Ca' Foscari University of Venice

Master's Degree programme
Second Cycle (D.M. 270/2004) in
Computer Science

Final Thesis

M-String Segmentation: a refined abstract domain for static analysis of C programs

Supervisor
Ch. Prof. Agostino Cortesi

Graduand
Martina Olliaro
Matriculation Number 834397

Academic Year
2016/2017

To my mother, a precious woman.

Acknowledgements

Thanks to my supervisor, Agostino Cortesi, for giving me the opportunity of writing this thesis and for following me with resolution and dedication. I feel a deep esteem towards him.

Thanks to my family for giving me the possibility to study, for helping me and for always supporting me during my studies and during the composition of this thesis.

Thanks to my colleagues and to all my beloved ones.

Abstract

The goal of the thesis is to present a refined segmentation abstract domain for the analysis of strings in C programming language. We properly extend the parametric segmentation approach to array representation by P. Cousot to the case of text values. In particular, we capture the so-called *string of interest* of an array of char, and we are able to distinguish well-formed string arrays. A concrete and abstract semantics of the main C header file string.h functions are presented in full detail.

Contents

Chapter 1

Introduction

In this thesis we present a refined segmentation abstract domain for the analysis of strings in C programming language. We properly extend the parametric segmentation approach to array representation by P. Cousot [1] to the case of text values and we provide a sound static analysis of char arrays as renderers of strings.

1.1 Context

In C programming language, strings (a string is a sequence of characters treated as a single unit) are represented as char arrays structured data types in which the last significant element of each string is followed by the terminating null character. It is a programmer responsibility managing a char array structure in a consistent way with the concept of string, for example, ensuring the presence of the terminating null character in it. Programming with C strings is often error prone. The four most common errors are: unbounded string copies, off-by-one errors, null termination errors, and string truncation. There are many standard string handling functions that are highly susceptible to error, like the `strcpy()` and the `strcat()` functions, that are frequent sources of buffer overflows because they do not allow the caller to specify the size of the destination array. However, since C-style strings are arrays of characters, it is possible to perform an insecure string operation even without invoking a function [8]. We aim to detect these strings manipulation errors by taking advantage of the static program analysis, in particular of the abstract interpretation technique.

Static program analysis is the art of reasoning about the behaviour of computer programs, defining their mathematical meaning, checking all their possible executions and providing guarantees about their properties, with the right kind of approximation [10]. Static analysis aims to generate code avoiding redundant and superfluous computations and to validate software, in order to reduce the likelihood of malicious or unintended behaviours, detecting program errors. One of the main approaches to static program analysis is the abstract interpretation technique that, starting from the analysed program concrete semantics definition (the concrete semantics formalises the set of all the program possible executions in all possible execution environments) and an abstract domain (an abstract domain is an abstraction of concrete environments), derives the so-called abstract semantics, which allows to run the program on the abstract domain to compute the property that the abstract domain models. Informally, abstract interpretation deals with expressing the semantics of a program as an approximation of its concrete semantics, allowing the capture of safe (but not complete) information on all the behaviour of the program.

In the context of string static analysis, the *character inclusion* domain, the *prefix and suffix* domains, the *bricks* domain, presented in [11], the *string set* domain, the *constant string* domain, the *prefix-suffix* domain and the *string hash* domain, defined in [12], are abstracts domains for the static analysis of string values, each of which tracks a different kind of information.

A concrete and abstract semantics about a parametric segmentation approach to array representation has been proposed by P. Cousot, in [1]. In particular, Cousot introduced the `FunArray` parametric segmentation abstract domain functor for the fully automatic and scalable analysis of array content properties. The functor enables a natural, painless and efficient lifting of existing abstract domains for scalar variables to the analysis of uniform compound data-structures such as arrays. This analysis aims to automatically and semantically divides arrays into consecutive non-overlapping possibly empty segments. Segments are delimited by sets of bound expressions and abstracted uniformly. All symbolic expressions appearing in a bound set are equal in the concrete. The `FunArray` can be naturally combined via reduced product with any existing analysis for scalar variables. Cousot presented the analysis as a general framework, parametrized by the choice of bound expressions, segment abstractions and the reduction operator. Once the functor has been instantiated with fixed parameters, the analysis is fully automatic. Array element values are related to their indexes, assuming that the concrete value of an array A is a quadruple $a = (\rho, A.low, A.high, A)$ where $\rho \in \mathcal{R}_v$ is a scalar variable environment, $A.low$ and $A.high$ denotes respectively the expressions whose values, evaluated in the environment ρ, yield the integer lower bound and the integer upper bound of the array indexes. Lastly, A is a function that maps every index $i \in [[\![A.low]\!]\rho, [\![A.high]\!]\rho)$ to the pair $A[i] = (i, v)$ of the index i and the corresponding array element value v.

1.2 Problem

The parametric representation of arrays introduced in [1] is well defined for any type of C array, but does not capture the specific features of arrays representing strings. Precisely, is not able to detect in an explicit way, the common strings manipulation errors mentioned above.

1.3 Metodology

Our goal is to capture the so-called "***string of interest***" of an array of char, guaranteeing us to distinguish well-formed strings in a char array from the rest of its content. For this purpose, we have defined the "***split***" of a segmented char array, supported by an *ad hoc* concrete semantics.

Given an array of char, str, and its segmentation, the $split(str)$ is equal to the pair (s_{str}, ns_{str}) such that s_{str} corresponds to the sub-segmentation of the string of interest of str, that is the segmented sequence of characters before the first terminating null character, and ns_{str} corresponds to the sub-segmentation of the char array content which falls outside the string of interest, that is the segmented sequence of characters after the first terminating null character. The split segmentation over-approximates an array of characters and takes into account the needful presence of the terminating null character in it, to establish the existence of the string of interest of the considered char array and to isolate the meaningful information, for further analysis on it. We have presented a refined unidimensional array concrete semantics, suitable to the split array of character segmented representation: the concrete value of an array of char, *str*, is a quintuple $str = (\rho, str.low, str.high, S, \mathsf{T}_{str})$, where the first fourth elements are instantiated to be equal to the Cousot quadruple parameters and the fifth element corresponds to the set of all the indexes i such that str[i] = '\0'. Furthermore we have explicitly assumed that each possible sub-segmentation of a char array has its own concrete value. This means that, given a program managing an array of characters that, for example, performs a string character function on it and print the output, the function result is a portion of the analysed char array, a suffix of its string of interest (if it is defined), represented by a sub-segmentation that preserve its bound expressions. So, after the program point in which the string character function is declared, we keep track of both the managed char array, whose concrete value does not change from the program point in which the function is stated, and the char sub-array string character statement result. Notice that, also the *split* parameters of a char array are sub-segmentations of the whole segmentation of it and as such have their own concrete value.

We have defined in full detail the concrete and abstract semantics of the main C header file string.h functions. In particular, we have capture the behaviour of the `strcpy()`, `strcat()`, `strlen()`, `strchr()`, `strcmp()` functions in the cases in which both the manipulated char arrays were well-formed strings both when the strings common error occurred. We have also studied the behaviour of the modification operation of a char array, constructing a suitable function. We have implemented these functions, in C programming language, so that we could exploit the segmentation of a char array and we tested the effectiveness of our refined char array representation. Moreover, we applied the bioinformatics structures of *suffix array* and *longest common prefix array* [7] in the creation of alternative algorithms to express the `strchr()` and the `strcmp()` functions.

1.4 Results

We designed a segmentation abstract domain for the analysis of string in C programming language and a set of sound operations on it.

We implemented the reviewed string.h functions and collected them in the M-String library, in order to test the correctness of our *ad hoc* char array representation and to capture the string of interest, for further analysis on it.

Finally, some preliminary experimental results are discussed, that show the effectiveness of the proposed domain for string analysis.

1.5 Structure

This thesis is organized as follows:

- Chapter 2 describes the basis of abstract interpretation analysis, with particular attention to the abstract domains for static analysis of string values;
- Chapter 3 presents in full detail the parametric segmentation approach to array representation introduced in [1];
- Chapter 4 introduces our refined segmentation abstract domain for the analysis of strings in C programming language and describes our concrete and abstract semantics of the main C header file string.h functions;
- Chapter 5 presents the M-String library of the reviewed string.h functions defined in Chapter 4;
- Chapter 6 compares our *ad hoc* char array representation to the one introduced in [1], the advantages and disadvantages of our strings array abstract domain and the possible practical applications, by discussing some preliminary experimental results;
- Chapter 7 concludes;
- Appendix A provides the M-String library implementation.

Chapter 2

Abstract Interpretation

Abstract interpretation is a program verification analysis approach, introduced in 1988 by Cousot.

The program verification problem is undecidable; this implies that the possible solution to the program verification problem is by abstract interpretation to simplify the given proof obligation [17]. The abstract interpretation technique, starting from the analysed program concrete semantics definition and an abstract domain, derives the so-called abstract semantics, which allows to run the program on the abstract domain to compute the property that the abstract domain models.

In this chapter will be informally introduced the main concepts of the abstract interpretation analysis and will be defined the abstracts domains for the static analysis of string values, presented in the Section 1.1.

2.1 Concrete and abstract semantics

The concrete semantics of a program provides a formal mathematical model of all the possible behaviours of a computer system executing this program in interaction with any possible environment, where an environment can be regarded as a memory that holds the values for each program variable, and it is the foundation on which abstract interpretation is built [18].

The concrete semantics of a program is not computable and all the non-trivial properties on the concrete program semantics are either undecidable or very hard to solve. Abstract interpretation technique helps to prove specific program properties and it consists in considering an abstract semantics that is an over-approximation of the concrete one of a program. The abstract semantics guarantees correctness: if the abstract semantics is safe then it is also the concrete semantics.

2.1.1 Abstract domain

The abstraction can be parametrized in order to tune it with respect to different application domains. First of all we need to choose an abstract domain, that is an abstraction of the concrete semantics in the form of abstract properties and abstract operations [19]. So, an abstract domain is a complete lattice whose elements capture the information to be analysed. Then, it is necessary the definition of an abstract function α which maps a set of concrete objects (the concrete domain $\langle \mathcal{D}, \subseteq \rangle$) to its most accurate representation in the abstract domain ($\langle \bar{\mathcal{D}}, \sqsubseteq \rangle$). The inverse concretization function is γ, which maps an abstract object to the concrete one that it represents. The correspondence between concrete and abstract domains is given by a *Galois Connection* [21].

There exist non-relational and relational abstract domains. In a non-relational domain all the program variables are abstracted independently of the others. Relational domains are more precise than non-relational ones since the relationships between values of the program variables are preserved by the abstraction. Two well known non-relational abstract domains are the *sign domain* and the *interval domain.* The *sign abstract domain* consists in replacing integers by their sign thus ignoring their absolute value ($Sign = \{\perp, -, 0, +, \top\}$) [2]. The *interval abstract domain* approximates a set of integers by its minimal and maximal values ($Int = \{[x, y] | x, y \in \mathbb{Z} \cup \{-\infty, +\infty\}\}$) [18] [19].

2.2 Strings abstract domains

We introduce different abstract domains that have been proposed for string analysis. The *character inclusion domain*, the *prefix and suffix domains* and the *bricks domain*, defined in [11], the *string set and the constant string domain*, the *prefix-suffix domain*, the *string hash domain*, presented in [12], are generic abstract domains to the static analysis of string values, each of which tracks a different kind of information.

2.2.1 Character inclusion domain

The *character inclusion domain* [11], denoted by $\overline{\mathcal{CI}}$, approximates strings with the characters we know the strings surely contain, and once they could contain. In this domain, a string will be represented by a pair of sets, the set of certainly contained characters $\overline{\mathsf{C}}$ and the set of maybe contained characters $\overline{\mathsf{MC}}$: $\overline{\mathcal{CI}} = \{(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) : \overline{\mathsf{C}}, \overline{\mathsf{MC}} \in \wp(\mathsf{K}) \wedge \overline{\mathsf{C}} \subseteq \overline{\mathsf{MC}}\} \cup \perp_{\overline{\mathcal{CI}}}$, where K is a given alphabet.

2.2.2 Prefix and suffix domains

The *prefix domain* [11], denoted by $\overline{\mathcal{PR}}$, abstracts strings by their prefix. A prefix is represented by a sequence of characters followed by an asterisk *. The asterisk represents any string. Because the asterisk * at the end of the representation is always present, it is not included in the domain and abstract elements are considered made only of sequence of characters. Formally: $\overline{\mathcal{PR}} = \mathsf{K}^* \cup \perp_{\overline{\mathcal{PR}}}$, where K^* is a sequence of characters.

The *suffix domain*, denoted by $\overline{\mathcal{SU}}$, abstracts strings by the end of a certain sequence of characters. Formally: $\overline{\mathcal{SU}} = \mathsf{K}^* \cup \perp_{\overline{\mathcal{SU}}}$.

2.2.3 Bricks domain

The *bricks domain* [11], denoted by $\overline{\mathcal{BR}}$, approximates strings by a sequence of bricks. A single brick is defined by $\overline{\mathcal{B}} = [\wp(\mathsf{S})]^{\mathsf{min},\mathsf{max}}$, where min and max are two integer positive values and S is the set of all strings. A brick represents all the strings, which can be built through concatenation of the given strings, taken between min and max times altogether. Elements in $\overline{\mathcal{BR}}$ represent strings as ordered lists of bricks. Formally, concatenation between bricks is defined as follows: $\overline{\mathsf{B}}_1\overline{\mathsf{B}}_2 = \{\alpha\beta : \alpha \in \mathit{strings}(\overline{\mathsf{B}}_1) \wedge \beta \in \mathit{stirngs}(\overline{\mathsf{B}}_2)\}$, where $\mathit{strings}(\overline{\mathsf{B}})$ represents all the strings that can be built from the single brick $\overline{\mathsf{B}}$. Because a particular set of strings could be represented by more than one combination of bricks, it has been adopted a normalized form: $\overline{\mathit{normBricks}}(\overline{\mathsf{L}})$ is the function that, given a list of bricks $\overline{\mathsf{L}}$ returns its normalized version. So, the abstract domain of bricks is defined by $\overline{\mathcal{BR}} = \overline{\mathit{normBricks}}(\overline{\mathcal{B}}^*)$, that is, the set of all finite normalized sequences of bricks.

