Then, $\bigcap_{\overline{m} \in \overline{\mathsf{X}}} {}_{\mathbf{M}} \gamma_{\overline{\mathbf{M}}}(\overline{m}) = \emptyset$.

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

Definition 6 (*lift* operation) Let `s` be a character array. Given its concrete value $\mu(\mathtt{s})$ and its abstract value $\alpha_{\overline{\mathbf{M}}}(\mu(\mathtt{s}))$ in M-String, we define the *lift* operation of `s` as follows:

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

Definition 7 (*lower* operation) Let `s` be a character array and let $\overline{\mathtt{s}}$ denote $lift(\mathtt{s})$ (cf. Definition 6). Given the concrete value of `s`, i.e., $\mu(\mathtt{s})$, its abstract value $\alpha_{\overline{\mathbf{M}}}(\mu(\mathtt{s}))$ and the concretization function on M-String $\gamma_{\overline{\mathbf{M}}}$, we define the *lower* operation of $\overline{\mathtt{s}}$ as follows:

$$lower(\overline{\mathtt{s}}) = \gamma_{\overline{\mathbf{M}}}(\alpha_{\overline{\mathbf{M}}}(\mu(\mathtt{s}))).$$

6.2 Abstract Semantics

The abstract semantics of the operators introduced in 5.3 is formalized below.

We refer to $emb_{\overline{\mathbf{M}}}$ as the embedding function between split segmentations. Precisely,

$$emb_{\overline{\mathbf{M}}}(\overline{m}_1, \overline{n}_1, \overline{e}_1, \overline{m}_2, \overline{n}_2, \overline{e}_2)$$

embeds the segment abstract predicates which occur from $\overline{n}_2$ to $\overline{e}_2$ (included) in $\overline{m}_2$ into $\overline{m}_1$ from $\overline{n}_1$ to $\overline{e}_1$ where $\overline{n}_1, \overline{e}_1 \in [\ell_{\overline{m}_1}, u_{\overline{m}_1})$, $\overline{n}_2, \overline{e}_2 \in [\ell_{\overline{m}_2}, u_{\overline{m}_2})$ ($\ell_{\overline{m}}$ and $u_{\overline{m}}$ respectively denote the lower and the upper bound of $\overline{m}$) and $\overline{m}_1$ is modified accordingly. Moreover, $\overline{e}_1$ corresponds to $\overline{n}_1 +_{\overline{\mathbf{B}}} (\overline{e}_2 -_{\overline{\mathbf{B}}} \overline{n}_2)$ and question marks, if present, are left unchanged. Notice that, in the case in which the upper bound of the string of interest segmentation abstraction is considered as a starting or ending point for the embedding, the `is_null` segment abstract predicate is embedded too.

We define the minimum length of a split segmentation abstract predicate $\overline{m}$, i.e., $min\ell en_{\overline{m}}$. Precisely, if $\overline{m}.ns$ is different from the emptyset and its upper bound ($u_{\overline{m}.ns}$) is followed by a question mark then $min\ell en_{\overline{m}}$ corresponds to the difference between $\overline{b}_k$ and the lower bound of $\overline{m}$ ($\ell_{\overline{m}}$), with $\overline{b}_k$ the greatest segment bound in $\overline{m}.ns$ not followed by a question mark. On the other hand, if $u_{\overline{m}.ns}$ is not followed by a question mark then $min\ell en_{\overline{m}}$ is the difference between the upper bound of $\overline{m}$ ($u_{\overline{m}}$) and $\ell_{\overline{m}}$ (here $u_{\overline{m}}$ and $u_{\overline{m}.ns}$ coincide). In the case in which $\overline{m}.ns$ is equal to the emptyset then if $\overline{m}.s$

approximates `null` strings of interest then $min\ell en_{\overline{m}}$ is equal to 1, the difference between the upper bound of $\overline{m}.s$ ($u_{\overline{m}.s}$) plus 1 and the lower bound of $\overline{m}.s$ ($\ell_{\overline{m}.s}$) otherwise.

6.2.1 Abstract Array Access

The semantics operator $\mathfrak{C}_{\overline{\mathbf{M}}}$ is the abstract counterparts of $\mathfrak{C}$. In particular, $\texttt{access}_{\overline{j}}(\overline{m})$ returns, if $\overline{j}$ is valid for $\overline{m}$ (i.e., there exist a segment bounds interval $[\overline{b}_i[?^i], \overline{b}_{i+1}[?^{i+1}])$ in $\overline{m}$ to which $\overline{j}$ belongs), the segment abstract predicate $\overline{p}_i$ if its upper bound is not question marked, or the least upper bound $\sqcup_{\overline{\mathbf{C}}}$ between successive abstract predicates whose upper bounds are question marked; otherwise it returns $\top_{\overline{\mathbf{C}}}$. It is implied that, if $\overline{\mathtt{j}}$ is the upper bound of $\overline{m}.s$ (i.e., $u_{\overline{m}.s}$) then the operator returns a `is_null` segment abstract predicate. Formally,

$$\mathfrak{C}_{\mathbf{M}}[\![\texttt{access}_{\overline{\mathtt{j}}}]\!](\overline{m}) = \begin{cases} \overline{p}_i & \text{if } \exists i : \overline{j} \in [\overline{b}_i[?^i], \overline{b}_{i+1}) \\ \bigsqcup_{\overline{\mathbf{C}}}{}_{h=i}^{k-1} \overline{p}_h & \text{if } \exists i : \overline{j} \in [\overline{b}_i[?^i], \overline{b}_{i+1}?^{i+1}) \wedge \\ & \quad \{\overline{b}_z?^z\}_{z\in[i+1,k)} \text{ where } \overline{b}_k \text{ is not} \\ & \qquad \text{question marked or } \overline{b}_k = u_{\overline{m}} \\ \top_{\overline{\mathbf{C}}} & \text{otherwise} \end{cases}$$

6.2.2 Abstract String Concatenation

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

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

where $\overline{m}_1'$ is given by $emb_{\mathbf{M}}(\overline{m}_1, \overline{n}_1, \overline{e}_1, \overline{m}_2, \overline{n}_2, \overline{e}_2)$ such that,

$\overline{n}_1 = u_{\overline{m}_1.s}$ and $\overline{e}_1 = u_{\overline{m}_1.s} +_{\overline{\mathbf{B}}} (u_{\overline{m}_2.s} -_{\overline{\mathbf{B}}} \ell_{\overline{m}_2.s})$

$\overline{n}_2 = \ell_{\overline{m}_2.s}$ and $\overline{e}_2 = u_{\overline{m}_2.s}$

and `size.condition` is `true` if

$min\ell en_{\overline{m}_1} \geqslant_{\overline{\mathbf{B}}} (\ell en_{\overline{m}_1.s} +_{\overline{\mathbf{B}}} \ell en_{\overline{m}_2.s} +_{\overline{\mathbf{B}}} 1)$

where $\ell en_{\overline{m}_1.s}$ (resp. $\ell en_{\overline{m}_2.s}$) corresponds to the difference between the upper bound and the lower bound of $\overline{m}_1.s$ (resp. $\overline{m}_2.s$).