2.2.4 String set domain and constant string domain

The *string set domain* [12], denoted by $\mathcal{SS}$, enables precise representation of at most $k \geqslant 1$ concrete strings. Formally, $\mathcal{SS}_k = \{\top_{\mathcal{SS}_k}\} \cup \{S \in \wp(\mathsf{K}^*) \mid |S| \leqslant k\}$. One instance of $\mathcal{SS}_k$ is the *constant string ($\mathcal{CS}$) domain*, which is able to represents a single concrete string exactly (i.e., $\mathcal{CS} = \mathcal{SS}_k$).

2.2.5 Prefix-Suffix domain

The *prefix-suffix domain* [12], denoted by $\mathcal{PS}$, is a pair $\langle p, s\rangle \in \mathsf{K}^* \times \mathsf{K}^*$, corresponding to all the concrete strings that start as p and end as s. The domain is $\mathcal{PS} = \{\bot_{\mathcal{PS}}\} \cup (\mathsf{K}^* \times \mathsf{K}^*)$.

2.2.6 String hash domain

The *string hash domain* [12], denoted by $\mathcal{SH}$, was proposed by Madsen and Andreasen [20]. For some fixed integer range $U = [0, b]$ and hash function $h : \mathsf{K}^* \to U$, a concrete string s is mapped into a "bucket" of U according to the sum of the characters codes of s.

Chapter 3

Array Content Analysis

In this chapter will be introduced the array content analysis basic elements defined by Cousot in [1] and will be provided several examples, in order to be able to better understand the subsequent char array content analysis, *ad hoc* extension of the first one, object of this thesis. We have also introduced appropriate changes of notation, in order to make it coherent throughout the whole analysis.

Cousot defined `FunArray`, a parametric segmentation abstract domain functor for the fully automatic and scalable analysis of array content properties. The analysis automatically and semantically divides array into consecutive non-overlapping possibly empty segments. Segments are delimited by sets of bound expressions and abstracted uniformly. All symbolic expressions appearing in a bound set are equal in the concrete. The `FunArray` can be naturally combined, via reduced product, with any existing analysis for scalar variables. Cousot presented the analysis as a general framework, parametrized by the choice of bound expressions, segment abstractions and the reduction operator. Once the functor has been instantiated with fixed parameters, the analysis is fully automatic.

3.1 Concrete Semantics

In this section will be reported the elements of the semantics of programming languages to which the Cousot array content analysis does applies, such as: scalar variables, simple expressions and unidimensional arrays and corresponding assignments.

3.1.1 Scalar variables semantics

The operational semantics of scalar variables with basic types (`bool`, `char`, `int`, `float`, etc.) is assumed to be expressed by concrete variable environments $\rho \in \mathcal{R}_v$, where $\mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathcal{V}$, mapping variable names $\mathtt{x} \in \mathbb{X}$ to their values $\rho(\mathtt{x}) \in \mathcal{V}$.

We use the notation $[\![\mathtt{x}]\!]\rho \triangleq \rho(\mathtt{x})$.

Example **3.1.** Let $\mathtt{x} \in \mathbb{X}$ a program variable. Assuming that the values in $\mathcal{V}$ are integers, an environment $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathbb{Z}$ maps the variable $\mathtt{x}$ to its value $\rho(\mathtt{x})$. For instance, if $\rho(\mathtt{x}) = 10$, we say $[\![\mathtt{x}]\!]\rho = 10$. $\triangle$

3.1.2 Simple expressions semantics

The simple expressions $\mathtt{e} \in \mathbb{E}$, containing only constant, scalar variables, and mathematical unary and binary operators, have a semantics $[\![\mathtt{e}]\!]\rho$ in the concrete variable environment ρ so that $[\![\mathtt{e}]\!] \in \mathcal{R}_v \mapsto \mathcal{V}$, where $\mathcal{V}$ is any type of values.

The semantics of scalar variable assignment is $[\![\mathtt{x := e}]\!]\rho \triangleq \rho[\mathtt{x} \mapsto [\![\mathtt{e}]\!]\rho]$ where $\rho[\mathtt{x} \mapsto a](\mathtt{x}) = a$ and $\rho[\mathtt{x} \mapsto a](\mathtt{y}) = \rho(\mathtt{y})$ when $\mathtt{x} \neq \mathtt{y}$.

Example **3.2.** Let $\mathtt{x} \in \mathbb{X}$ a program variable and let the environments be:

$$v \in \mathcal{V} \triangleq \mathbb{Z}$$
$$\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathcal{V}$$

The value of an arithmetic expression like $\mathtt{x} + 2$ depends on the value of the free variable $\mathtt{x}$ in the expression. We take $\rho(\mathtt{x}) = 10$ then, the value of the expression $\mathtt{x} + 2$ is given by $\rho(\mathtt{x}) + 2$. Formally:

$$\begin{array}{lclcl} [\![\mathtt{x} + 2]\!]\rho & \triangleq & \rho(\mathtt{x} + 2) & = & \rho(\mathtt{x}) + 2 \\ & & & = & 10 + 2 \\ & & & = & 12 \end{array}$$

$\triangle$

Example **3.3.** Given $\mathtt{x} := \mathtt{x} + 2$ and $\rho(\mathtt{x}) = 10$, then:

$$\begin{array}{lclcl} [\![\mathtt{x} := \mathtt{x} + 2]\!]\rho & \triangleq & \rho[\mathtt{x} \mapsto [\![\mathtt{x} + 2]\!]\rho](\mathtt{x}) & = & \rho[\mathtt{x} \mapsto ([\![\mathtt{x}]\!]\rho + 2)](\mathtt{x}) \\ & & & = & \rho[\mathtt{x} \mapsto (\rho(\mathtt{x}) + 2)](\mathtt{x}) \\ & & & = & \rho[\mathtt{x} \mapsto (10 + 2)](\mathtt{x}) \\ & & & = & \rho[\mathtt{x} \mapsto 12](\mathtt{x}) \\ & & & = & 12 \end{array}$$

$\triangle$

3.1.3 Unidimensional arrays semantics

Cousot, in [1], relates array element values with their indexes, assuming that the concrete value of an array `A` is a quadruple $a = (\rho, \texttt{A.low}, \texttt{A.high}, A) \in \mathcal{A}$, where:

- $\rho \in \mathcal{R}_v$ is a scalar variable environment;
- $\texttt{A.low} \in \mathbb{E}$ is an expression which value, $[\![\texttt{A.low}]\!]\rho$, evaluated in the variable environment ρ, yields the integer lower bound of the array indexes;
- $\texttt{A.high} \in \mathbb{E}$ is an expression which value, $[\![\texttt{A.high}]\!]\rho$, evaluated in the variable environment ρ, yields the integer upper bound of the array indexes;
- A maps every index $i \in \big[[\![\texttt{A.low}]\!]\rho, [\![\texttt{A.high}]\!]\rho\big)$ to the pair $A[i] = (i, v)$ of the index i and the corresponding array element value v.

The operational semantics of array variables (such as $\texttt{A} \in \mathbb{A}$) are concrete array environments $\theta \in \mathcal{R}_a$, where $\mathcal{R}_a \triangleq \mathbb{A} \mapsto \mathcal{A}$, mapping array names $\texttt{A} \in \mathbb{A}$ to their values $\theta(\texttt{A}) \in \mathcal{A} \triangleq \mathcal{R}_v \times \mathbb{E} \times \mathbb{E} \times (\mathbb{Z} \mapsto (\mathbb{Z} \times \mathcal{V}))$.

As we all know, an array is a collection of data that holds fixed number of values of the same type. The size and type of arrays cannot be changed after its declaration. Moreover, we can access elements of an array by indexes and arrays always have 0 as the first index. It is also important to remember that, given an array `A` with length $|\texttt{A}| = n$ and an index i, that describes the position of a given element within the array itself, $i \in [0, n)$. So, supposing we have declared an array of n elements, we can use the array members from 0 to $n - 1$; if we try to access array elements outside of its bounds, the compiler may not show any error, however, this may cause unexpected output (undefined behaviour). Informally, when we talk about sound static analysis of program arrays, we have to consider the existence of a symbolic array lower and upper bound. The presence of these bounds is regardless of the array content. Given that, when we perform the array segmentation analysis presented by Cousot and we define the syntax quadruple value of an array, we can easily assume that `A.low` and `A.high` can be always re-conducted to the symbolic expressions whose results are, respectively, zero and the length of the analysed array, that always exists, even if its value is unknown. Indeed, Cousot considers a "buffer overrun" runtime error when $i < [\![\texttt{A.low}]\!]\rho$ or $i > [\![\texttt{A.high}]\!]\rho$, in which case the value of `A[e]` (that identifies the semantics of an array element access where the expression `e` is evaluated to an index i) is undefined, so that program execution is assumed to stop. The explicit inclusion of the array bounds in a is also useful to handle arrays of parametric length such as JavaScript arrays or collections in managed languages.

Furthermore, the instrumented semantics of arrays, defined by Cousot, makes explicit the fact that arrays relate indexes to indexed element values by considering array elements to be a pair of an index and an array element value. This instrumented semantics is in contrast with the classical semantics $a \in [l, h) \mapsto \mathcal{V}$ of arrays, mapping indexes in $[l, h)$ to array element values in $\mathcal{V}$. Storing (i, v) instead of v is useless but for the fact that the instrumented semantics can be used to make the array content analysis more precise and handles relational abstract analysis, as we will see later.

Example **3.4.** Given the array $\texttt{A} = < 7, 2, 9, 4 >$, of type `int`, with length $|\texttt{A}| = 4$ and $i \in [0, 4)$, its syntactically representation is given by the tuple $a = (\rho, \texttt{A.low}, \texttt{A.high}, A)$. Assuming that:

$$v \in \mathcal{V} \triangleq \mathbb{Z}$$
$$[\![\texttt{e}]\!]\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathcal{V}$$

`A.low` and `A.high`, evaluated in the environment ρ, are respectively equal to $[\![\texttt{A.low}]\!]\rho \triangleq \rho(\texttt{A.low}) = 0$ and $[\![\texttt{A.high}]\!]\rho \triangleq \rho(\texttt{A.high}) = 4$. The co-domain of the a quadruple parameter A corresponds to the set of all the pairs $\{(i, v) \mid i \in \big[[\![\texttt{A.low}]\!]\rho, [\![\texttt{A.high}]\!]\rho\big)\}$. So, we have that $codom(A) = \{\texttt{A}[0] = (0, 7),\ \texttt{A}[1] = (1, 2),\ \texttt{A}[2] = (2, 9),\ \texttt{A}[3] = (3, 4)\}$.

$\triangle$

3.2 Abstract Domains and Functors

An *abstract domain* $\mathbf{D}$ includes a set $\overline{\mathcal{D}}$ of abstract properties as well as abstract functions and operations $\mathbf{D}.op$ for the partial order structure of abstract properties ($\sqsubseteq$), the join ($\sqcup$), the meet ($\sqcap$), convergence acceleration operators: widening ($\triangle$) and narrowing ($\triangledown$), the abstract property transformers involved in the definition of the semantics of the programming language: the abstract evaluation of program arithmetic and boolean expressions, the assignment to scalar variables ... [4]. A monotonic concretization function γ provides the meaning of abstract properties in terms of concrete properties.

An *abstract domain functor* $\mathbf{D}$ is a function from the parameter abstract domains $\mathbf{D}_1, \ldots, \mathbf{D}_n$ to a new abstract domain $\mathbf{D}(\mathbf{D}_1, \ldots, \mathbf{D}_n)$. The formal parameters $\mathbf{D}_1, \ldots, \mathbf{D}_n$ of the abstract domain functor $\mathbf{D}$ can be instantiated to various actual abstract domains. The abstract domain functor $\mathbf{D}(\mathbf{D}_1, \ldots, \mathbf{D}_n)$ composes abstract properties $\overline{\mathcal{D}}_1, \ldots, \overline{\mathcal{D}}_n$ of the parameter abstract domains $\mathbf{D}_1, \ldots, \mathbf{D}_n$ to build a new class of abstract properties $\overline{\mathcal{D}}$ and operations.

EXAMPLE **3.5.** The domain refinement reduced product $\otimes$ is an abstract domain functor.

$\triangle$

3.2.1 Scalar variable abstraction

Let $\mathbf{X}$ be an abstract domain encoding program variables including a special variable $\mathtt{v}_0$ which value is assumed to be always zero; so, the elements of the domain are $\overline{\mathcal{X}} = \mathbb{X} \cup \{\mathtt{v}_0\}$ where $\mathtt{v}_0 \notin \mathbb{X}$.

Properties and property transformers of concrete variable environments in $\wp(\mathcal{R}_v)$ are abstracted by the variable environment abstract domain $\mathbf{R}(\mathbf{X})$ which depends on the variable abstract domain $\mathbf{X}$ (so that $\mathbf{R}$ is an abstract domain functor).

3.2.1.1 Concretization

The abstract properties $\overline{\rho} \in \overline{\mathcal{R}}$ are called abstract variable environments. The concretization $\gamma_v(\overline{\rho})$ denotes the set of concrete variable environments having this abstract property. It follows that $\gamma_v \in \overline{\mathcal{R}} \mapsto \wp(\mathcal{R}_v)$.

The static analysis of scalar variables may or may not be relational. For non-relational abstractions, $\wp(\mathcal{R}_v)$ is first abstracted to $\mathbb{X} \mapsto \wp(\mathcal{V})$ and $\overline{\mathcal{R}} \triangleq \mathbb{X} \mapsto \overline{\mathcal{V}}$ where the abstract domain $\mathbf{V}$ abstracts properties of values in $\mathcal{V}$ with concretization $\gamma_v \in \overline{\mathcal{V}} \mapsto \wp(\mathcal{V})$.

EXAMPLE **3.6.** Given the concrete variable environments $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathcal{V}$ and $\mathcal{V} = \mathbb{Z}$, let *Sign* be the sign abstract domain. Since *Sign* is a non-relational abstract domain, properties of concrete variable environments in $\wp(\mathcal{R}_v)$ are first abstracted to $\mathbb{X} \mapsto \wp(\mathbb{Z})$ and $\overline{\rho} \in \overline{\mathcal{R}} \triangleq$

$\mathbb{X} \mapsto \mathit{Sign}$, where the abstract domain Sign abstracts properties of values in $\mathbb{Z}$ with concretization $\gamma_v \in \mathit{Sign} \mapsto \wp(\mathbb{Z})$. So, given $\mathtt{x} \in \mathbb{X}$ and $\overline{\rho}(\mathtt{x}) = +$, its concretization is $\gamma_v(\overline{\rho}) = \{\rho \in \mathbb{X} \mapsto \mathbb{Z} \mid \rho(\mathtt{x}) > 0\}$.

$\triangle$

EXAMPLE **3.7.** Let $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathbb{Z}$ and consider the powerset operator abstract domain functor $\wp$ on Sign, such that $\wp(\mathit{Sign}) = \big\{\emptyset, [+], [-], [0], [+,-], [0,+], [-,0], [-,0,+]\big\}$ [5], with $\overline{\rho} \in \overline{\mathcal{R}} \triangleq \mathbb{X} \mapsto \wp(\mathit{Sign})$. The abstract domain $\wp(\mathit{Sign})$ abstracts properties of values in $\mathbb{Z}$ with concretization $\gamma_v \in \wp(\mathit{Sign}) \mapsto \wp(\mathbb{Z})$. So, given $\mathtt{x} \in \mathbb{X}$ and $\overline{\rho}(\mathtt{x}) = [0,+]$, its concretization is $\gamma_v(\overline{\rho}) = \{\rho \in \mathbb{X} \mapsto \mathbb{Z} \mid \rho(\mathtt{x}) = 0 \vee \rho(\mathtt{x}) > 0\}$.

$\triangle$

3.2.2 Expressions abstraction

The symbolic expressions appearing in segment bounds belong to the expression abstract domain $\mathbf{E}(\mathbf{X})$ that depends on the variable abstract domain $\mathbf{X}$ and $\mathbf{E}$ is an abstract domain functor. The abstract properties $\overline{\mathcal{E}}$ consist in a set of symbolic expressions depending on the variables in $\overline{\mathcal{X}}$ restricted to a canonical normal form plus the bottom expression, denoted by $\bot$, corresponding to unreachability and the top expression, denoted by $\top$, abstracting all symbolic expressions which cannot be put in the considered normal form.

The array bound expressions are assumed to be converted in canonical normal form. Different canonical forms for expressions correspond to different expression abstract domains $\mathbf{E}(\mathbf{X})$. In the analysis developed by Cousot, the abstract expressions $\overline{\mathcal{E}}$ are restricted to the normal form $\mathtt{v} + k$ where $\mathtt{v} \in \overline{\mathcal{X}}$ is an integer variable plus an integer constant $k \in \mathbb{Z}$ ($\mathtt{v}_0 + k$ represents the integer constant k).

3.2.2.1 Concretization

Given an abstract domain for scalar variables with concretization $\gamma_v \in \overline{\mathcal{R}} \mapsto \wp(\mathbb{X} \mapsto \mathbb{Z})$, the concretization $\gamma_e(\mathtt{e})\overline{\rho}$ of an expression $\mathtt{e} \in \overline{\mathcal{E}}$ depends on the abstract value $\overline{\rho} \in \overline{\mathcal{R}}$ of the scalar variables in $\overline{\mathcal{X}}$ and is the set of possible concrete values of the expression. So, $\gamma_e \in \overline{\mathcal{E}} \mapsto \overline{\mathcal{R}} \mapsto \wp(\mathcal{V})$ such that $\gamma_e(\bot)\overline{\rho} \triangleq \emptyset$, $\gamma_e(\top)\overline{\rho} \triangleq \mathcal{V}$, $\gamma_e(\mathtt{v}_0 + i)\overline{\rho} \triangleq \{i\}$, and otherwise $\gamma_e(\mathtt{v} + i)\overline{\rho} \triangleq \{\rho(\mathtt{v}) + i \mid \rho \in \gamma_v(\overline{\rho})\}$.

EXAMPLE **3.8.** Let the concrete variable environments $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathcal{V}$, with $\mathcal{V} = \mathbb{Z}$ and let *Sign* be the sign abstract domain. We want to know the sign of the arithmetic expression $\mathtt{x} + 2$, given that 2 is positive and $\overline{\rho}(\mathtt{x}) = +$. So:

$$\begin{array}{rcl} \overline{\rho}(\mathtt{x}+2) & = & \overline{\rho}(\mathtt{x}) \mathbin{\overline{+}} \overline{\rho}(2) \\ & = & + \mathbin{\overline{+}} + \\ & = & + \end{array}$$

The expression concretization is $\gamma_e(\mathtt{x}+2)\overline{\rho} = \{\rho(\mathtt{x}) + 2 \mid \rho \in \gamma_v(\overline{\rho})\}$ where $\gamma_v(\overline{\rho}) = \{\rho \in \mathbb{X} \mapsto \mathbb{Z} \mid \rho(\mathtt{x}) > 0\}$.

△

EXAMPLE **3.9.** Given the concrete variable environments $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathbb{Z}$ and the $\wp(Sign)$ abstract domain, we want to know the sign of the arithmetic expression $\mathtt{x} + 2$, given that 2 is positive and $\overline{\rho}(\mathtt{x}) = [-, 0]$. So:

$$\begin{array}{rcl} \overline{\rho}(\mathtt{x}+2) & = & \overline{\rho}(\mathtt{x}) \mathbin{\overline{+}} \overline{\rho}(2) \\ & = & [-, 0] \mathbin{\overline{+}} [+] \\ & = & [-, 0, +] \end{array}$$

The expression concretization is $\gamma_e(\mathtt{x}+2)\overline{\rho} = \{\rho(\mathtt{x}) + 2 \mid \rho \in \gamma_v(\overline{\rho})\}$ where $\gamma_v(\overline{\rho}) = \{\rho \in \mathbb{X} \mapsto \mathbb{Z} \mid \rho(\mathtt{x}) < 0 \ \vee \ \rho(\mathtt{x}) = 0\}$

△

3.2.3 Segment bounds abstract domain functor

The segment bound abstract domain functor **B** takes any of the expression abstract domains **E** previously discussed, and produces an instantiated segment bound abstract domain **B**(**E**), whose abstract properties are sets of expressions $\overline{\mathcal{B}} \triangleq \wp(\overline{\mathcal{E}} \setminus \{\bot, \top\})$. The empty set $\emptyset$ denotes unreachability, while, non-empty sets $\{e_1 \dots e_m\}$ of expressions $e_1, \dots, e_m \in \overline{\mathcal{E}}$ are all equivalent symbolic denotations of some concrete value (generally unknown in the abstract except when one of the e_i is a constant).

3.2.3.1 Concretization

The concretization $\gamma_b \in \overline{\mathcal{B}} \mapsto \wp(\mathcal{R}_v)$ of segment bounds is the set of scalar variables concrete environments ρ making the concrete values of all expressions in the set to be equal $[\![e_1]\!]\rho = \dots = [\![e_m]\!]\rho$. So, $\gamma_b(\emptyset) = \emptyset$

and $\gamma_b(S) = \{\rho \mid \forall \texttt{e}, \texttt{e'} \in S : [\![\texttt{e}]\!]\rho = [\![\texttt{e'}]\!]\rho\}$, where $[\![\texttt{e}]\!]\rho$ is the concrete value of expression $\texttt{e}$ in the concrete environment ρ.

When normal expressions and segment bounds are simplified and compared in the context of variable abstract environments $\overline{\rho} \in \overline{\mathcal{R}}$, the concretization can be chosen as $\gamma_b \in \overline{\mathcal{B}} \mapsto \overline{\mathcal{R}} \mapsto \wp(\mathcal{R}_v)$ such that $\gamma_b(S)\overline{\rho} = \{\rho \in \gamma_v(\overline{\rho}) \mid \forall \texttt{e}, \texttt{e'} \in S : [\![\texttt{e}]\!]\rho = [\![\texttt{e'}]\!]\rho\}$.

Example **3.10.** Consider:

- The concrete variable environments $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathbb{Z}$
- The sign abstract domain *Sign*
- An array `A`, of unspecified type, with length $|\texttt{A}| = 5$
- Two variables $\texttt{x}, \texttt{y} \in \mathbb{X}$ with $\rho(x) = 3$ and $\rho(y) = 5$

Assuming that $\{\texttt{x} + 2\ \ \texttt{y} + 0\}$ is the `A` segment upper bound then, $\texttt{e}_1 = \texttt{x} + 2$ and $\texttt{e}_2 = \texttt{y} + 0$ belong to E, the set of expressions in the considered bound. Since $\texttt{e}_1$ and $\texttt{e}_2$ belong to the same bound, $[\![\texttt{e}_1]\!]\rho$ and $[\![\texttt{e}_2]\!]\rho$ have to be equal. As a matter of fact:

$$[\![\texttt{x} + 2]\!]\rho \triangleq \rho(\texttt{x} + 2) = \rho(\texttt{x}) + 2 = 3 + 2 = 5$$

$$[\![\texttt{y} + 0]\!]\rho \triangleq \rho(\texttt{y} + 0) = \rho(\texttt{y}) + 0 = 5 + 0 = 5$$

In the concrete, $\texttt{e}_1$ and $\texttt{e}_2$ are both equal to a strictly positive integer value, so, under the sign abstract domain:

$$\begin{array}{rcl}
\overline{\rho}(\texttt{e}_1) & = & \overline{\rho}(\texttt{x} + 2) \\
& = & \overline{\rho}(\texttt{x}) \mathbin{\overline{+}} \overline{\rho}(2) \\
& = & + \mathbin{\overline{+}} + \\
& = & + \\
\overline{\rho}(\texttt{e}_2) & = & \overline{\rho}(\texttt{y} + 0) \\
& = & \overline{\rho}(\texttt{y}) \mathbin{\overline{+}} \overline{\rho}(0) \\
& = & + \mathbin{\overline{+}} 0 \\
& = & +
\end{array}$$

The abstracted segment upper bound of `A` corresponds to $\{+\ +\}$. Let $S = \{+,\ +\}$ be the set of the expressions in the abstracted `A` segment upper bound, its concretization is $\gamma_b(S)\overline{\rho} = \{\rho \in \gamma_v(\overline{\rho}) \mid \text{for } \texttt{e}_1, \texttt{e}_2 \in S : [\![\texttt{e}_1]\!]\rho = [\![\texttt{e}_2]\!]\rho\}$ where $\gamma_v(\overline{\rho}) = \{\rho \in \mathbb{X} \mapsto \mathbb{Z} \mid \rho(\texttt{x}) > 0 \land \rho(\texttt{y}) > 0\}$.

$\triangle$

<span style="font-variant:small-caps">Example</span> **3.11.** Consider:

- The concrete variable environments $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathbb{Z}$
- The abstract domain functor $\wp(Sign)$
- An array `A`, of unspecified type, with length $|\mathtt{A}| > 0$
- Two variable $\mathtt{x}, \mathtt{y} \in \mathbb{X}$

As in the previous example, the `A` segment upper bound is $\{\mathtt{x}+2\ \mathtt{y}+4\}$ and E is the set of expressions appearing in the considered bound, such that $E = \{\mathtt{x}+2, \mathtt{y}+4\}$. Assuming that:

i) $[\![\mathtt{x}+2]\!]\rho = [\![\mathtt{y}+4]\!]\rho$

ii) $\overline{\rho}(\mathtt{x}) = \overline{\rho}(\mathtt{y}) = [0,+]$

Then:

$$\begin{array}{rcl} \overline{\rho}(\mathtt{x}+2) & = & \overline{\rho}(\mathtt{x}) \mathbin{\overline{+}} \overline{\rho}(2) \\ & = & [0,+] \mathbin{\overline{+}} [+] \\ & = & [+] \end{array}$$

$$\begin{array}{rcl} \overline{\rho}(\mathtt{y}+4) & = & \overline{\rho}(\mathtt{y}) \mathbin{\overline{+}} \overline{\rho}(4) \\ & = & [0,+] \mathbin{\overline{+}} [+] \\ & = & [+] \end{array}$$

The abstracted segment upper bound of `A` corresponds to $\big\{[+]\ [+]\big\}$. Let $S = \big\{[+],\ [+]\big\}$ the set of the expressions values in the abstracted `A` segment upper bound, its concretization is $\gamma_b(E)\overline{\rho} = \{\rho \in \gamma_v \mid$ for $\mathtt{e}_1, \mathtt{e}_2 \in E : [\![\mathtt{e}_1]\!]\rho = [\![\mathtt{e}_2]\!]\rho\}$ where $\gamma_v(\overline{\rho}) = \{\rho \in \mathbb{X} \mapsto \mathbb{Z} \mid \rho(\mathtt{x}) \geqslant 0 \wedge \rho(\mathtt{y}) \geqslant 0\}$.

△

3.2.4 Array element abstract domain

The array element abstract domain **A** abstracts properties of pairs *(index, value of indexed array element).*

3.2.4.1 Concretization

The concretization is $\gamma_a \in \overline{\mathcal{A}} \mapsto \wp(\mathbb{Z} \times \mathcal{V})$.

Properties in $\wp(\mathbb{Z} \times \mathcal{V})$ may not or may be first abstracted to $\wp(\mathcal{V})$ when we do not want to relate array element values to their indexes. In the first case we have a relational analysis, in the second a non-relational.

Example **3.12.** Consider:

- The concrete variable environments $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathbb{Z}$
- The sign abstract domain *Sign*
- A non-relational analysis
- An array `A`, of type `int`, such that:

$$[\![(i, v)]\!]\overline{\rho} = [\![v]\!]\overline{\rho} = +, \forall i \in \big[[\![\texttt{A.low}]\!]\rho, [\![\texttt{A.high}]\!]\rho\big)$$

- The set V of all the `A` array elements values

The array elements concretization is $\gamma_a(+)\overline{\rho} = \{\rho \in \gamma_v(\overline{\rho}) \mid \forall v \in V : [\![v]\!]\rho > 0\}$.

△

Example **3.13.** Consider:

- The concrete variable environments $\rho \in \mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathbb{Z}$
- The powerset of sign abstract domain $\wp(Sign)$
- A non-relational analysis
- An array `A`, of type `int`, such that:

$$[\![(i, v)]\!]\overline{\rho} = [\![v]\!]\overline{\rho} = [0, +], \forall i \in \big[[\![\texttt{A.low}]\!]\rho, [\![\texttt{A.high}]\!]\rho\big)$$

- The set V of all the `A` array elements values

The array elements concretization is $\gamma_a([0, +])\overline{\rho} = \{\rho \in \gamma_v(\overline{\rho}) \mid \forall v \in V : [\![v]\!]\rho = 0 \vee [\![v]\!]\rho > 0\}$.

△

3.2.5 Array segmentation abstract domain functor

The array segmentation abstract domain $\mathbf{S}(\mathbf{B}(\mathbf{E}), \mathbf{A}, \mathbf{R})$ abstracts a set of possible array contents by consecutive, non-overlapping segments covering all array elements.