6.2.3 Abstract String Character

The semantics operator $\mathfrak{M}_{\overline{\mathbf{M}}}$, when applied to $\texttt{strchr}_{\overline{c}}$ $(\overline{m})$, returns a split segmentation abstract predicate $\overline{s}$ with left hand side parameter equal to the suffix segmentation of the input $\overline{m}.s$ from the first segment to which $\overline{c}$ occurs and right hand side equal to the emptyset if $\overline{m}$ approximates character arrays which contain a well-formed string and the abstract character $\overline{c}$ occurs in $\overline{m}.s$. Otherwise, if $\overline{m}$ approximates character arrays which contain a well-defined string of interest and the abstract character $\overline{c}$ does not occur in $\overline{m}.s$, it returns $\perp_{\overline{\mathbf{M}}}$; otherwise it returns $\top_{\overline{\mathbf{M}}}$. Formally,

$$\mathfrak{M}_{\mathbf{M}}[\![\texttt{strchr}_{\overline{c}}]\!](\overline{m}) = \begin{cases} \overline{s} & \text{if } \overline{m}.s \neq \emptyset \wedge \overline{c} \text{ occurs in } \overline{m}.s \\ \perp_{\overline{\mathbf{M}}} & \text{if } \overline{m}.s \neq \emptyset \wedge \overline{c} \text{ does not occur in } \overline{m}.s \\ \top_{\overline{\mathbf{M}}} & \text{otherwise} \end{cases}$$

6.2.4 Abstract String Compare

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

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

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

In particular, $\overline{n}$ is obtained as defined in Algorithm 2.

6.2.5 Abstract String Copy

The semantics $\mathfrak{M}_{\overline{\mathbf{M}}}$, when applied to $\texttt{strcpy}(\overline{m}_1, \overline{m}_2)$, behaves similarly to the abstract string concatenation operator above. Formally,

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

Algorithm 2 Computing $\overline{n}$

1: $\overline{\mathtt{n}}' = \bot_{\overline{\mathbf{C}}}$
2: $\overline{\mathtt{j}}_1 = \ell_{\overline{m}_1.s}$
3: $\overline{\mathtt{j}}_2 = \ell_{\overline{m}_2.s}$
4: **for** $\overline{\mathtt{j}}_1 \in [\ell_{\overline{m}_1.s}, u_{\overline{m}_1.s}] \wedge \overline{\mathtt{j}}_2 \in [\ell_{\overline{m}_2.s}, u_{\overline{m}_2.s}]$ **do**
5: $\quad\overline{\mathtt{n}} = (\mathtt{access}_{\overline{\mathtt{j}}_1}(\overline{m}_1) -_{\overline{\mathbf{C}}} \mathtt{access}_{\overline{\mathtt{j}}_2}(\overline{m}_2)) \sqcup_{\overline{\mathbf{C}}} \overline{\mathtt{n}}'$
6: $\quad$**if** $\overline{\mathtt{n}} \neq 0 \wedge \overline{\mathtt{j}}_1 \in [\overline{b}^1_i[?^i], \overline{b}^1_{i+1}) \wedge \overline{\mathtt{j}}_2 \in [\overline{b}^2_i[?^i], \overline{b}^2_{i+1})$ **then**
7: $\quad\quad$return $\overline{\mathtt{n}}$
8: $\quad$**if** $\overline{\mathtt{n}} = 0 \wedge \overline{\mathtt{j}}_1 \in [\overline{b}^1_i[?^i], \overline{b}^1_{i+1}) \wedge \overline{\mathtt{j}}_2 \in [\overline{b}^2_i[?^i], \overline{b}^2_{i+1})$ **then**
9: $\quad\quad\overline{\mathtt{j}}_1 = \overline{\mathtt{j}}_1 +_{\overline{\mathbf{B}}} 1$
10: $\quad\quad\overline{\mathtt{j}}_2 = \overline{\mathtt{j}}_2 +_{\overline{\mathbf{B}}} 1$
11: $\quad$**if** $\overline{\mathtt{j}}_1 \in [\overline{b}^1_i[?^i], \overline{b}^1_{i+1}?^{i+1}) \vee \overline{\mathtt{j}}_2 \in [\overline{b}^2_i[?^i], \overline{b}^2_{i+1}?^{i+1})$ **then**
12: $\quad\quad\overline{\mathtt{j}}_1 = \overline{\mathtt{j}}_1 +_{\overline{\mathbf{B}}} 1$
13: $\quad\quad\overline{\mathtt{j}}_2 = \overline{\mathtt{j}}_2 +_{\overline{\mathbf{B}}} 1$
14: $\quad\quad\overline{\mathtt{n}}' = \overline{\mathtt{n}}$
15: return $\overline{\mathtt{n}}$

where $\overline{m}_1'$ is given by $emb_{\mathbf{M}}(\overline{m}_1, \overline{n}_1, \overline{e}_1, \overline{m}_2, \overline{n}_2, \overline{e}_2)$ such that,

$\overline{n}_1 = \ell_{\overline{m}_1.s}$ and $\overline{e}_1 = \ell_{\overline{m}_1.s} +_{\overline{\mathbf{B}}} (u_{\overline{m}_2.s} -_{\overline{\mathbf{B}}} \ell_{\overline{m}_2.s})$

$\overline{n}_2 = \ell_{\overline{m}_2.s}$ and $\overline{e}_2 = u_{\overline{m}_2.s}$

and `size.condition` is `true` if

$min\ell en_{\overline{m}_1} \geqslant_{\overline{\mathbf{B}}} \ell en_{\overline{m}_2.s} +_{\overline{\mathbf{B}}} 1$

6.2.6 Abstract String Length

The semantics $\mathfrak{Z}_{\overline{\mathbf{M}}}$ is the abstract counterpart of $\mathfrak{Z}$. In particular, `strlen` returns, a value $\overline{n}$ if $\overline{m}$ approximates character arrays which contain a well-formed string; otherwise it returns $\top_{\overline{\mathbf{B}}}$. Formally,

$$\mathfrak{Z}_{\mathbf{M}}[\![\mathtt{strlen}]\!](\overline{m}) = \begin{cases} \overline{n} & \text{if } \overline{m}.s \neq \emptyset \\ \top_{\overline{\mathbf{B}}} & \text{otherwise.} \end{cases}$$

In particular, $\overline{n}$ is obtained as follows:

1) $\overline{b} -_{\overline{\mathbf{B}}} \overline{b}$ if $\overline{m}.s = \overline{b}_\llcorner$
2) $\bigsqcup_{\overline{\mathbf{B}}}\{\sqcup_{\overline{\mathbf{B}}}[\overline{b}_i[?^i], \overline{b}_{i+1}[?^{i+1}]) - \ell_{\overline{m}.s} \mid \mathtt{is_null}(\overline{p}_i) = \mathtt{maybe}\} \sqcup_{\overline{\mathbf{B}}} u_{\overline{m}.s} -_{\overline{\mathbf{B}}} \ell_{\overline{m}.s}$ if $\overline{m}.s \neq \overline{b}_\llcorner \neq \emptyset$

6.2.7 Abstract Array Update

The semantics $\mathfrak{M}_{\overline{\mathbf{M}}}$, when applied to $\mathtt{update}_{\overline{j},\overline{c}}(\overline{m})$, returns $\overline{m}'$ that is $\overline{m}$ where the segment abstract predicate occurring in the segment to which $\overline{j}$ belongs has been substituted with $\overline{c}$ if $\overline{j}$ is valid for $\overline{m}$; otherwise it returns $\top_{\overline{\mathbf{M}}}$.