Abstract predicates

The Cousot array segmentation analysis has been instantiated with constant propagation [3], thus automatically producing abstract invariant predicates. `A:p`i is the abstract invariant predicate at program point $i \in [\mathtt{0}, \mathtt{n}]$, related to array `A`. The array segmentation abstract predicate `p`i belong to the set $\overline{\mathcal{S}} \triangleq \{(\overline{\mathcal{B}} \times \overline{\mathcal{A}}) \times (\overline{\mathcal{B}} \times \overline{\mathcal{A}} \times \{\text{␣},?\})^k \times \{\overline{\mathcal{B}} \times \{\text{␣},?\} \mid k \geqslant 0\} \cup \{\bot\}$ and has the form $\{e^1_1 \ldots e^1_{m^1}\}P_1\{e^2_1 \ldots e^2_{m^2}\}[?^2]P_2 \ldots\ P_{n-1}\{e^n_1 \ldots e^n_{m^n}\}[?^n]$, where:

- n is the length of the array `A`;
- the segment bounds $\{e^i_1 \ldots e^i_{m^i}\} \in \overline{\mathcal{B}},\ i \in [1, n],\ n > 1$, are finite non-empty sets of symbolic expressions in normal form $e^i_j \in \overline{\mathcal{E}}$;
- $P_i \in \overline{\mathcal{A}}$ is an abstract predicate chosen in an abstract domain $\mathbf{A}$ denoting possible values of pairs (index, indexed array element) in a segment; and
- the optional question mark $[?^i]$ follows the upper bound of a segment. Its presence, denoted by `?`, means that the segment might be empty. Its absence, denoted by ␣, means that the segment cannot be empty. Because this information is attached to the segment upper bound (which is also the lower bound of the next segment), the lower bound $\{e^1_1 \ldots e^1_{m^1}\}$ of the first segment never has a question mark. The tuple $(\{\text{␣},?\}, \preceq, \text{␣}, ?, \curlyvee, \curlywedge)$ form a complete lattice with $\text{␣} \prec ?$.

The symbolic expressions $e^k_i \in \overline{\mathcal{E}}$ in a given segment bound depend on scalar variables but not on array elements.

Definition 3.1 (*Seg function*)**.** We denote by α_{Seg} the function that maps an array `A` to the array representation $\overline{\mathcal{A}}$.

◊

3.2.5.1 Concretization

Given the concretization $\gamma_v \in \overline{\mathcal{R}} \mapsto \wp(\mathcal{R}_v)$ for the variable abstract domain and $\gamma_a \in \overline{\mathcal{A}} \mapsto \wp(\mathbb{Z} \times \mathcal{V})$ for the array elements abstract domain, the concretization γ_s of an abstract array segmentation is an array property, so $\gamma_s \in \overline{\mathcal{S}} \mapsto \overline{\mathcal{R}} \mapsto \wp(\mathcal{A})$. The concretization of a segment $B\ P\ B'\ [?]$ is the set of arrays whose elements in the segment $[B, B')$ satisfy the abstract property P ($<_{\llcorner\!\lrcorner}$ stands for $<$, while, $<?$ stands for $\leqslant$):

$$\begin{aligned}
&\gamma_s'(B\ P\ B'\ [?])\overline{\rho} \triangleq \\
&\quad \{(\rho, l, h, A) \mid \rho \in \gamma_v(\overline{\rho})\ \wedge\ \forall \mathsf{e}_1, \mathsf{e}_2 \in B : \forall \mathsf{e}_1', \mathsf{e}_2' \in B' : \\
&\qquad \llbracket l \rrbracket\rho \leq \llbracket \mathsf{e}_1 \rrbracket\rho = \llbracket \mathsf{e}_2 \rrbracket\rho < [?] \llbracket \mathsf{e}_1' \rrbracket\rho = \llbracket \mathsf{e}_2' \rrbracket\rho \leq \llbracket h \rrbracket\rho\ \wedge \\
&\qquad\quad \forall i \in [\llbracket \mathsf{e}_1 \rrbracket\rho, \llbracket \mathsf{e}_1' \rrbracket\rho) : A[i] \in \gamma_a(P)\}
\end{aligned}$$

The concretization of an array segmentation $B_1 P_1 B_2 [?^2] P_2\ \ldots\ P_{n-1} B_n$ $[?^n]$ is the set of arrays whose elements in all segments $[B_i, B_{i+1}), i = 1, \ldots, n-1$ satisfy abstract property P_i and whose lower and upper bounds are respectively given by B_1 and B_n.

$$\begin{aligned}
&\gamma_s(B_1 P_1 B_2 [?^2] P_2\ \ldots\ P_{n-1} B_n [?^n])\overline{\rho} \triangleq \\
&\quad \{(\rho, l, h, A) \in \bigcap_{i=1}^{n-1} \gamma_s'(B_i\ P_i\ B_{i+1}[?^{i+1}])\overline{\rho} \mid \\
&\qquad \forall \mathsf{e}_1 \in B_1 : \llbracket \mathsf{e}_1 \rrbracket\rho = \llbracket l \rrbracket\rho\ \wedge\ \forall \mathsf{e}_n \in B_n : \llbracket \mathsf{e}_n \rrbracket\rho = \llbracket h \rrbracket\rho\}
\end{aligned}$$

and $\gamma_s(\perp) = \emptyset$

Example **3.14.** Let A be an array, of type int, with length $|\mathtt{A}| = 10$ and consider the sign abstract domain *Sign*, abstracting the A array's elements (no abstraction is applied to the expressions in the array segment bounds) and a non-relational analysis. Suppose that the A abstract segmentation corresponds to A: $\{\mathtt{0}\}\ +\ \{\mathtt{10}\}$ then, its concretization is $\gamma_s(\{\mathtt{0}\}\ +\ \{\mathtt{10}\})\overline{\rho} \triangleq \{(\rho, l, h, A) \mid \rho \in \gamma_v(\overline{\rho}) \wedge \mathtt{0} \in \{\mathtt{0}\} : \mathtt{10} \in \{\mathtt{10}\} : \llbracket l \rrbracket\rho \leqslant \mathtt{0} < \mathtt{10} \leqslant \llbracket h \rrbracket\rho\ \wedge\ \forall i \in [0, 10) : A[i] \in \gamma_a(+)\}$.

$\triangle$

3.2.6 Segmentation unification

Given two segmentations with *compatible* extremal segment bounds (in general for the same array) or when given two segmentations, the first (the last) segment bounds should have a non-empty intersection, the objective of segmentation unification is to modify the two segmentations so that they coincide. In practice this is always the case as the first segment bound always contains `0` and the last segment bound always contains the symbolic name for the array length. Basically:

$$\begin{aligned}&unify(\{e_1^1\ \dots\ e_{m^1}^1\}\ P_1\ \{e_1^2\ \dots\ e_{m^2}^2\}[?^2]\ P_2\ \dots\ P_{n-1}\{e_1^n\ \dots\ e_{m^n}^n\}[?^n],\\ &\quad \{f_1^1\ \dots\ f_{m^1}^1\}\ P_1'\ \{f_1^2\ \dots\ f_{m^2}^2\}[?^2]'\ P_2'\ \dots\ P_{n-1}'\ \{f_1^n\ \dots\ f_{m^n}^n\}[?^n]') =\\ &\quad\quad \{e_1^1\ \dots\ e_{m^1}^1\} \cap \{f_1^1\ \dots\ f_{m^1}^1\}\ P_1 \sqcup P_1'\ \{e_1^2\ \dots\ e_{m^2}^2\} \cap \{f_1^2\ \dots\ f_{m^2}^2\}\\ &\quad\quad\quad [?^2] \sqcup [?^2]'\ P_2 \sqcup P_2'\ \dots\ P_{n-1} \sqcup P_{n-1}'\ \{e_1^n\ \dots\ e_{m^n}^n\} \cap \{f_1^n\ \dots\ f_{m^n}^n\}\\ &\quad\quad\quad\quad [?^n] \sqcup [?^n]'\end{aligned}$$

The problem of segmentation unification admits a partially ordered set of solutions, in general not forming a lattice. The minimal elements, hence the least precise unifications are those where all the segments are joined, and only the extremes are preserved. On the other end, the maximal elements, hence the most precise unification are the coarsest common refinements of both segmentations. Roughly speaking, what can be considered a "good" segmentation unification is such that the bounds: (i) do appear in one or the other initial segmentation; and (ii) preserve the original orderings.

It is important to note that one segment can be empty in one segmentations (like {`0 i`}) and non-empty in the other one (like {`0`} P {`1 i`}). Therefore segmentation must include the splitting of empty segments (like {`0 i`} $\rightarrow$ {`0`} P' {`1 i`}). Such an empty segment splitting is used in the comparison/join/meet/widening/narrowing of segments (which are not all commutative) so that the abstract value P' of the created empty segment must be chosen as the left/right neutral element of the considered operation (e.g. P' is $\bot$ for join, $\top$ for meet, $\bot$ on the left and $\top$ on the right of the partial order $\sqsubseteq$).

The segmentations involved in a unification are usually related to different program contexts; therefore, the well-definedness of the coarsest common refinement, if any, depends on the abstract variable environment, too. To sum up, the array segmentation analysis should have: (i) the possibility of being completely independent of the variable analysis; (ii) a deterministic behaviour in presence of several maximal common refinements. Therefore, Cousot presented a segmentation unification which does not provide any guarantee on the maximality of the result, but instead one which: i) is always well-defined in absence of knowledge of the contexts of the segmentations; ii) does terminate; (iii) is deterministic.

Algorithm 3.1 (*Segmentation unification by Cousot - Unify algorithm*)**.** The first step of the algorithm is checking the compatibility of the two input segmentations to verify that they do have common lower and upper bounds. Then, the unification proceeds recursively from left to right and maintains the invariant that the left part is already unified. Let I_l (resp. I_r) denote the left (resp. right) neutral element.

1 $B[?_1]\ P_1\ B'_1[?'_1]\ \ldots$ and $B[?_2]\ P_2\ B'_2[?'_2]\ \ldots$ have some lower bounds and so keep the first segments as they are and go on with $B'_1[?'_1]\ \ldots$ and $B'_2[?'_2]\ \ldots$

2 In case $(B \cup B_1)[?_1]\ P_1\ B'_1[?'_1]\ \ldots$ and $B[?_2]\ P_2\ B'_2[?'_2]\ \ldots$ with $B_1 \neq \emptyset$ and $B \cap B_1 = \emptyset$, let $\overline{B}_1$ be the set of expressions in B_1 appearing in the second segmentation blocks $B'_2, \ldots$.

2.1 If $\overline{B}_1$ is empty, then go on with $B[?_1]\ P_1\ B'_1[?'_1]\ \ldots$ and $B[?_2]\ P_2$
$B'_2\ [?'_2]\ \ldots$ following case 1.

2.2 Otherwise, go on with $B[?_1]\ \mathrm{I}_l\ B_1?\ P_1\ B'_1[?'_1]\ \ldots$ and $B[?_2]\ P_2$ $B'_2[?'_2]\ \ldots$ as in case 1.

3 The symmetrical case is similar.

4 In case $(B \cup B_1)[?_1]\ P_1\ B'_1[?'_1]\ \ldots$ and $(B \cup B_2)[?_2]\ P_2\ B'_2[?'_2]\ \ldots$ with $B_1, B_2 \neq \emptyset$ and $B \cap B_1 = B \cap B_2 = \emptyset$, let $\overline{B}_1$ (resp.$\overline{B}_2$) be the set of expressions in B_1 (resp. B_2) appearing in the second (resp. first) segmentation blocks $B'_2, \ldots$ ($B'_1, \ldots$).

4.1 If $\overline{B}_1$ and $\overline{B}_2$ are both empty, go on with $B[?_1]\ P_1\ B'_1[?'_1]\ \ldots$ and $B[?_2]\ P_2\ B'_2[?'_2]\ \ldots$ as in case 1.

4.2 Else if $\overline{B}_1$ is empty (so that $\overline{B}_2$ in not empty) then go on with $B[?_1]\ P_1\ B'_1[?'_1]\ \ldots$ and $B[?_2]\ \mathrm{I}_r\ \overline{B}_2?\ P_2\ B'_2[?'_2]\ \ldots$ (where I_r is the right neutral element).

4.3 The symmetrical case is similar.

4.4 Finally if $\overline{B}_1$ and $\overline{B}_2$ are both non-empty, then go on with $B[?_1]\ \mathrm{I}_l$
$\overline{B}_1?\ P_1\ B'_1[?'_1]\ \ldots$ and $B[?_2]\ \mathrm{I}_r\ \overline{B}_2?\ P_2\ B'_2[?'_2]\ \ldots$ as in case 1.

5 In case $B_1[?_1]\ P_1\ B'_1[?'_1]\ \ldots$ and $B_2[?_2]\ P_2\ B'_2[?'_2]\ \ldots$ with $B_1 \cap B_2 = \emptyset$, we cannot be on the first left segment block so we have on the left $B_0[?_0]$ P_0 $B_1[?_1]$ P_1 $B_1'[?_1']$ $\ldots$ and $B_0'[?_0']$ P_0' $B_2[?_2]$ P_2 B_2' $[?_2']$ $\ldots$ and go on by merging these consecutive blocks $B_0[?_0]$ $P_0 \sqcup P_1$ $B_1'[?_1 \curlywedge ?_1']$ $\ldots$ and $B_0'[?_0']$ $P_0' \sqcup P_2$ $B_2[?_2 \curlywedge ?_2']$ $\ldots$.

6 Finally, at the end either we are left with the right limits that have both been checked to be equal or else we have $B_1[?_1]$ P_1 $B_1'[?_1']$ and $B_2[?_2]$ with $B_1' = B_2$. Because we have maintained the invariant that B_1 is always equal to B_2 in the concrete (so necessarily $[?_1'] =?$ since then $B_1 = B_2 = B_1'$), and so we end up with $(B_1 \cup B_1' \cup B_2)[?_1]$ and $(B_1 \cup B_1' \cup B_2)[?_2]$.

□

The algorithm never adds any new expression to the segment bounds nor increments the total number of segment bounds in splits and so does terminate.

Theorem 3.1. *The Unify algorithm is a sound implementation of an upper bound operator over* $\mathbb{A}$.

▽

Array segmentation rules

1. At each program point, the program arrays content is described by a segmentation without holes, since a hole can always be represented by a $\top$ segment, that is, a segment whose properties are unknown, possibly empty if the hole may or may not be absent.

   Note that the segments are not empty, except if the upper bound of the segment is marked with `?`.

2. A segmented array consists of consecutive non-overlapping segments.

3. Each segment uniformly describes the array elements within that segment bounds, but different segments of a segmentation can have different abstract properties.

4. The consecutive segment bounds are in strictly increasing order in the concrete except when followed by a question mark, meaning that the preceding block may be empty. The first block limit always contains an expression in normal form denoting the array lower bound, while, the last block always contains an expression in normal form denoting the array upper bound.

5. In case of program loop invariant the array segmentation analysis performs a segmentation unification.
6. The array content analysis always terminates since the only two reasons for non-termination are impossible: i) The array might have infinitely many symbolic segments as in {0} ... ... {n-3} ... {n-2} ... {n-1} ... {n} which is prevented by segmentation unification and widening; ii) A segment might take successive strictly increasing abstract values which is prevented by the use of a widening/narrowing convergence acceleration for segment content analysis [4]. No widening was necessary for constant propagation which satisfies the ascending chain condition ($\perp \sqsubset i \sqsubset \top$, $i \in \mathbb{Z}$).

The array content analysis presented by Cousot can be clarified through a suitably extended array initialization example [1], involving a loop invariant.

Example **3.15.** Given the program:

```
          void Init(int[] A) {
/* 0: */    int i = 0;
/* 1: */    while /* 2: */ (i < A.Length) {
/* 3: */      A[i] = 0;
/* 4: */      i = i+1;
/* 5: */    }
/* 6: */  }
```

The loop invariant at program point 2 states that if A.Length $= 0$ then i $= 0$ and the array A is empty. Otherwise A.Length $\geqslant 1$; in this case either i $= 0$ and the array A is not yet initialized, or i > 0 so that A$[0]$ = A$[1]$ = ... = A$[\mathtt{i}-1] = 0$. Formally, the invariant $(\mathtt{A.Length} = 0 \wedge \mathtt{i} = 0) \vee (\mathtt{A.Length} \geqslant 1 \wedge 0 \leqslant \mathtt{i} \leqslant \mathtt{A.Length} \wedge \forall j \in [0, \mathtt{i}) : \mathtt{A}[j] = 0)$ holds at point 2.

In our case A:pi is the abstract invariant predicate at program point $i \in$ [0,6] and we get:

A:p0 = A: {0} T {A.Length}?

A:p1 = A: {0 i} T {A.length}?

A:p2 = A: {0} 0 {i}? T {A.length}?

A:p6 = A: {0} 0 {A.Length i}?

- At the program point `0`, the array `A` content is described by the segmentation:

  $$\{\texttt{0}\}\ \texttt{T}\ \{\texttt{A.Length}\}?$$

  which captures the initial condition `A.Length` $\geqslant$ `0`.

- At the program point `1`, the array `A` content is described by the segmentation:

  $$\{\texttt{0 i}\}\ \texttt{T}\ \{\texttt{A.Length}\}?$$

  where the symbolic segment bound $\{$`0 i`$\}$ defines `i` $=$ `0`, since all expressions in a bound are equal. The `T` element is the top element of the domain **A** and it means that the array values, in the considered segment (lower bound included, upper bound excluded), are unknown ($\top$) and the symbolic segment bound $\{$`A.length`$\}$? leads to $0 =$ `i` $\leqslant$ `A.Length`, since the segment bounds are in increasing order (strictly increasing without ?). So, the invariant `A:p1` states that `i` $= 0 \leqslant$ `A.Length` $\wedge\, \forall j \in [0,$ `A.Length`$)$: `A`$[j] \in \mathbb{Z}$. In particular, when `i = A.Length` $= 0$, the interval $[0,$ `A.length`$)$ is empty, so the quantified expression holds vacuously.

  Given `A:p1 = A:` $\{$`0 i`$\}$ `T` $\{$`A.Length`$\}$? it is possible consider its lower segment bound, that, taken individually, can be interpret as an empty segment, and we can split it as follows:

  $$\{\texttt{0 i}\}\ \texttt{T}\ \{\texttt{A.Length}\}? \equiv \{\texttt{0}\}\ \perp\ \{\texttt{i}\}?\ \texttt{T}\ \{\texttt{A.Length}\}?$$

- At the program point `2`, the array `A` content is described by the segmentation:

  $$\{\texttt{0}\}\ \texttt{0}\ \{\texttt{i}\}?\ \texttt{T}\ \{\texttt{A.length}\}?$$

  How can we get to `A:p2`? Initially `A:p2` $=$ `A:p3` $= \ldots =$ `A:p5` $=$ $\perp$ denotes the unreachability of the loop so that the abstract loop invariant is initially `A:p2` $=$ `A:p1` $\sqcup$ `A:p5` $=$ `A:p1`, where, the join $\sqcup$ behaviour in the constant abstract domain for segments is:

- $x \sqcup \bot = \bot \sqcup x = x$
- $x \sqcup \top = \top \sqcup x = \top$
- $i \sqcup i = i$
- $i \sqcup j = \top$, when $i \neq j$

So, `A:p2 = A:p1 = A: {0 i} T {A.length}?`

First iteration:

`A:p3 = A: {0 i} T {A.Length}`

`A:p4 = A: {0 i} 0 {1 i+1} T {A.Length}?`

`A:p5 = A: {0} 0 {1 i} T {A.Length}?`

Moreover:

- $\textvisiblespace \curlyvee \textvisiblespace = \textvisiblespace$
- $\textvisiblespace \curlyvee\ ? = ?\ \curlyvee \textvisiblespace = ?\ \curlyvee\ ? = ?$

Now we have to unify `A:p1 = A: {0}` $\bot$ `{i}? T {A.length}?` and `A:p5 = A: {0} 0 {1 i} T {A.Length}?`. The segmentation unification produces:

- `{0}` $\cap$ `{0}` $=$ `{0}`
- $\bot \sqcup$ `0` $=$ `0`
- `{i}` $\cap$ `{1 i}` $=$ `{i}`
- $?\ \curlyvee \textvisiblespace = ?$
- `T` $\sqcup$ `T` $=$ `T`
- `{A.Length}` $\cap$ `{A.Length}` $=$ `{A.Length}`
- $? \curlyvee ? = ?$

`A:p2 = A:p1` $\cup$ `A:p5 = A: {0} 0 {i}? T {A.Length}?`

Second iteration:

`A:p3 = A: {0} 0 {i}? T {A.Length}`

`A:p4 = A: {0} 0 {i}? 0 {i+1} T {A.Length}?`

`A:p5 = A: {0} 0 {i-1}? 0 {i} T {A.Length}?`

`A:p2 = A:p1` $\cup$ `A:p5 = A: {0} 0 {i}? T {A.Length}?`.

Note that `A:p2` is a maximal solution.

After some iterations, the analysis reaches a fixpoint so, the invariant `A:p2` states that $0 \leqslant$ `i` $\leqslant$ `A.Length`, that `A[0] = A[1] =` $\ldots =$ `A[i-1]` $= 0$ when `i` > 0 and that the values `A[i], A[i+1]`, $\ldots$, `A[A.Length-1]` are unknown when `A.Length` $>$ `i`.

- At the program point `6`, the array `A` content is described by the segmentation:

  $$\{\mathtt{0}\}\ \mathtt{0}\ \{\mathtt{A.Length\ i}\}?$$

  that is `A:p2[i>=A.Length]`, where `A.Length = i`, since the segmentation of `A:p2` provides the information that $0 \leqslant$ `i` $\leqslant$ `A.Length`.

Furthermore, considering the additional assumption that `A.Length` $>$ 1, at program point `6` the final values of the scalar variables are given by ρ_6, such that: $\rho_6(\mathtt{i}) = \rho_6(\mathtt{A.Length}) = n$ where $n > 1$ is the unknown array length. The final value of `A` is $a_6 = (\rho_6, \mathtt{0}, \mathtt{A.Length}, A_6)$ with $A_6[i] = (i, 0)$, $\forall i \in [0, n)$.

$\triangle$

Example **3.16.** Given the program of the Example 3.15 and its abstract invariant predicates, it is possible perform an abstraction analysis on them. The employed abstraction uses the reduced product [6] of the powerset of sign and intervals where pairs of an element of the sign's powerset domain and an interval denote the conjunction of both properties. In the following analysis, the abstraction is used both for variables and array elements, ignoring their relationship to indexes, such that $[\![(i, v)]\!]\overline{\rho} = (\wp(Sign)([\![v]\!]\overline{\rho}), Int([\![v]\!]\overline{\rho}))$. Taking into account the previously performed analysis we get that:

$$\begin{aligned}
\mathtt{A:p0} &= (\mathtt{A:}\ \{\mathtt{0}\}\ (\top, [-\infty, +\infty])\ \{\mathtt{A.length}\}?,\\
&\quad\ \mathtt{A.length:}\ ([0, +],\ [0, +\infty]))\\
\mathtt{A.p1} &= (\mathtt{A:}\ \{\mathtt{0\ i}\}\ (\top, [-\infty, +\infty])\ \{\mathtt{A.length}\}?,\\
&\quad\ \mathtt{i:}\ (0, [0, 0]),\ \mathtt{A.length:}\ ([0, +],\ [0, +\infty]))\\
\mathtt{A.p2} &= (\mathtt{A:}\ \{\mathtt{0}\}\ (0, [0, 0])\ \{\mathtt{i}\}?\ (\top, [-\infty, +\infty])\ \{\mathtt{A.length}\}?,\\
&\quad\ \mathtt{i:}\ ([0, +], [0, +\infty]),\ \mathtt{A.length:}\ ([0, +],\ [0, +\infty]))\\
\mathtt{A.p6} &= (\mathtt{A:}\ \{\mathtt{0}\}\ (0, [0, 0])\ \{\mathtt{A.length\ i}\}?,\\
&\quad\ \mathtt{i:}\ ([0, +], [0, +\infty]),\ \mathtt{A.length:}\ ([0, +],\ [0, +\infty]))
\end{aligned}$$

$\triangle$

Example **3.17.** Given the program of the Example 3.15 and its abstract invariant predicates, it is also possible abstract them in a relational way. The employed abstraction uses the reduced cardinal power [6] of intervals by the powerset of sign, relating the sign of an array index to the interval of possible variation of the corresponding element, such that $[\![(i,v)]\!]\overline{\rho}$ is a map of $\wp(Sign)([\![i]\!]\overline{\rho})$ to interval $Int([\![v]\!]\overline{\rho})$. Since the indexes of an array take values in $[0,n)$, we can just consider indexes abstracted to the elements of the sign's powerset lattice equal to 0, $+$ and $[0,+]$. So:

$$\begin{aligned}
\texttt{A:p0} = (\texttt{A: } \{\texttt{0}\}\ & (0 \to [-\infty,+\infty], + \to [-\infty,+\infty],\\
& [0,+] \to [-\infty,+\infty])\ \{\texttt{A.length}\}?,\\
\texttt{A.length: }& ([0,+],\ [0,+\infty]))
\end{aligned}$$

$$\begin{aligned}
\texttt{A:p1} = (\texttt{A: } \{\texttt{0 i}\}\ & (0 \to [-\infty,+\infty], + \to [-\infty,+\infty],\\
& [0,+] \to [-\infty,+\infty])\ \{\texttt{A.length}\}?,\\
\texttt{i: }& (0,[0,0]),\ \texttt{A.length: } ([0,+],\ [0,+\infty]))
\end{aligned}$$

$$\begin{aligned}
\texttt{A:p2} = (\texttt{A: } \{\texttt{0}\}\ & (0 \to [0,0], + \to [0,0], [0,+] \to [0,0])\ \{\texttt{i}\}?\\
& (0 \to [-\infty,+\infty], + \to [-\infty,+\infty],\\
& [0,+] \to [-\infty,+\infty])\ \{\texttt{A.length}\}?,\\
\texttt{i: }& ([0,+],[0,+\infty]),\ \texttt{A.length: } ([0,+],\ [0,+\infty]))
\end{aligned}$$

$$\begin{aligned}
\texttt{A:p6} = (\texttt{A: } \{\texttt{0}\}\ & (0 \to [0,0], + \to [0,0], [0,+] \to [0,0])\ \{\texttt{A.length i}\}?,\\
\texttt{i: }& ([0,+],[0,+\infty]),\ \texttt{A.length: } ([0,+],\ [0,+\infty]))
\end{aligned}$$

△

Chapter 4

Char Array Segmentation Analysis

Strings, in C programming language, are represented as char arrays, containing the sequence of characters composing the string, plus the string terminating character: ‘`\0`’. Char arrays that do not contain the string terminating character, do not represent a string. Notice that a char array may contain more than one string but, only the first one is considered of our interest. If a char array contains more than one string, excluding the first one, the others can be intended ad the strings of some specific instances of the considered char array.

We apply the array segmentation abstract domain presented by Cousot in [1] to a program that manages a string initialization and we track the resulting information. Essentially, we want verify if the Cousot array content analysis is able to capture if the declared char array still contains a string of interest at the end of the program. For this purpose, we modify the program presented in the Example 3.15 and we perform the analysis.

Example **4.1.** Given the program:

```
           void Init(char[] C) {
/* 0: */     int i = 0;
/* 1: */     while /* 2: */ (i < C.Length-1) {
/* 3: */       C[i] = ‘a’;
/* 4: */       i = i+1;
/* 5: */     }
/* 6: */     if(C.Length > 0) {
/* 7: */       C[C.Length-1] = ‘\0’;
/* 8: */     }
/* 9: */   }
```

The `while` condition has been changed in order to reserve the space for the string terminating character in the char array program result, once it has been filled with one or more '`a`' characters. Furthermore, it has been added the final `if` condition in order to be sure to get a char array containing a string as result in the case in which `C.Lenght` is greater than 0. We get the following abstract predicates:

`C:p0 = C:` {0} T {C.Length}?

`C:p1 = C:` {0 i} T {C.Length}?

`C:p2 = C:p1 = C:` {0 i} T {C.Length}?

First iteration:

- if `C.Length` $= 0$ then `C:p6 = C:` {0 i} T {C.Length}? (in this case the program execution stops at the program point 6 since the `if` condition cannot be satisfied). Interpreting the analysis we get that `0` $=$ `i` $=$ `C.Length` and the segmented array program result can be rewritten as an empty segment `C:p6 = C:` $\emptyset$ that, obviously, does not represent a char array containing a string. Notice that in this case, the analysis does not explicitly capture the nature of the execution stop.