Formally,

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

In particular, $\overline{m}'$ is obtained as follows:

1) $\overline{m}[\overline{p}_i/\overline{c}]$ if $\exists i : \overline{j} \in [\overline{b}_i[?^i], \overline{b}_{i+1}) \wedge \overline{b}_{i-1} -_{\overline{\mathbf{B}}} \overline{b}_i > 1$
2) $\overline{m}[\overline{p}_i/(\overline{c} \sqcup_{\overline{\mathbf{C}}} \overline{p}_i)]$ if $\exists i : \overline{j} \in [\overline{b}_i[?^i], \overline{b}_{i+1}) \wedge \overline{b}_{i-1} -_{\overline{\mathbf{B}}} \overline{b}_i > 1$
3) $\overline{m}[\overline{p}_i/\overline{c} \sqcup_{\overline{\mathbf{C}}} \overline{p}_i,\ \overline{p}_{i+1}/\overline{c} \sqcup_{\overline{\mathbf{C}}} \overline{p}_{i+1}, ..., \overline{p}_k/\overline{c} \sqcup_{\overline{\mathbf{C}}} \overline{p}_k]$ if $\exists i : \overline{j} \in [\overline{b}_i[?^i], \overline{b}_{i+1}?^{i+1}) \wedge \{\overline{b}_z?^z\}_{z\in[i+1,k)}$ where $\overline{b}_k$ is not question marked or $\overline{b}_k = u_{\overline{m}}$

Notice that we could have also defined the $\mathtt{update}_{\overline{\mathtt{j}},\overline{\mathtt{c}}}$ by splitting the segment (when needed) where substitution applies.

6.3 Soundness

Theorem 2 *$\mathfrak{C}_{\overline{\mathbf{M}}}$, $\mathfrak{M}_{\overline{\mathbf{M}}}$, $\mathfrak{P}_{\overline{\mathbf{M}}}$ and $\mathfrak{Z}_{\overline{\mathbf{M}}}$ are sound over-approximations of $\mathfrak{C}$, $\mathfrak{M}$, $\mathfrak{P}$ and $\mathfrak{Z}$ respectively. Formally,*

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

Proof We prove the soundness separately for each operator.

- Consider the unary operator $\mathtt{access}_{\overline{j}}$ and let $\overline{m}$ be a split segmentation abstract predicate. We have to prove that

  $$\gamma_{\overline{\mathbf{C}}}(\mathfrak{C}_{\overline{\mathbf{M}}}[\![\mathtt{access}_{\overline{j}}]\!](\overline{m})) \supseteq \{\mathfrak{C}[\![\mathtt{access}_j]\!](m) : m \in \gamma_{\overline{\mathbf{M}}}(\overline{m})\}$$

  $\mathtt{access}_j$ of m returns the character array value c that occurs at position j, if j is a valid index for m, $\top_{\mathsf{C}}$ otherwise, by definition of $\mathfrak{C}$. Then c belongs to $\gamma_{\overline{\mathbf{C}}}(\mathfrak{C}_{\overline{\mathbf{M}}}[\![\mathtt{access}_{\overline{j}}]\!](\overline{m}))$ because $\mathtt{access}_{\overline{j}}$ of $\overline{m}$, by definition of $\mathfrak{C}_{\overline{\mathbf{M}}}$, is equal to:

  i) the segment abstract predicate $\overline{p}_i$ if $\overline{j}$ is valid for $\overline{m}$ and $\overline{j} \in [\overline{b}_i[?], \overline{b}_{i+1})$
  ii) the least upper bound between subsequent segment abstract predicate whose upper bounds are all question marked up to the first segment upper bound which is not followed by a question mark or which corresponds to the upper bound of the segmentation $u_{\overline{m}}$ if $\overline{j}$ is valid for $\overline{m}$, $\overline{j} \in [\overline{b}_i[?], \overline{b}_{i+1}?^{i+1})$

iii) $\top_{\overline{\mathbf{C}}}$ otherwise.

Notice that $\alpha_{\overline{\mathbf{B}}}(j) = \overline{j}$.

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

$$\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{strcat}]\!](\overline{m}_1, \overline{m}_2)) \supseteq \{\mathfrak{M}[\![\texttt{strcat}]\!](m_1, m_2) : m_1 \in \gamma_{\overline{\mathbf{M}}}(\overline{m}_1) \wedge m_2 \in \gamma_{\overline{\mathbf{M}}}(\overline{m}_2)\}$$

`strcat` of m_1 and m_2 returns $m_1^{'}$ where the first null-terminating memory block of m_2 (including the null terminator), i.e., its string of interest, is embedded into m_1 starting from the index to which occurs the first null character in m_1 if both m_1 and m_2 contain a well-formed string and the size condition on the destination character array value is fulfilled, $\top_{\mathbf{M}}$ otherwise, by definition of $\mathfrak{M}$. Then, $m_1^{'}$ belongs to $\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{strcat}]\!](\overline{m}_1, \overline{m}_2))$ because `strcat` of $\overline{m}_1$ and $\overline{m}_2$, by definition of $\mathfrak{M}_{\overline{\mathbf{M}}}$, is equal to:

  i) $\overline{m}_1^{'}$ that is $\overline{m}_1$ into which $\overline{m}_2.s$ has been embedded starting from $u_{\overline{m}_1.s}$ if both $\overline{m}_1$ and $\overline{m}_2$ approximate character arrays which contain a well-formed string and the size condition on the destination segmentation abstract predicate is fulfilled
  ii) $\top_{\mathbf{M}}$ otherwise.

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

$$\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{strchr}_{\overline{c}}]\!](\overline{m})) \supseteq \{\mathfrak{M}[\![\texttt{strchr}_c]\!](m) : m \in \gamma_{\overline{\mathbf{M}}}(\overline{m})\}$$

$\texttt{strchr}_c$ of m returns s that corresponds to the suffix of the string of interest of m starting from the index to which appears the first occurrence of c if m contains a well-formed string and c occurs in $\overline{m}$, the emptyset (i.e., $\bot_{\mathbf{M}}$) if m contains a well-formed string and c does not occur in m, $\top_{\mathbf{M}}$ otherwise, by definition of $\mathfrak{M}$. Then s belongs to $\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{strchr}_{\overline{c}}]\!](\overline{m}))$ because $\texttt{strchr}_{\overline{c}}$ of $\overline{m}$, by definition of $\mathfrak{M}_{\overline{\mathbf{M}}}$, is equal to:

  i) $\overline{s}$ that is the split segmentation abstract predicate having $\overline{s}.s$ equal to the sub-segmentation of $\overline{m}.s$ starting from the first segment to which $\overline{c}$ occurs and $\overline{s}.ns$ equal to the empty set if $\overline{m}$ approximates character arrays which contain a well-formed string and $\overline{c}$ appears in $\overline{m}$
  ii) $\bot_{\overline{\mathbf{M}}}$ if $\overline{m}$ approximates character arrays which contain a well-formed string and $\overline{c}$ does not appear in $\overline{m}$
  iii) $\top_{\overline{\mathbf{M}}}$ otherwise.

  Notice that $\alpha_{\overline{\mathbf{C}}}(c) = \overline{c}$

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

$$\gamma_{\overline{\mathbf{C}}}(\mathfrak{P}_{\overline{\mathbf{M}}}[\![\texttt{strcmp}]\!](\overline{m}_1, \overline{m}_2)) \supseteq \{\mathfrak{P}[\![\texttt{strcmp}]\!](m_1, m_2) : m_1 \in \gamma_{\overline{\mathbf{M}}}(\overline{m}_1) \ \wedge \ m_2 \in \gamma_{\overline{\mathbf{M}}}(\overline{m}_2)\}.$$

`strcmp` of m_1 and m_2 returns an integer value n, resulting from the difference between corresponding character array elements, denoting the lexicographic order between the strings of interest of m_1 and m_2 if both contain a well-formed string, $\top_{\mathsf{z}}$ otherwise, by definition of $\mathfrak{P}$. Then n belongs to $\gamma_{\overline{\mathbf{C}}}(\mathfrak{P}_{\overline{\mathbf{M}}}[\![\texttt{strcmp}]\!](\overline{m}_1, \overline{m}_2))$ because `strcmp` of $\overline{m}_1$ and $\overline{m}_2$, by definition of $\mathfrak{P}_{\overline{\mathbf{M}}}$, is equal to:

  i) $\overline{n}$ that is the difference between corresponding segment abstract predicates, denoting the lexicographic order between $\overline{m}_1.s$ and $\overline{m}_2.s$ if both approximate character arrays which contain a well-formed string where $\top_{\mathsf{c}}$ does not occur
  ii) $\top_{\overline{\mathbf{C}}}$ otherwise.

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

$$\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{strcpy}]\!](\overline{m}_1, \overline{m}_2)) \supseteq \{\mathfrak{M}[\![\texttt{strcpy}]\!](m_1, m_2) : m_1 \in \gamma_{\overline{\mathbf{M}}}(\overline{m}_1) \wedge m_2 \in \gamma_{\overline{\mathbf{M}}}(\overline{m}_2)\}$$

`strcpy` of m_1 and m_2 returns $m_1^{'}$ where the first null-terminating memory block of m_2 (including the null terminator), i.e., its string of interest, is embedded into m_1 starting from the beginning of m_1 if both m_1 and m_2 contain a well-formed string and the size condition on the destination character array value is fulfilled, $\top_{\mathbf{M}}$ otherwise, by definition of $\mathfrak{M}$. Then, $m_1^{'}$ belongs to $\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{strcpy}]\!](\overline{m}_1, \overline{m}_2))$ because `strcpy` of $\overline{m}_1$ and $\overline{m}_2$, by definition of $\mathfrak{M}_{\overline{\mathbf{M}}}$, is equal to:

  i) $\overline{m}_1^{'}$ that is $\overline{m}_1$ into which $\overline{m}_2.s$ has been embedded starting from $\ell_{\overline{m}_1.s}$ if both $\overline{m}_1$ and $\overline{m}_2$ approximate character arrays which contain a well-formed string and the size condition on the destination segmentation abstract predicate is fulfilled
  ii) $\top_{\overline{\mathbf{M}}}$ otherwise.

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

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

`strlen` of m returns an integer value n which denotes the length of the sequence of character before the first null one in m if m contains a well-formed string,

$\top_{\mathrm{Z}}$ otherwise, by definition of $\mathfrak{Z}$. Then n belongs to $\gamma_{\overline{\mathbf{B}}}(\mathfrak{Z}_{\overline{\mathbf{M}}}[\![\texttt{strlen}]\!](\overline{m}))$ because `strlen` of $(\overline{m})$, by definition of $\mathfrak{Z}_{\overline{\mathbf{M}}}$ is equal to:

i) the difference between the lower bound of $\overline{m}.s$ and itself if $\overline{m}$ approximates character arrays which contain a `null` string of interest
ii) the least upper bound between successive segment bounds into which a `maybe` null abstract predicate occurs minus the lower bound of $\overline{m}.s$ and the difference between the upper bound of $\overline{m}.s$ and its lower bound if $\overline{m}$ approximates character arrays which contain a well-defined string of interest
iii) $\top_{\overline{\mathbf{B}}}$ otherwise.

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

$$\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{update}_{\overline{j},\overline{c}}]\!](\overline{m})) \supseteq \{\mathfrak{M}[\![\texttt{update}_{j,c}]\!](m) : m \in \gamma_{\overline{\mathbf{M}}}(\overline{m})\}$$

$\texttt{update}_{j,c}$ of m returns m' that is m where the character at position j has been substituted by the character$c$ if j is a valid index for m, $\top_{\mathbf{M}}$ otherwise, by definition of $\mathfrak{M}$. Then m' belongs to $\gamma_{\overline{\mathbf{M}}}(\mathfrak{M}_{\overline{\mathbf{M}}}[\![\texttt{update}_{\overline{j},\overline{c}}]\!](\overline{m}))$ because $\texttt{update}_{\overline{j},\overline{c}}$ of $\overline{m}$, by definition of $\mathfrak{M}_{\overline{\mathbf{M}}}$ is equal to:

i) $\overline{m}'$ that is $\overline{m}$ where,
  - the segment abstract predicate $\overline{p}_i$ has been substituted with $\overline{c}$ if $\overline{j}$ is valid for $\overline{m}$, $\overline{j} \in [\overline{b}_i[?^i], \overline{b}_{i+1})$ and $\overline{b}_{i+1} -_{\overline{\mathbf{B}}} \overline{b}_i = 1$
  - $\overline{p}_i$ has been joined with $\overline{c}$ if $\overline{j}$ is valid for $\overline{m}$, $\overline{j} \in [\overline{b}_i[?^i], \overline{b}_{i+1})$ and $\overline{b}_{i+1} -_{\overline{\mathbf{B}}} \overline{b}_i > 1$
  - $\overline{p}_i$ and the following segment abstract predicates have been joined with $\overline{c}$ if $\overline{j}$ is valid for $\overline{m}$, $\overline{j} \in [\overline{b}_i[?^i], \overline{b}_{i+1}?^{i+1})$ and $\{\overline{b}_z?^z\}_{z\in[i+1,k)}$ where $\overline{b}_k$ is not question marked or $\overline{b}_k = u_{\overline{m}}$

ii) $\top_{\overline{\mathbf{M}}}$ otherwise.

Notice that $\alpha_{\overline{\mathbf{B}}}(j) = \overline{j}$ and $\alpha_{\overline{\mathbf{C}}}(c) = \overline{c}$.

7 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 needs 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, instead of 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.

Since in this paper, we focus on string abstraction, we would like to be able to perform the remainder of the program (i.e., the portions that do not work with strings) concretely. In fact, we only want to abstract some of the strings and string operations in the program, since the domain at hand is an approximation: in cases, where the program works with strings that exhibit minimal variation, e.g., string literals, using the M-String representation would not offer any benefit, and could actually hurt performance or introduce spurious counterexamples.

These considerations lead us to conclude that it would be beneficial to re-use, or rather re-purpose, existing tools which work with explicit programs to implement abstract interpretation in a modular fashion. A design in this style (compilation-based abstract interpretation) was proposed and implemented in [22].

However, as presented, the approach was limited to abstracting scalar values. In this paper, we extend this approach to work with strings and other domains that represent more complex objects.