- if `C.Length` $= 1$ then `C:p8 = C:` {0 i} '\0' {C.Length}.

- if `C.Length` > 1 then:

  `C:p3 = C:` {0 i} T {C.Length}

  `C:p4 = C:` {0 i} 'a' {1 i+1} T {C.Length}

  `C:p5 = C:` {0} 'a' {1 i} T {C.Length}

  The next approximation of the loop invariant is `C:p2 = C:p1` $\cup$ `C:p5 = C:` {0} 'a' {i}? T {C.Length}?.

The second iteration is similar to the following ones and a fixpoint is immediately reached. It remains to compute `C:p6`, `C:p7`, `C:p8` and `C:p9`.

`C:p6 = C:` {0} 'a' {i}? T {C.Length}?

At this point of the analysis we know for sure that in the case in which the `if` condition is satisfied, then `C.Length` $> 0 \wedge$ `i` $=$ `C.Length` $- 1$ from which we derive that `i` is strictly smaller than `C.Length`, so:

`C:p7 = C: {0} ‘a’ {i}? T {C.Length}`

`C:p8 = C:p9 = C: {0} ‘a’ {i C.length-1}? ‘\0’ {C.length}`

$\triangle$

As we can notice from the Example 4.1, the Cousot array segmentation analysis applied on a program that manages the fully initialization of a string char array, results to be general and unable to track both the correctness of the program result and the information of interest, unless we do not interpret the analysis. The Cousot array segmentation analysis is a sophisticated representation of what is, at the static analysis level, an array, but can not infer any implicit information from it, specifically when it is applied on char arrays. In a char array segmentation analysis, we want to highlight so-called ***string of interest***, if it exists, compared to the whole char array content and we want a "method" able to identify the presence of that string of interest.

We refine the Cousot array segmentation (C-segmentation) so that it can be better applied to char arrays. The proposed char array segmentation analysis will be called M-String segmentation.

Definition 4.1 (*string of interest*)**.** Let `str` be an array of char, we consider the *string of interest* of `str` the function *string*(`str`) defined by: if ‘`\0`’ occurs in `str`, let k be the least index of `C` such that: `str`$[k] =$ ‘`\0`’ in *string*(`str`) $=<$ `str`$[i]$: $i \leqslant k >$, otherwise *string*(`str`) $=$ `undef`.

$\diamondsuit$

Definition 4.2 (*splitting*)**.** Let `str` be an array of char and given its C-segmentation `str:{str.low}`$\ldots$`{i}? ‘\0’ {i+1}`$\ldots$`{str.high}?`, where `{i}? ‘\0’ {i+1}` is the first segment in which occurs the string terminating character, we define by *split*(`str`) the pair $(\texttt{s}_{\texttt{str}}, \texttt{ns}_{\texttt{str}})$ such that:

- $\texttt{s}_{\texttt{str}}$: `{str.low}`$\ldots$`{i}?`
- $\texttt{ns}_{\texttt{str}}$: `{i+1}`$\ldots$`{str.high}?`

Where:

- $\mathtt{s}_{\mathtt{str}}$ corresponds to the sub-segmentation representing *string*(`str`) (the sequence of characters before the NULL one),
- $\mathtt{ns}_{\mathtt{str}}$ corresponds to the sub-segmentation of the char array content which falls outside the string of interest.

In the case in which `str` does not contain the '\0' character (i.e. *string*(`str`) = `undef`), then $\mathtt{s}_{\mathtt{str}} = \emptyset$. So, depending by the presence or not and by the position of the first string terminating character in the considered char array, we have that:

1. if `str`: $\{\mathtt{str.low}\}\ldots\{\mathtt{str.high}\}?$
   then $split(\mathtt{str}) = (\emptyset, \{\mathtt{str.low}\}\ldots\{\mathtt{str.high}\}?)$
2. if `str`: $\{\mathtt{str.low}\}$ '\0' $\{\mathtt{str.low+1}\}\ldots\{\mathtt{str.high}\}?$
   then $split(\mathtt{str}) = (\{\mathtt{str.low}\}_{\llcorner\!\lrcorner}, \{\mathtt{str.low+1}\}\ldots\{\mathtt{str.high}\}?)$
3. if `str`: $\{\mathtt{str.low}\}\ldots\{\mathtt{str.high-1}\}$ '\0' $\{\mathtt{str.high}\}$
   then $split(\mathtt{str}) = (\{\mathtt{str.low}\}\ldots\{\mathtt{str.high-1}\}, \emptyset)$
4. if `str`: $\{\mathtt{str.low}\}\ldots\{\mathtt{i}\}$ '\0' $\{\mathtt{i+1}\}\ldots\{\mathtt{str.high}\}$
   then $split(\mathtt{str}) = (\{\mathtt{str.low}\}\ldots\{\mathtt{i}\}, \{\mathtt{i+1}\}\ldots\{\mathtt{str.high}\})$

◊

In this way it is possible to take into account the needful presence of the string terminating character in a char array, to establish the existence of the string of interest in the analysed char array and to isolate the string of interest for further analysis on it.

Definition 4.3 (*unification*)**.** Let `str1` and `str2` be two char arrays and consider their splitting:

- $split(\mathtt{str1}) = \big(\mathtt{s}_{\mathtt{str1}}, \mathtt{ns}_{\mathtt{str1}}\big)$ and
- $split(\mathtt{str2}) = \big(\mathtt{s}_{\mathtt{str2}}, \mathtt{ns}_{\mathtt{str2}}\big)$,

we define the unification of them (where $\cup$ is the segmentation unification algorithm presented in the subsection 3.2.6) as:

$$\big(\mathtt{s}_{\mathtt{str1}}, \mathtt{ns}_{\mathtt{str1}}\big) \cup \big(\mathtt{s}_{\mathtt{str2}}, \mathtt{ns}_{\mathtt{str2}}\big) = \big(\mathtt{s}_{\mathtt{str1}} \cup \mathtt{s}_{\mathtt{str2}}, \mathtt{ns}_{\mathtt{str1}} \cup \mathtt{ns}_{\mathtt{str2}}\big)$$

◊

Example **4.2.** Recovering the program of the Example 4.1 and given the split segmentation abstract representation of its managed `C` char array, at each program point we get the following extended abstract predicates:

$\texttt{C:p0 = split(C):}\ \big(\emptyset,\ \{\texttt{0}\}\ \texttt{T}\ \{\texttt{C.Length}\}?\big)$

$\texttt{C:p1 = split(C):}\ \big(\emptyset,\ \{\texttt{0 i}\}\ \texttt{T}\ \{\texttt{C.Length}\}?\big)$

$\texttt{C:p2 = C:p1 = split(C):}\ \big(\emptyset,\ \{\texttt{0 i}\}\ \texttt{T}\ \{\texttt{C.Length}\}?\big)$

First iteration:

- if $\texttt{C.Length} = 0$ then $\texttt{C:p6 = split(C):}\ \big(\emptyset, \emptyset\big)$.
- if $\texttt{C.Length} = 1$ then $\texttt{C:p8 = split(C):}\ \big(\{\texttt{0 i}\}_{␣},\ \emptyset\big)$.
- if $\texttt{C.Length} > 1$ then:

  $\texttt{C:p3 = split(C):}\ \big(\emptyset,\ \{\texttt{0 i}\}\ \texttt{T}\ \{\texttt{C.Length}\}\big)$
  $\texttt{C:p4 = split(C):}\ \big(\emptyset,\ \{\texttt{0 i}\}\ \texttt{`a'}\ \{\texttt{1 i+1}\}\ \texttt{T}\ \{\texttt{C.Length}\}\big)$
  $\texttt{C:p5 = split(C):}\ \big(\emptyset,\ \{\texttt{0}\}\ \texttt{`a'}\ \{\texttt{1 i}\}\ \texttt{T}\ \{\texttt{C.Length}\}\big)$

  The next approximation of the loop invariant is $\texttt{C:p2 = C:p1} \cup \texttt{C:p5 =}\ \big(\emptyset,\ \{\texttt{0}\}\ \perp\ \{\texttt{i}\}?\ \texttt{T}\ \{\texttt{C.Length}\}?\big) \cup \big(\emptyset,\ \{\texttt{0}\}\ \texttt{`a'}\ \{\texttt{1 i}\}\ \texttt{T}\ \{\texttt{C.Length}\}\big)$.
  The segmentation unification produces:

  - $s_{\texttt{C:p1 = split(C)}} \cup s_{\texttt{C:p5 = split(C)}}$
    - $\emptyset \cup \emptyset = \emptyset$
  - $ns_{\texttt{C:p1 = split(C)}} \cup ns_{\texttt{C:p5 = split(C)}}$
    - $\{\texttt{0}\} \cap \{\texttt{0}\} = \{\texttt{0}\}$
    - $\perp \sqcup \texttt{`a'} = \texttt{`a'}$
    - $\{\texttt{i}\} \cap \{\texttt{1 i}\} = \{\texttt{i}\}$
    - $? \curlyvee ␣ = ?$
    - $\texttt{T} \sqcup \texttt{T} = \texttt{T}$
    - $\{\texttt{C.Length}\} \cap \{\texttt{C.Length}\} = \{\texttt{C.Length}\}$
    - $? \curlyvee ␣ = ?$

  $\texttt{C:p2 = split(C):}\ \big(\emptyset,\ \{\texttt{0}\}\ \texttt{`a'}\ \{\texttt{i}\}?\ \texttt{T}\ \{\texttt{C.Length}\}?\big)$.

The second iteration is similar to the following ones and a fixpoint is immediately reached. It remains to compute `C:p6`, `C:p7`, `C:p8` and `C:p9`.

`C:p6 = split(C):` $\big(\emptyset$, `{0} 'a' {i}? T {C.Length}?`$\big)$

`C:p7 = split(C):` $\big(\emptyset$, `{0} 'a' {i}? T {C.Length}`$\big)$

`C:p8 = C:p9 = split(C):` $\big($`{0} 'a' {i C.length-1}?`, $\emptyset\big)$

$\triangle$

As we can notice, the M-String char array content representation is able to infer the presence of well-formed strings at each point of the program presented in the Example 4.1, capturing the real nature of the execution stops and safely tracking the information flow.

4.1 Concrete Semantics

In this section we present an extension of the Cousot unidimensional arrays concrete semantics, suitable to the previously settled *ad hoc* char array split segmentation abstract representation.

Observe that the scalar variables and the simple expressions semantics are the same as those defined by Cousot in [1].

4.1.1 Unidimensional char arrays semantics

We extend the previous concrete array representation in order to strengthen the *ad hoc* char array segmented depiction, presented in the Definition 4.2. First of all, we retain A, the fourth element of the a quadruple representing the concrete value of an array, defined in the sub-section 3.1.3, an invertible function.

Definition 4.4 (*invertibility*)**.** Given an array `A`, let I be the set of all the array indeces and let P be the set of all the (i, v) pairs, $A : \mathrm{I} \to \mathrm{P}$ is an invertible function if there exists a function $S : \mathrm{P} \to \mathrm{I}$ such that:

- $S(A[i]) = i, \ \forall i \in \mathrm{I}$
- $A(S((i, v))) = (i, v), \ \forall (i, v) \in \mathrm{P}$

If A is an invertible function, the function S is unique and it is the inverse function of A ($S = A^{-1}$).

$\Diamond$

From now on, we assume that the concrete value of a char array, $\mathtt{C}$, is represented by the quintuple $c = (\rho, \mathtt{C.low}, \mathtt{C.high}, C, \mathsf{T}) \in \text{ACVal}$ where:

- $\rho \in \mathcal{R}_v$ is a scalar variable environment;
- $\mathtt{C.low} \in \mathbb{E}$ is an expression which value $[\![\mathtt{C.low}]\!]\rho$ evaluated in the variable environment ρ yields the integer lower bound of the char array;
- $\mathtt{C.high} \in \mathbb{E}$ is an expression which value $[\![\mathtt{C.high}]\!]\rho$ evaluated in the variable environment ρ yields the integer upper bound of the char array;
- C maps an index $i \in \big[[\![\mathtt{C.low}]\!]\rho, [\![\mathtt{C.high}]\!]\rho\big)$ to a pair $C[i] = (i, v)$ of the index i and the corresponding char array element value v;
- T is the set of indices to which the string terminating character occurs. Formally: $\mathsf{T} = \{i \mid C[i] = (i, \text{`}\backslash 0\text{'}), \forall i \in \big[[\![\mathtt{C.low}]\!]\rho,$ $[\![\mathtt{C.high}]\!]\rho\big)\}$. This result is guaranteed by the definition 4.4.

The operational semantics of char array variables are concrete array environments $\sigma \in \mathcal{R}_c$ mapping char array names, $\mathtt{C} \in \mathbb{C}$, to their values $\sigma(\mathtt{C}) \in \text{ACVal} \triangleq \mathcal{R}_v \times \mathbb{E} \times \mathbb{E} \times (\mathbb{Z} \mapsto (\mathbb{Z} \times \text{ASCII})) \times \{\mathbb{Z}\}$, where ASCII is the considered characters domain, so that $\mathcal{R}_c \triangleq \mathbb{C} \mapsto \text{ACVal}$ and ACVal denotes the set of all the quintuples. Afterwards, given $\mathit{split}(\mathtt{C}) = \big(\mathtt{s}_{\mathtt{C}}, \mathtt{ns}_{\mathtt{C}}\big)$, depending on the content of T, we can incur in three different situations:

1. $\mathsf{T} = \emptyset$ (no string terminating character occurs in the char array).

   The $\mathtt{s}_{\mathtt{C}} = \emptyset$, as presented in the Definition 4.2.

   The $\mathtt{ns}_{\mathtt{C}}$ sub-segmentation is delimited by the parameters:

   $\mathtt{NS.low}_{\mathtt{C}}$
   - $\mathtt{NS.low}_{\mathtt{C}} \in \mathbb{E}$
   - $\mathtt{NS.low}_{\mathtt{C}}$ is the greatest lower bound of $\mathtt{ns}_{\mathtt{C}}$
   - $[\![\mathtt{NS.low}_{\mathtt{C}}]\!]\rho = [\![\mathtt{C.low}]\!]\rho$

   $\mathtt{NS.high}_{\mathtt{C}}$
   - $\mathtt{NS.high}_{\mathtt{C}} \in \mathbb{E}$
   - $\mathtt{NS.high}_{\mathtt{C}}$ is the least upper bound of $\mathtt{ns}_{\mathtt{C}}$

- $[\![\texttt{NS.high}_\texttt{C}]\!]\rho = [\![\texttt{C.high}]\!]\rho$

2. $\mathsf{T} = \{\texttt{C.high-1}\}$ (the char array just contains the string of interest).