In the remainder of this section, we will first summarize the general approach to abstraction as a program transformation. In Section 7.3, we explore the implications of aggregate (as opposed to scalar) domains within this framework. Sections 7.4 and 7.5 then go on to discuss the semantic (run-time) aspects of the abstraction and which operations we consider as primitives of the abstraction.

7.1 Compilation-based approach

To perform abstraction, instead of (re-)interpreting instructions abstractly, we transform abstract instructions into equivalent explicit code, which implements the abstract computation. The transformation occurs before model checking (or other dynamical analysis), during the compilation process.

The transformed program can be further analyzed or processed without special knowledge of the abstract domains in use, because those are now encoded directly in the program. Comparison of this compilation-based approach and the approach of more traditional abstract interpreters (an interpretation-based approach) is shown in Figure 1.

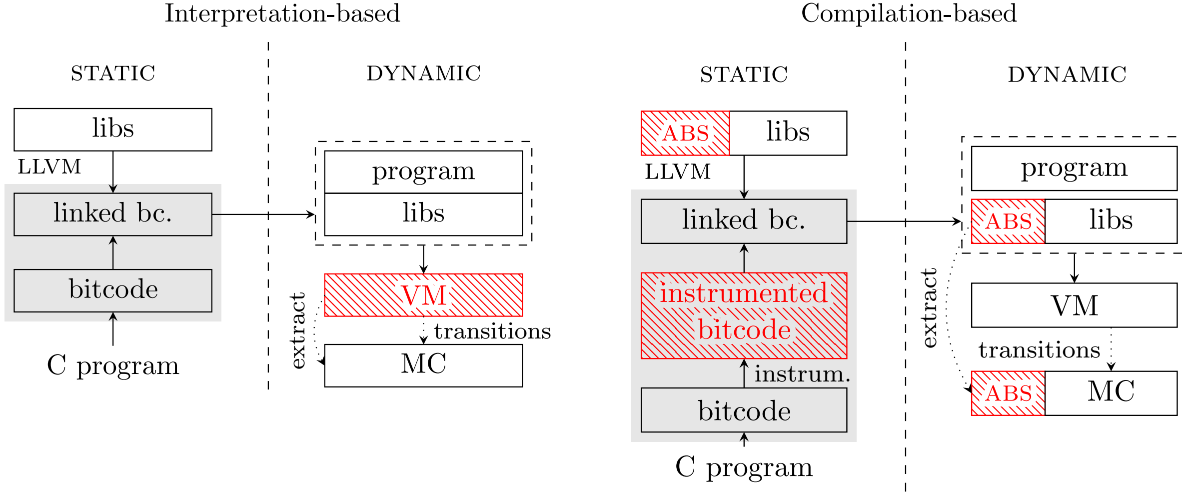

Fig. 1: The figure depicts a comparison of interpretation/compilation-based approaches. In interpretation-based approach, entire abstract interpretation is performed during runtime. A virtual machine (VM) interprets bitcode operations abstractly and maintain an abstract state. Consequently, it generates an abstract state-space for a model-checking algorithm (MC). On the other hand, compilation-based approach instruments abstract operations into the compiled program and provides their implementation as a library. A virtual machine then executes the instrumented program as regular bitcode.

In compilation-based approach, we consider two levels of abstraction:

1. *static*, concerning the syntax and the type system,
2. *dynamic*, or semantic, concerning execution and values.

LART performs syntactic (*static*) abstraction on LLVM bitcode [21]. The goal of syntactic abstraction is to replace some of the LLVM instructions in the program with their abstract counterparts. We illustrate syntactic abstraction in Figure 2.

Concrete program:

```
a:str ← abstract()
b:str ← string()
c:str ← strcat(a, b)
l:int ← strlen(c)
```

Transformed program:

```
a:a_str ← a_string()
b:str ← string()
c:a_str ← a_strcat(a, lift(b))
l:a_int ← a_strlen(c)
```

Fig. 2: Syntactic abstraction.

7.2 Syntactic abstraction

During syntactic abstraction, LART performs a data flow analysis, starting from annotated abstract values (`abstract`) as the roots. The result of this analysis is the set of all operations that may come into contact with an abstract value. These are then substituted by their abstract counterparts (`a_strcat`, `a_strlen`). Abstract counterparts implement manipulations with M-String values described in Section 6.

An abstract instruction takes abstract values as its inputs and produces an abstract value as its result. The specific meaning of those abstract instructions and abstract values then defines the semantic abstraction.

To formulate syntactic abstraction unambiguously, we take advantage of the static type system of LLVM. By assigning types to program variables, we can maintain a precise boundary between concrete and abstract values in our program.

We recognize a set of *concrete scalar types* S. We give a map Γ that inductively defines finite (non-recursive) algebraic types over the set of given scalars. To be specific, the set of all types $\Gamma(T)$ derived from a set of scalars $T \subseteq \mathcal{V}$ is defined as follows:

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

In syntactic abstraction, we extend the concrete set of types by abstract types. From these, we generate admissible types using Γ. Depending on the level of abstraction, we define a different set of basic abstract types. In the case of scalar abstraction, a set of basic abstract types contains abstract scalar types $\overline{S}$. Correspondence between abstract and concrete scalars is given by a bijective map $\Lambda : S \to \overline{S}$. Finally, each value, which exists in the abstracted program, has an assigned type of $\Gamma(S \cup \overline{S})$. In particular, this means that the abstraction works with *mixed types* – products and unions with both concrete and abstract fields. Likewise, it is possible to form pointers to both abstract values and to mixed aggregates.

7.3 Aggregate domains

Scalars in a program are simple values which cannot be further decomposed into meaningful constituent parts. A typical example would be an integer, or a pointer. However, programs typically also work with more complex data, that we can think of as compositions – aggregates – of multiple scalar values. Depending on the nature of such aggregates, we can classify them as arrays, which contain a variable number of items which all belong to a single type, records (structures), which contain a fixed number of items in a fixed layout, but each of these can be of a different type. The items in such aggregates can be (and often are) scalars, but more complicated aggregates are also possible: arrays of records, records which in turn contain other records, and so on.

In contrast to scalar domains, which deal with scalar values, an *aggregate domain* represents composite data, in the spirit of the above definition. An *abstract* aggregate domain approximates (concrete) aggregate values by keeping track of certain properties of the aggregate, for instance the length of an array, or a set of scalars that appear in the array. In the case of M-String, the information it tracks is a segmentation, where segments are represented using their bounds and a single value abstracting their content.

Aggregate domains could be equipped with quite arbitrary operations, though there are two that stand out, because they are in some sense universal, and those are byte-wise access and modification (update) of the content of the aggregate. The universality of those operations stems from the fact that in a low-level representation of a program, all operations with aggregate values take this form. In LLVM, it is possible (though not guaranteed), that access to the aggregate is encoded at a slightly higher level: as extraction and modification of entire scalars (as opposed to individual bytes). For M-String, though, this distinction is not important because the scalars stored in `C` strings are individual bytes. It should be also noted that the `access` and `update` form the interface between scalars and aggregates (even in the case of byte-oriented access, since bytes are also scalars). We refer the reader to the Section 5.3.1 for abstract semantics of `access`, respectively to the Section 5.3.7 for the abstract semantics of `update`.

Therefore, the types of those two operations contain a single aggregate and (at least) a single scalar domain. Some (or all) of those domains may be abstract domains.

Syntactic abstraction has to handle aggregate domains differently from scalar domains. In LLVM, aggregate values are usually represented using pointers of a specific (aggregate) type. For this reason, aggregate abstraction starts from the types that represent its objects. In the case of arrays, those are concrete pointers into those arrays: let us call them P^*, where $P \subseteq \Gamma(S)$. We use the set of abstract pointers $\overline{P}^*$ to represent the types of abstract values in an aggregate domain. Thus the set of admissible types in the abstract program is generated by $\Gamma(S \cup \overline{P}^*)$. Like in scalar domains, we define a natural correspondence between pointers to concrete values P^* as a bijective map $\Lambda : P^* \to \overline{P}^*$.

Please note that pointers in general contain two pieces of information: they determine the *object* and an *offset* into that object. In explicit programs, this distinction is not very important, since those two parts are represented uniformly and often cannot be distinguished at all. The distinction, however, becomes important when we deal with abstract aggregate values. In this case, the *object* portion of the pointer is concrete, since it determines a single specific abstract object. However, the *offset* may or may not be concrete – depending on the specific abstract aggregate domain, it may be more advantageous to represent the offset abstractly. In either case, however, all memory access through such a pointer needs to be treated as an abstract `access` or `update` operation.

In LLVM, there are two basic memory access operations – *load* and *store*, which correspond to the `access` and `update` operations. Rather importantly, memory access is always explicit – memory is never directly used in a computation. We use this fact in the design of aggregate abstraction, where we can assume that access to the content of an aggregate will always go through a pointer associated with the abstract object.

7.4 Semantic abstraction

Where syntactic abstraction was concerned with the syntax of operations, their types and the types of values

and variables, semantic abstraction is concerned with the runtime values that appear during the computation performed by a program. While syntactic abstraction introduced the maps Λ and Λ^{-1} to transfer between concrete and abstract *types*, semantic abstraction introduces *lift* and *lower* (cf. Definition 6 and Definition 7): operations (instructions) which convert between concrete and abstract *values*. They represent a realization of the abstraction (α) and concretization (γ) functions.

The *lift* operation, as shown in Figure 2, enables abstraction of a set of concrete values by a single over-approximating abstract value. In comparison to Λ, which was a purely syntactic construct, *lift* and *lower* accomplish actual conversion of values between domains during program runtime. From the point of view of abstration, *lift* represents a realization of the abstraction function of a domain ($\alpha_{\overline{\mathbf{M}}}$ in the case of M-String). Likewise, *lower* is an executable form of the concretization function $\gamma_{\overline{\mathbf{M}}}$. During program execution, lowering an abstract value into multiple concrete values can be seen as non-deterministic branching in the program (and the *lower* operator is indeed based on a non-deterministic choice[4] operator).

While *lift* and *lower* form a boundary between concrete and abstract scalar computation, the `access` and `update` operations of an aggregate domain form a boundary between scalar and aggregate domains. We kindly refer to [22], where a reader may find how LART transforms an abstract program into an executable form.

7.5 Abstract operations

After syntactic abstraction, the program temporarily contains abstract instructions. Abstract instructions take abstract values as operands and give back abstract values as their results. However, after transformation, we require that the resulting program is semantically valid LLVM bitcode. Hence, it is crucial that each abstract instruction can be realized as a suitable sequence of concrete instructions. This makes it possible to obtain an abstract program that does not actually contain any abstract instructions and execute it using standard (concrete, explicit) methods.

In detail, syntactic abstraction replaces concrete instructions with their abstract counterparts: an instruction with type $(t_1, \ldots, t_n) \to t_r$ is substituted by an abstract instruction of type $(\Lambda(t_1), \ldots, \Lambda(t_n)) \to \Lambda(t_r)$. Moreover, *lift* and *lower* are inserted as needed. The implementation is free to decide which instructions to abstract and where to insert value lifting and lowering, so long as it obeys type constraints.

Additionally, in string abstraction, we also want to abstract function calls such as `strcat`, `strcpy` etc. From the perspective of abstraction, we treat these functions as single operations that take abstract values and produce results. Therefore, we can process them in the same way as instructions. For example, by transforming `strcat` of type $(m,\, m) \to m$ we obtain `a_strcat` of type $(\Lambda(m),\, \Lambda(m)) \to \Lambda(m)$ where m is the concrete value of a character array `m`. Afterwards, all abstract operations are realized using concrete subroutines, for details see [22].

We could have also transformed standard library functions (`strcat`, `strcmp`, etc.) instruction by instruction using only abstract access and update of a content, but in this way we would lose a certain degree of precision in the abstraction, the exact amount depending on the operation.

8 Instantiating M-String

M-String, as a content domain, enables a parametrization of string abstraction. To be specific, it supports the parametrization of string segmentation representation in which we can substitute different domains of bounds and characters. As a representation of string values, we can use a scalar domain equipped with the correct operations, and the same holds for bounds of segments as described in Section 6.

A particular M-String instance can be automatically derived from a parametric description, given well-defined abstract domains $\overline{\mathbf{C}}$ for characters and $\overline{\mathbf{B}}$ to represent segment bounds. M-String also requires that both $\overline{\mathbf{C}}$ and $\overline{\mathbf{B}}$ support certain operations that appear in the generic implementation of the abstract operations described in Section 6. These are mainly basic arithmetic and relational operators.

8.1 Symbolic scalar values

In program verification, it is common practice to represent certain values symbolically (for instance inputs from the environment). This type of representation enables a verification procedure to consider all the possible values with a reasonably small overhead. In DIVINE, symbolic computation is implemented using abstraction of the same type as described here: computations on scalar values are lifted into the term domain, which simply keeps track of values using terms (expressions) in form

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

of abstract syntax trees. Those trees contain atoms (unconstrained values) and operators of the bitvector logic. The term domain additionally keeps track of any constraints derived from the control flow of the program (a *path condition*). A more detailed description is presented in [22].

Paired with a constraint solver for the requisite theory,[5] the term domain coincides with symbolic computation. The solver makes it possible to detect computations that have reached the bottom of the term domain (those are the infeasible paths through the program) and also to check for equality or subsumption of program states. With those provisions, the bitvector theory is completely precise (i.e., it is not an approximation, but rather models the program state faithfully).

8.2 Concrete characters, symbolic bounds

For evaluation purposes, we have instantiated the M-String domain by setting $\overline{\mathbf{C}}$, the domain of the individual characters, to be the concrete domain (i.e., characters are represented by themselves) and $\overline{\mathbf{B}}$, the domain of segment bounds, to be symbolic 64b integers. The main motivation for this instantiation is a balance between simplicity on one hand (both the domains we used for parameters were already available in the tools we used) and the ability to describe strings with undetermined length and structure.

At the implementation level (as explained in more detail in the following section), the domain continues to be parametric: the specific domains we picked could be easily swapped for other domains (an immediate candidate would be using both symbolic characters and symbolic bounds). Compared to the theoretical description of M-String, the implementation uses a slightly simplified representation using a pair of arrays (cf. Figure 3), where the specific type of characters and bounds is given by the parameter domains $\overline{\mathbf{C}}$ and $\overline{\mathbf{B}}$ respectively.

M-String, when instantiated like this, is particularly suitable for representing strings with runs of a single character of variable length, i.e., the strings of the form $a^k b^l c^m$... where relationships between $k, l, m, ...$ can be specified using standard arithmetic and relational operators and each of a, b, c is a specific letter. This in turn allows M-String to be used for checking program behaviour on broad classes of input strings described this way. A more detailed account of this approach can be found in Section 9.