   The $\texttt{s}_\texttt{C}$ sub-segmentation is delimited by the parameters:

   $\texttt{S.low}_\texttt{C}$

   - $\texttt{S.low}_\texttt{C} \in \mathbb{E}$
   - $\texttt{S.low}_\texttt{C}$ is the greatest lower bound of $\texttt{s}_\texttt{C}$
   - $[\![\texttt{S.low}_\texttt{C}]\!]\rho = [\![\texttt{C.low}]\!]\rho$

   $\texttt{S.high}_\texttt{C}$

   - $\texttt{S.high}_\texttt{C} \in \mathbb{E}$
   - $\texttt{S.high}_\texttt{C}$ is the least upper bound of $\texttt{s}_\texttt{C}$
   - $[\![\texttt{S.high}_\texttt{C}]\!]\rho = [\![\texttt{C.high-1}]\!]\rho$

   The $\texttt{ns}_\texttt{C} = \emptyset$, as presented in the Definition 4.2.

3. $\mathsf{T} \neq \{\texttt{C.high-1}\} \wedge \mathsf{T} \neq \emptyset$ (the dimension of the chat array is greater than the length of the string of interest).

   The $\texttt{s}_\texttt{C}$ sub-segmentation is delimited by the parameters:

   $\texttt{S.low}_\texttt{C}$

   - $\texttt{S.low}_\texttt{C} \in \mathbb{E}$
   - $\texttt{S.low}_\texttt{C}$ is the greatest lower bound of $\texttt{s}_\texttt{C}$
   - $[\![\texttt{S.low}_\texttt{C}]\!]\rho = [\![\texttt{C.low}]\!]\rho$

   $\texttt{S.high}_\texttt{C}$

   - $\texttt{S.low}_\texttt{C} \in \mathbb{E}$
   - $\texttt{S.low}_\texttt{C}$ is the least upper bound of $\texttt{s}_\texttt{C}$
   - $[\![\texttt{S.low}_\texttt{C}]\!]\rho = min(\mathsf{T})$

   The $\texttt{ns}_\texttt{C}$ sub-segmentation is delimited by the parameters:

   $\texttt{NS.low}_\texttt{C}$

   - $\texttt{NS.low}_\texttt{C} \in \mathbb{E}$
   - $\texttt{NS.low}_\texttt{C}$ is the greatest lower bound of $\texttt{ns}_\texttt{C}$
   - $[\![\texttt{NS.low}_\texttt{C}]\!]\rho = min(\mathsf{T}) + 1$

$\texttt{NS.high}_{\texttt{C}}$

- $\texttt{NS.high}_{\texttt{C}} \in \mathbb{E}$
- $\texttt{NS.high}_{\texttt{C}}$ is the least upper bound of $\texttt{ns}_{\texttt{C}}$
- $[\![\texttt{NS.high}_{\texttt{C}}]\!]\rho = [\![\texttt{C.high}]\!]\rho$

Definition 4.5 (*sub-segmentation*)**.** Let `str` be a char array, with concrete value represented by the quintuple $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$. $\texttt{str}[i, j]$ refers to the sub-segmentation of `str` that goes from the index $i \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)$ to the index $j \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)$.

◊

We assume that any sub-segmentation of the whole C-segmentation of a char array is represented by its own concrete value. As just declared, since the split segmentation components are, to all effects, sub-segmentations of the entire C-segmentation of a char array, we have that:

- $\texttt{s}_{\texttt{str}}$ ($\neq \emptyset$) corresponds to the sub-segmentation:

$$\texttt{str}[[\![\texttt{S.low}_{\texttt{str}}]\!]\rho, [\![\texttt{S.high}_{\texttt{str}}]\!]\rho - 1]$$

  with $\texttt{s}_{\texttt{str}} = (\rho, \texttt{S.low}_{\texttt{str}}, \texttt{S.high}_{\texttt{str}}, S, \mathsf{T}_{\texttt{s}_{\texttt{str}}})$.

  We expect that the set $\mathsf{T}_{\texttt{s}_{\texttt{str}}}$ is always empty.

- $\texttt{ns}_{\texttt{str}}$ ($\neq \emptyset$) corresponds to the sub-segmentation:

$$\texttt{str}[[\![\texttt{NS.low}_{\texttt{str}}]\!]\rho, [\![\texttt{NS.high}_{\texttt{str}}]\!]\rho - 1]$$

  with $\texttt{ns}_{\texttt{str}} = (\rho, \texttt{NS.low}_{\texttt{str}}, \texttt{NS.high}_{\texttt{str}}, NS, \mathsf{T}_{\texttt{ns}_{\texttt{str}}})$.

Example **4.3.** Given the char array $\texttt{C} = <\texttt{a a a a b b \textbackslash 0 c c c}>$, such that:

$c = (\rho, \texttt{C.low}, \texttt{C.high}, C, \mathsf{T}_{\texttt{C}})$

- $[\![\texttt{C.low}]\!]\rho = 0$
- $[\![\texttt{C.high}]\!]\rho = 10$
- $codom(C) = \{(0,\text{‘}\texttt{a}\text{’}), (1,\text{‘}\texttt{a}\text{’}), (2,\text{‘}\texttt{a}\text{’}), (3,\text{‘}\texttt{a}\text{’}), (4,\text{‘}\texttt{b}\text{’}), (5,\text{‘}\texttt{b}\text{’}),$
  $(6,\text{‘}\texttt{\textbackslash 0}\text{’}), (7,\text{‘}\texttt{c}\text{’}), (8,\text{‘}\texttt{c}\text{’}), (9,\text{‘}\texttt{c}\text{’})\}$

- $\mathtt{T_C} = \{6\}$

Here, $\mathit{split}(\mathtt{C}) = \big(\{\mathtt{0}\}\ `\mathtt{a}'\ \{\mathtt{4}\}\ `\mathtt{b}'\ \{\mathtt{6}\},\ \{\mathtt{7}\}\ `\mathtt{c}'\ \{\mathtt{10}\}\big)$. Recovering the Definition 4.5 we have that $\mathtt{s_C}$ is the sub-segmentation $\mathtt{C}[0,5]$ and its concrete value is represented by the quintuple:

$$s_{\mathtt{C}} = (\rho, \mathtt{0}, \mathtt{6}, s_c, \emptyset)$$
$$\mathit{codom}(s_c) = \{(0,`\mathtt{a}'), (1,`\mathtt{a}'), (2,`\mathtt{a}'), (3,`\mathtt{a}'), (4,`\mathtt{b}'), (5,`\mathtt{b}')\}$$

On the other end, $\mathtt{ns_C}$ is the sub-segmentation $\mathtt{C}[7,9]$ and its concrete value is represented by the quintuple:

$$ns_{\mathtt{C}} = (\rho, \mathtt{7}, \mathtt{10}, ns_c, \emptyset)$$
$$\mathit{codom}(ns_c) = \{(7,`\mathtt{c}'), (8,`\mathtt{c}'), (9,`\mathtt{c}')\}$$

Consider now the $\mathtt{C}$ sub-segmentation $\mathtt{C}[3,7]$: $\{\mathtt{3}\}\ `\mathtt{a}'\ \{\mathtt{4}\}\ `\mathtt{b}'\ \{\mathtt{6}\}\ `\backslash\mathtt{0}'\ \{\mathtt{7}\}\ `\mathtt{c}'\ \{\mathtt{8}\}$. Its concrete value is represented by the quintuple:

$$c_{[3,7]} = (\rho, \mathtt{3}, \mathtt{8}, C_{[3,7]}, \{6\})$$
$$\mathit{codom}(C_{[3,7]})\{(3,`\mathtt{a}'), (4,`\mathtt{b}'), (5,`\mathtt{b}'), (6,`\backslash\mathtt{0}'), (7,`\mathtt{c}')\}$$

and it is over-approximated by the split segmentation:

$\mathit{split}(\mathtt{C}[3,7]) = (\{\mathtt{3}\}\ `\mathtt{a}'\ \{\mathtt{4}\}\ `\mathtt{b}'\ \{\mathtt{6}\}, \{\mathtt{7}\}\ `\mathtt{c}'\ \{\mathtt{8}\})$ and:

- $s_{\mathtt{C}[3,7]} = (\rho, \mathtt{3}, \mathtt{6}, s_{c_{[3,7]}}, \emptyset)$
  $\mathit{codom}(s_{c_{[3,7]}}) = \{(3,`\mathtt{a}'), (4,`\mathtt{b}'), (5,`\mathtt{b}')\}$
- $ns_{\mathtt{C}[3,7]} = (\rho, \mathtt{7}, \mathtt{8}, ns_{c_{[3,7]}}, \emptyset)$
  $\mathit{codom}(ns_{c_{[3,7]}}) = \{(7,`\mathtt{c}')\}$

$\triangle$

We highlight the role of the environment $\rho \in \mathcal{R}_v$ in "translating" what a char array contains. Let $\mathtt{str}$ be a char array, with concrete value represented by the quintuple $\mathit{str} = (\rho, \mathtt{str.low}, \mathtt{str.high}, S, \mathtt{T_{str}})$. Every cell $\mathtt{str}[i]$, $\forall i \in [[\![\mathtt{str.low}]\!]\rho, [\![\mathtt{str.high}]\!]\rho)$, behaves as a char scalar variable representing the i-th array element. So, the concrete variable environment $\rho \in \mathcal{R}_v$ maps variable names $\mathtt{str}[i] \in \mathbb{C}$ to their values $\rho(\mathtt{str}[i]) \in \text{ASCII}$. In general, we are interested in the character depiction of the element represented by a char array cell (ASCII_{ch}) but, as we will see, there will be situations in which we will be interested in the constant numeric value of that element (ASCII_{cn}).

We also give a formal definition of what is a char array segment.

Definition 4.6 (*array segment*)**.** Let `str` be a char array. Given its C-segmentation, we consider a segment as a triple:

$$seg(\texttt{str})_i = (lb, x, ub)$$

- $i = 1, \ldots, n$ denotes the segment number (in the order that the segments appear in the C-segmentation)
- lb is the segment lower bound
- x is the character belonging to that segment (we assume that, less than further specifications, $\rho(\texttt{str}[k]) \in \text{ASCII}_{\text{ch}}$, $\forall k \in [lb, ub)$)
- ub is the segment upper bound

$\Diamond$

Example 4.4. Let `C`: $<$ `h e l l o \0 T T T` $>$ be a char array. Its C-segmentation is `{0}` ‘`h`’ `{1}` ‘`e`’ `{2}` ‘`l`’ `{4}` ‘`o`’ `{5}` ‘`\0`’ `{6}` `T` `{9}`. The C-segmentation of `C` has six segments, each of which characterized by its triple, as presented in the Definition 4.6. Let $Seg(\texttt{C})$ be the set of all the segments of the C-segmentation of `C`, its elements are:

- $seg(\texttt{C})_1 = (0, \text{‘}\texttt{h}\text{’}, 1)$
- $seg(\texttt{C})_2 = (1, \text{‘}\texttt{e}\text{’}, 2)$
- $seg(\texttt{C})_3 = (2, \text{‘}\texttt{l}\text{’}, 4)$
- $seg(\texttt{C})_4 = (4, \text{‘}\texttt{o}\text{’}, 5)$
- $seg(\texttt{C})_5 = (5, \text{‘}\texttt{\textbackslash 0}\text{’}, 6)$
- $seg(\texttt{C})_6 = (6, \texttt{T}, 9)$

Notice that, for example, the $seg(\texttt{C})_3 = (2, \texttt{'l'}, 4)$ contains the character `'l'` that corresponds to the element represented by the char array cells $[\![\texttt{C}[2]]\!]\rho \triangleq \rho(\texttt{C}[2]) = \texttt{'l'}$ and $[\![\texttt{C}[3]]\!]\rho \triangleq \rho(\texttt{C}[3]) = \texttt{'l'}$, in the ASCII character domain.

$\triangle$

Definition 4.7 (*join of segments*)**.** Let `str` be a char array and let $\dotplus$ be the join segments operator. Given `str` and its C-segmentation, we consider two successive segments of `str`, $seg(\texttt{str})_i = (lb, x, ub)$ and

$seg(\mathtt{str})_{i+1} = (lb, x, ub)$, such that $ub_{seg(\mathtt{str})_i} = lb_{seg(\mathtt{str})_{i+1}}$ then, if $x_{seg(\mathtt{str})_i} = x_{seg(\mathtt{str})_{i+1}}$, the join segments operator behaves as follows: $seg(\mathtt{str})_i \dotplus seg(\mathtt{str})_{i+1} = seg(\mathtt{str})_i = (lb_{seg(\mathtt{str})_i}, x_{seg(\mathtt{str})_i}, ub_{seg(\mathtt{str})_{i+1}})$. Otherwise, if $x_{seg(\mathtt{str})_i} \neq x_{seg(\mathtt{str})_{i+1}}$, the join between the two successive segments produces nothing (the result is the identity).

In the case in which it would be necessary to append one segmentation to another, the join segments operator can be very useful. As a matter of fact, given two char arrays, `str1` and `str2`, and their C-segmentations, let $Seg(\mathtt{str1})$ be the set of all the segments $seg(\mathtt{str1})_i = (lb, x, ub)$ of `str1`, for $i \in [1, n]$ and let $Seg(\mathtt{str2})$ be the set of all the segments $seg(\mathtt{str2})_j = (lb, x, ub)$ of `str2`, for $j \in [1, m]$. Appending one segmentation to another means perform the "unification" of the two set of segments by joining the last segment of `str1`, $seg(\mathtt{str1})_n$, with the first segment of `str2`, $seg(\mathtt{str2})_1$ and by properly modifying the `str2` segments bounds. Informally, we expand `str1`.