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

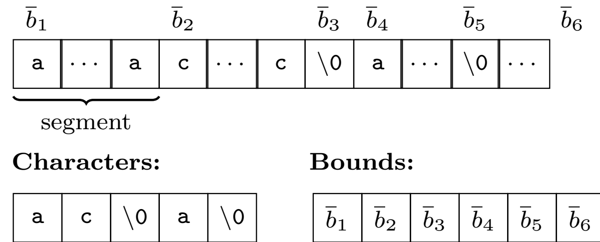

Fig. 3: M-String value with symbolic bounds, where string of interest is from $\bar{b}_1$ to $\bar{b}_3$.

8.3 Symbolic characters, symbolic bounds

In the benchmarks where the computation with M-String values encountered abstract scalars from the program represented in the term domain, we have instantiated the M-String domain by setting the domain of characters $\overline{\mathbf{C}}$ to be the term domain, which keeps track of symbolic 8b integers (characters in C language). In this way, we do not need to lower abstract characters to the concrete domain used in the previous instantiation. However, we pay the price for more expensive computation with symbolic characters.

8.4 Implementation

We have implemented the abstract semantics of operations in the M-String domain as a `C++` library, in a form that allows programs to be automatically lifted into this domain by LART and later model-checked with 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 8.2 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 those 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.

The abstract domain is equipped with a set of essential operations, which appear in all programs that work with strings: these are `lift`, `update` and `access`. All other operations which involve strings can be, in principle, derived automatically using the same procedure that is applied to user programs. However, abstracting only `access` and `update` causes either a loss of precision or a blowup in complexity. For this reason, we also

include hand-crafted implementations of the following abstract operations: `strcmp`, `strcpy`, `strcat`, `strchr`, and `strlen`. These are all based on the abstract semantics of the respective operations as described in Section 6.

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 7.3), and the representation of the *offset* component must be compatible with the bound domain $\overline{\mathbf{B}}$.

A more complete description of the implementation of LART and DIVINE, their source code, and technical details of the M-String domain can be found online.[6]

9 Experimental evaluation

For evaluation purposes, we have picked four scenarios. In first of those, we show that the provided implementation of basic string functions is more efficient than lifting them automatically based on the `access` and `update` operations. In the second scenario, we analyse various implementations of the same string functions by lifting them automatically and checking that their outputs match the ones we expect based on the concrete semantics of those operations – in this case, the inputs are provided in the form of specific abstract (M-String) values. 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. In the last scenario, we have picked a few real-world programs to demonstrate that M-String can be successfully used in analysis of moderately complex `C` code. To this end, we have chosen two context-free grammars and used them to generate `C` parsers using the `bison` and `flex` tools, again providing abstract strings as inputs to the generated parsers. All experiments were performed with an identical set of resource constraints: 1 hour of CPU time, 80 GB of RAM and 4 CPU cores.[7]

[6] https://divine.fi.muni.cz/2020/mstring

[7] The processor used to run the benchmarks was AMD EPYC 7371 clocked at 2.60GHz.

Abstract operations: The first set of benchmarks covers resource usage measurements of M-String operations. Results are presented in Table 2. We run each operation separately on two different types of M-String inputs parametrized by length l:

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

We have measured how much time we spend in the abstract operations which are part of the M-String domain (see Table 2) and compare them to the same programs, but with the functions abstracted automatically, using only the M-String definitions of `access` and `update` (see Table 3).

One of the results is that the size of the state space does not depend on the length of the string when using the operations from M-String. This is because the number of segments does not change and the operations perform the same amount of work. In comparison, analysis of automatically lifted implementations of the same functions[8] does not terminate in a 1-hour time limit for strings of length 64 and more. This is caused by the fact that the concrete implementations need to iterate over each character individually causing exponential blowup of possible character combinations, while the M-String implementation directly works with segments.

Sequence

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

Table 3: Benchmark of of standard library functions abstracted using only the M-String definitions of *access* and *update*v operations. Verification for *Word* strings times out in most of the instances.

C standard library: The second scenario deals with correctness of various concrete implementations of the set of standard library functions. Namely, we used three

[8] The implementations were taken from `PDClib`, a public-domain `libc` implementation.

	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 2: Benchmarks of abstract operations were evaluated on two types of M-Strings (*Word* and *Sequence*) – see Section 9 for description. The table depicts the number of states in the state space of the verified program, verification time in seconds for the different length of inputs and an average time of a transformation (LART).

sources: PDClib, musl-libc and μCLibc. Results for these libraries are very similar, hence we only present results for PDClib library – data for the remaining two are part of the supplementary material.

Sequence

	4		8		16	
	time(s)	states	time(s)	states	time(s)	states
strcmp	2.17	204	5.09	376	16.5	720
strcpy	0.83	183	2.49	347	9.14	675
strcat	8.56	751	113	2535	1940	9463
strchr	0.3	17	0.3	17	0.4	17
strlen	0.15	34	0.28	54	0.65	94

Word

	4		8		16	
	time(s)	states	time(s)	states	time(s)	states
strcmp	14.3	1005	105	2989	1350	9741
strcpy	5.15	515	57.4	1823	912	6935
strcat	468	5748	T	–	T	–
strchr	0.08	22	0.08	22	0.08	22
strlen	0.66	91	4.13	259	68.8	883

Table 4: Verification results of functions from PDCLib with timeout of 1 hour. For each type of input M-String of a given length, we present duration of verification and the size of the state-space.

In these benchmarks, we compare the results of the abstract implementation with the result of the automatically abstracted (originally concrete) implementation of each function and check that they give identical results.

Results show that analysis of strings with multiple-segments is more expensive. This is because segments might disappear when they have zero length and two segments are merged into one: the SMT queries arising from these events are hard to solve, because of the large number of possible overlaps in the segment bounds.

The library implementations access and update the string one character at a time, resulting in large SMT formulas – this causes the blowup in analysis time and hence timeouts with longer strings.

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

	correct		error found		
	benchs	time(s)	benchs	time(s)	timeout
apache	0	–	26	384.26	24
openser	43	234.13	45	105.93	6
wu-ftpd	8	35.78	14	2461.27	19
libgd	4	9.01	4	1.85	0
madwifi	5	0.51	5	0.55	0
gxine	1	0.53	1	0.25	0

Table 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 and accumulated time for each category.

Bison grammar: In the last scenario, we analyse two parsers generated by bison. First is a parser for numerical expressions which consist of binary operators and numbers (see Table 6). The second example is a parser for a simple programming language.

Like with the previous scenarios, inputs which contain long sequences of the same character perform the best, especially when contrasted with a similar task performed on an input with alternating digits.

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	–

BP Grammar

	10		100		1000	
	time(s)	states	time(s)	states	time(s)	states
value	6.58	38	90.4	488	1100	4988
loop	1.53	23	4.88	23	33.3	23
wrong	7.34	82	67.7	892	311	8992

Table 6: Evaluation on parsers of mathematical expressions (ME) and simple programs (BP). Inputs for ME were of 3 forms: `addition` is a string with two numbers with `+` between them, `ones` is a sequence of ones, and `alternation` represent a number with multiple digits. Inputs for BP were of the form: `value` constructs a constant, while `loop` is a program with a single bounded loop and `wrong` is a program with a syntax error.

10 Conclusion

We have presented a segmentation-based abstract domain for approximating `C` strings. The main novelty of the domain lies in its focus on string buffers, which consist of two parts: the string of interest itself, and a tail of allocated and possibly initialized but unused memory. This paradigm allows for precise modeling of string functions from the standard `C` library, including their often fragile handling of terminating zeroes and buffer bounds. In principle, this allows the M-String domain to identify string manipulation errors with security consequences, such as buffer overflows.

In addition to presenting the domain theoretically, we have implemented the abstract semantics in executable form (as `C++` code) and combined them with a tool that automatically lifts string-manipulating code in existing `C` programs to the M-String domain. Since M-String is a parametric domain – the domains for both segment content and segment bounds can be freely chosen – we have instantiated M-String (for evaluation purposes) with both concrete and symbolic characters and with symbolic (bitvector) bounds.

11 Declarations

11.1 Funding

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

11.2 Conflict of interest

The authors declare that they have no conflict of interest.

11.3 Availability of data and material

Supplementary materials and benchmarks are publicly available at https://divine.fi.muni.cz/2020/mstring.

11.4 Code availability

The supplementary materials contain binary distribution and sources of extended model-checker DIVINE, that is an open source software distributed under the ISC license.

References

1. Amadini R, Jordan A, Gange G, Gauthier F, Schachte P, Søndergaard H, Stuckey PJ, Zhang C (2017) Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I, pp 41–57, DOI 10.1007/978-3-662-54577-5_3
2. Baranová Z, Barnat J, Kejstová K, Kucera T, Lauko H, Mrázek J, Rockai P, Still V (2017) Model Checking of C and C++ with DIVINE 4. In: Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings, pp 201–207, DOI 10.1007/978-3-319-68167-2_14
3. Bultan T, Yu F, Alkhalaf M, Aydin A (2017) String Analysis for Software Verification and Security. Springer
4. Cass S (2019) The Top Programming Languages 2019. IEEE Spectrum Magazine. Available: https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019. Accessed 10 February 2020.
5. clc-wiki (2019) C Standard Library: string.h. Available: https://clc-wiki.net/wiki/C_standard_library:string.h. Accessed 10 February 2020.
6. Cortesi A, Olliaro M (2018) M-String Segmentation: a Refined Abstract Domain for String Analysis in C Programs. In: Theoretical Aspects of Software Engineering - 12th International symposium, TASE 2018, Guangzhou, China, August 29-31, 2018, Proceedings, DOI 10.1109/TASE.2018.00009

7. Cortesi A, Zanioli M (2011) Widening and Narrowing Operators for Abstract Interpretation. Computer Languages, Systems & Structures 37(1):24–42, DOI 10.1016/j.cl.2010.09.001
8. Cortesi A, Lauko H, Olliaro M, Rockai P (2019) String Abstraction for Model Checking of C Programs. In: Model Checking Software - 26th International Symposium, SPIN 2019, Beijing, China, July 15-16, 2019, Proceedings, pp 74–93, DOI 10.1007/978-3-030-30923-7_5
9. Costantini G, Ferrara P, Cortesi A (2015) A Suite of Abstract Domains for Static Analysis of String Values. Softw, Pract Exper 45(2):245–287, DOI 10.1002/spe.2218
10. Cousot P, Cousot R (1977) Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, pp 238–252, DOI 10.1145/512950.512973
11. Cousot P, Cousot R (1979) Systematic Design of Program Analysis Frameworks. In: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979, pp 269–282, DOI 10.1145/567752.567778
12. Cousot P, Cousot R (1992) Abstract Interpretation Frameworks. J Log Comput 2(4):511–547, DOI 10.1093/logcom/2.4.511
13. Cousot P, Cousot R, Logozzo F (2011) A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, pp 105–118, DOI 10.1145/1926385.1926399
14. Dor N, Rodeh M, Sagiv S (2003) CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003, pp 155–167, DOI 10.1145/781131.781149
15. Evans D, Larochelle D (2002) Improving Security Using Extensible Lightweight Static Analysis. IEEE Software 19(1):42–51, DOI 10.1109/52.976940
16. Holzmann GJ (2002) Uno: Static source code checking for userdefined properties. In: In 6th World Conf. on Integrated Design and Process Technology, IDPT '02
17. Jensen SH, Møller A, Thiemann P (2009) Type Analysis for JavaScript. In: Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings, pp 238–255, DOI 10.1007/978-3-642-03237-0_17
18. Jones RWM, Kelly PHJ (1997) Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In: AADEBUG, pp 13–26
19. Journault M, Miné A, Ouadjaout A (2018) Modular Static Analysis of String Manipulations in C Programs. In: Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings, pp 243–262, DOI 10.1007/978-3-319-99725-4_16
20. Kashyap V, Dewey K, Kuefner EA, John, Gibbons K, Sarracino J, Wiedermann B, Hardekopf B (2014) JSAI: a Static Analysis Platform for JavaScript. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, pp 121–132, DOI 10.1145/2635868.2635904
21. Lattner C, Adve V (2004) LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", booktitle = "International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California, DOI 10.1109/CGO.2004.1281665
22. Lauko H, Rockai P, Barnat J (2018) Symbolic Computation via Program Transformation. In: Theoretical Aspects of Computing - ICTAC 2018 - 15th International Colloquium, Stellenbosch, South Africa, October 16-19, 2018, Proceedings, pp 313–332, DOI 10.1007/978-3-030-02508-3_17
23. Madsen M, Andreasen E (2014) String Analysis for Dynamic Field Access. In: Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, pp 197–217, DOI 10.1007/978-3-642-54807-9_12
24. MathWorks (2001) Polyspace. Available: https://www.mathworks.com/products/polyspace.html. Accessed 10 February 2020.
25. Miné A (2006) Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), Ottawa, Ontario, Canada, June 14-16, 2006, pp 54–63, DOI 10.1145/1134650.1134659
26. One A (1996) Smashing The Stack For Fun And Profit. Phrack Magazine
27. OWASP (2019) Static Code Analysis. Available: https://www.owasp.org/index.php/

Static_Code_Analysis#Weaknesses. Accessed 10 February 2020.
28. Park C, Im H, Ryu S (2016) Precise and Scalable Static Analysis of jQuery Using a Regular Expression Domain. In: Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016, pp 25–36, DOI 10.1145/2989225.2989228
29. Seacord RC (2013) Secure Coding in C and C++, 2nd edn. Addison-Wesley Professional
30. Shahriar H, Zulkernine M (2010) Classification of Static Analysis-Based Buffer Overflow Detectors. In: Fourth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2010, Singapore, June 9-11, 2010 - Companion Volume, pp 94–101, DOI 10.1109/SSIRI-C.2010.28
31. Spoto F (2016) The Julia Static Analyzer for Java. In: Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, pp 39–57, DOI 10.1007/978-3-662-53413-7_3
32. Wagner DA, Foster JS, Brewer EA, Aiken A (2000) A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2000, San Diego, California, USA
33. Xie Y, Chou A, Engler DR (2003) ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In: Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003, pp 327–336, DOI 10.1145/940071.940115
34. Xu R, Godefroid P, Majumdar R (2008) Testing for Buffer Overflows with Length Abstraction. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, pp 27–38, DOI 10.1145/1390630.1390636