1. If $x_{seg(\mathtt{str1})_n} = x_{seg(\mathtt{str2})_1}$, then:

   $Seg(\mathtt{str1}) \cup Seg(\mathtt{str2}) = \{seg(\mathtt{str1})_i : i \in [1, n)\} \cup seg(\mathtt{str1})_n \dotplus seg(\mathtt{str2})_1 = seg(\mathtt{str1})_n = (lb_{seg(\mathtt{str1})_n}, x_{seg(\mathtt{str1})_n}, (ub_{seg(\mathtt{str1})_n} + (ub_{seg(\mathtt{str2})_1} - lb_{seg(\mathtt{str2})_1})) \cup \{seg(\mathtt{str1})_k = (ub_{seg(\mathtt{str1})_{k-1}}, x_{seg(\mathtt{str2})_j}, [ub_{seg(\mathtt{str1})_{k-1}} + (ub_{seg(\mathtt{str2})_j} - lb_{seg(\mathtt{str2})_j})]) : k \in (n, (n + m - 1)] \wedge j \in (1, m]\}$

2. If $x_{seg(\mathtt{str1})_n} \neq x_{seg(\mathtt{str2})_1}$, then:

   $Seg(\mathtt{str1}) \cup Seg(\mathtt{str2}) = \{seg(\mathtt{str1})_i : i \in [1, n]\} \cup \{seg(\mathtt{str1})_k = (ub_{seg(\mathtt{str1})_{k-1}}, x_{seg(\mathtt{str2})_j}, [ub_{seg(\mathtt{str1})_{k-1}} + (ub_{seg(\mathtt{str2})_j} - lb_{seg(\mathtt{str2})_j})]) : k \in (n, (n + m)] \wedge j \in [1, m]\}$

$\Diamond$

4.2 string.h functions semantics

In the rest of this section we formally introduce the concrete semantics of some of the string.h statements, capturing the impact of a statement with respect to the M-string abstract segmentation representation of array of char.

Definition 4.8. The concrete semantics:

$$\mathfrak{S} : \text{Stm} \times \text{ACState} \rightarrow \text{ACState} \uplus \mathbb{Z} \uplus \text{ACVal}$$

is a function that captures the effects of a statement.

- Stm denotes the set of string.h library functions.
- ACState denotes the set of σ functions mapping each char array type variable in the concrete representation of its C-segmentation, formally: $\text{ACState} = \{\sigma : \text{ACVar} \rightarrow \text{ACVal} \mid \text{ACVar} \in \text{Var(P)}\}$ where ACVar is the set of all the char array variables declared in a program and Var(P) represents the set of all the variables declared in a program.
- $\mathbb{Z}$ denotes, as usual, the set of integer numbers.
- ACVal denotes the set of all the quintuples, as presented in the subsection 4.1.1.
- $\uplus$ represents the disjoint union.

$\Diamond$

We cannot modify strings literals, but we can modify the char arrays holding strings. **string.h** is a header file that contains many functions for manipulating C strings, the commonly used functions are:

- **strcpy**(s_1, s_2) copies $\mathbf{s}_2$ in $\mathbf{s}_1$ and $\mathbf{s}_1$ is modified.
- **strcat**(s_1, s_2) concatenates $\mathbf{s}_2$ to the end of $\mathbf{s}_1$ and $\mathbf{s}_1$ is modified.
- **strlen**(s_1) returns the length of $\mathbf{s}_1$.
- **strchr**(s_1, ch) returns a pointer to the first occurrence of ch in s_1.
- **strcmp**(s_1, s_2) performs a lexicographic comparison between $\mathbf{s}_1$ and $\mathbf{s}_2$.

$$\textbf{strcmp}(s_1, s_2) = \begin{cases} 0 & \text{if } s_1 = s_2 \\ n < 0 & \text{if } s_1 < s_2 \\ n > 0 & \text{if } s_1 > s_2 \end{cases}$$

Definition 4.9 (*sub-segmentation semantics*)**.** Let `str` be a char array. We consider $\sigma[\texttt{str}](i, j)$ as the sub-segmentation representation of `str` from the segmentation lower bound i to the segmentation upper bound j, such that $i, j \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho) \wedge i < j$.

◊

4.2.1 The string copy statement

strcpy(dest,src) is a library function that allows null-terminated memory blocks to be copied from one location to another, including the null terminator. Since strings in C are not first-class data types and are implemented instead as contiguous blocks of bytes in memory, strcpy() will effectively copy strings given two pointers to blocks of allocated memory. The formal declaration for string copy function is:

char *strcpy(char *dest, const char *src)

dest is the pointer to the destination array in which the content is to be copied and src is the string to be copied. Implementation:

```
char *strcpy(char *dest, const char *src)
{
        char *ret = dest;
        while (*dest++ = *src++);
        return ret;
}
```

Note that source and destination may not overlap. The returned value is a pointer to the destination string dest; the function has no failure mode and no error return [13].

The analysis will be performed on static arrays. As a matter of fact, it is possible apply the string copy function on static arrays. Let s_1 and s_2 be two char arrays declared in a program, the parameters contained in the strcpy(s_1, s_2) correspond respectively to the memory address of the first cell of s_1 and to the memory address of the first cell of s_2. This permits to the string copy function to work on static arrays despite it originally requires pointers. Trivially, all the others functions presented in this section can be applied on static arrays.

The string copy original statement result - from the string.h header in the C standard library - may be misleading, or even erroneous, in the case in which one of the two char arrays function parameters, or both, had the string of interest equal to `undef`. Below, we stigmatize the behaviour of the string copy function, also the ambiguous one.

Definition 4.10 (*semantics of the string copy function*)**.** Let `str1` and `str2` be two char arrays declared in a program ($\mathtt{str1}, \mathtt{str2} \in$ ACVar), with concrete values respectively represented by the quintuples $str1 = (\rho, \mathtt{str1.low}, \mathtt{str1.high}, S_1, \mathsf{T}_{\mathtt{str1}})$ and $str2 = (\rho, \mathtt{str2.low}, \mathtt{str2.high}, S_2, \mathsf{T}_{\mathtt{str2}})$ ($str1$,$str2 \in$ ACVal), then:

$$\mathfrak{S}[\![\textbf{strcpy}(\mathtt{str1}, \mathtt{str2})]\!]\sigma = \sigma' \in \text{ACState}$$

1. $string(\mathtt{str1}) \neq \mathtt{undef} \wedge string(\mathtt{str2}) \neq \mathtt{undef}$

   a. If $|string(\mathtt{str2})| \leqslant |\mathtt{str1}|$:

   - $\sigma'(\mathtt{str1}) = (\rho, \mathtt{str1.low}', \mathtt{str1.high}', S'_1, \mathsf{T}'_{\mathtt{str1}})$

$$\mathtt{str1.low}' = \mathtt{str1.low}$$

$$\mathtt{str1.high}' = \mathtt{str1.high}$$

$$S'_1 : \begin{cases} \forall i \in [[\![\mathtt{str1.low}]\!]\rho, [\![\mathtt{str1.low}]\!]\rho + \\ \qquad |string(\mathtt{str2})|), \mathtt{str1'}[i] \to (i, v) \mid \\ \qquad\qquad (n, v) \in codom(S_2) \wedge \\ \qquad\qquad [\![\mathtt{S.low_{str2}}]\!]\rho \leqslant n \leqslant [\![\mathtt{S.high_{str2}}]\!]\rho \\ \forall i \in [[\![\mathtt{str1.low}]\!]\rho + |string(\mathtt{str2})|, \\ \qquad [\![\mathtt{str1.high}]\!]\rho), \mathtt{str1'}[i] \to (i, v) \mid \\ \qquad\qquad (m, v) \in codom(S_1) \wedge m = i \end{cases}$$

$$\mathsf{T}'_{\mathtt{str1}} = \{[\![\mathtt{str1.low}]\!]\rho + |string(\mathtt{str2})| - 1\} \cup \{i \in \mathsf{T}_{\mathtt{str1}} : i \geqslant [\![\mathtt{str1.low}]\!]\rho + |string(\mathtt{str2})| - 1\}$$

$$min(\mathsf{T}'_{\mathtt{str1}}) = [\![\mathtt{str1.low}]\!]\rho + |string(\mathtt{str2})| - 1$$

   - $\sigma'[\mathtt{s}_{\mathtt{str1}}] = \sigma[\mathtt{s}_{\mathtt{str2}}]$

   b. If $|string(\mathtt{str2})| > |\mathtt{str1}|$:

   - $\sigma'(\mathtt{str1}) = (\rho, \mathtt{str1.low}', \mathtt{str1.high}', S'_1, \mathsf{T}'_{\mathtt{str1}})$

$$\mathtt{str1.low}' = \mathtt{str1.low}$$

$$\mathtt{str1.high}' = \mathtt{str1.low} + |string(\mathtt{str2})|$$

$$\begin{aligned}&S'_1 : \forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.low}]\!]\rho + \\ &\quad |\mathit{string}(\texttt{str2})|), \texttt{str1'}[i] \rightarrow (i, v) \mid \\ &\qquad (n, v) \in \mathit{codom}(S_2) \ \wedge \\ &\qquad\quad [\![\texttt{S.low}_{\texttt{str2}}]\!]\rho \leqslant n \leqslant [\![\texttt{S.high}_{\texttt{str2}}]\!]\rho \\ &\mathsf{T'}_{\texttt{str1}} = \{[\![\texttt{str1.low}]\!]\rho + |\mathit{string}(\texttt{str2})| - 1\} \\ &\mathit{min}(\mathsf{T'}_{\texttt{str1}}) = [\![\texttt{str1.low}]\!]\rho + |\mathit{string}(\texttt{str2})| - 1\end{aligned}$$

- $\sigma'[\texttt{s}_{\texttt{str1}}] = \sigma[\texttt{s}_{\texttt{str2}}]$

2. $\mathit{string}(\texttt{str1}) = \texttt{undef} \wedge \mathit{string}(\texttt{str2}) \neq \texttt{undef}$

   `undefined behaviour`

   a. If $|\mathit{string}(\texttt{str2})| \leqslant |\texttt{str1}|$:

   - $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low}', \texttt{str1.high}', S'_1, \mathsf{T'}_{\texttt{str1}})$

$$\begin{aligned}&\texttt{str1.low}' = \texttt{str1.low} \\ &\texttt{str1.high}' = \texttt{str1.high} \\ &S'_1 : \begin{cases} \forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.low}]\!]\rho + \\ \quad |\mathit{string}(\texttt{str2})|), \texttt{str1'}[i] \rightarrow (i, v) \mid \\ \qquad (n, v) \in \mathit{codom}(S_2) \ \wedge \\ \qquad [\![\texttt{S.low}_{\texttt{str2}}]\!]\rho \leqslant n \leqslant [\![\texttt{S.high}_{\texttt{str2}}]\!]\rho \\ \forall i \in [[\![\texttt{str1.low}]\!]\rho + |\mathit{string}(\texttt{str2})|, \\ \quad [\![\texttt{str1.high}]\!]\rho), \texttt{str1'}[i] \rightarrow (i, v) \mid \\ \qquad (m, v) \in \mathit{codom}(S_1) \wedge m = i \end{cases} \\ &\mathsf{T'}_{\texttt{str1}} = \{[\![\texttt{str1.low}]\!]\rho + |\mathit{string}(\texttt{str2})| - 1\} \\ &\mathit{min}(\mathsf{T'}_{\texttt{str1}}) = [\![\texttt{str1.low}]\!]\rho + |\mathit{string}(\texttt{str2})| - 1\end{aligned}$$

   - $\sigma'[\texttt{s}_{\texttt{str1}}] = \sigma[\texttt{s}_{\texttt{str2}}]$

   b. If $|\mathit{string}(\texttt{str2})| > |\texttt{str1}|$:

   - $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low}', \texttt{str1.high}', S'_1, \mathsf{T'}_{\texttt{str1}})$

$$\begin{aligned}&\texttt{str1.low}' = \texttt{str1.low} \\ &\texttt{str1.high}' = \texttt{str1.low} + |\mathit{string}(\texttt{str2})| \\ &S'_1 : \forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.low}]\!]\rho + \\ &\quad |\mathit{string}(\texttt{str2})|), \texttt{str1'}[i] \rightarrow (i, v) \mid \\ &\qquad (n, v) \in \mathit{codom}(S_2) \ \wedge \\ &\qquad\quad [\![\texttt{S.low}_{\texttt{str2}}]\!]\rho \leqslant n \leqslant [\![\texttt{S.high}_{\texttt{str2}}]\!]\rho\end{aligned}$$

$$\mathsf{T'}_{\texttt{str1}} = \{[\![\texttt{str1.low}]\!]\rho + |\textit{string}(\texttt{str2})| - 1\}$$
$$\textit{min}(\mathsf{T'}_{\texttt{str1}}) = [\![\texttt{str1.low}]\!]\rho + |\textit{string}(\texttt{str2})| - 1$$

- $\sigma'[\mathsf{s}_{\texttt{str1}}] = \sigma[\mathsf{s}_{\texttt{str2}}]$

3. $\textit{string}(\texttt{str1}) \neq \texttt{undef} \wedge \textit{string}(\texttt{str2}) = \texttt{undef}$

   `undefined behaviour`

   In this case we assume that all the elements in `str2`, explicitly declared or not ($\top$), are copied in `str1`.

   a. If $|\texttt{str2}| < |\textit{string}(\texttt{str1})|$:

   - $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low}', \texttt{str1.high}', S'_1, \mathsf{T'}_{\texttt{str1}})$

$$\texttt{str1.low}' = \texttt{str1.low}$$
$$\texttt{str1.high}' = \texttt{str1.high}$$
$$S'_1 : \begin{cases} \forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.low}]\!]\rho + |\texttt{str2}|), \\ \qquad \texttt{str1'}[i] \to (i, v) \mid (n, v) \in \textit{codom}(S_2) \wedge \\ \qquad\qquad [\![\texttt{str2.low}]\!]\rho \leqslant n < [\![\texttt{str2.high}]\!]\rho \\ \forall i \in [[\![\texttt{str1.low}]\!]\rho + |\texttt{str2}|, [\![\texttt{str1.high}]\!]\rho), \\ \qquad \texttt{str1'}[i] \to (i, v) \mid (m, v) \in \textit{codom}(S_1) \\ \qquad\qquad \wedge\ m = i \end{cases}$$
$$\mathsf{T'}_{\texttt{str1}} = \mathsf{T}_{\texttt{str1}}$$
$$\textit{min}(\mathsf{T'}_{\texttt{str1}}) = \textit{min}(\mathsf{T}_{\texttt{str1}})$$

   - $\sigma'[\mathsf{s}_{\texttt{str1}}] = \sigma[\mathsf{ns}_{\texttt{str2}}] \dot{+} \sigma[\mathsf{s}_{\texttt{str1}}](i, j)$

     where $\sigma[\mathsf{s}_{\texttt{str1}}](i, j)$ denotes the sub-segmentation representation of $\mathsf{s}_{\texttt{str1}}$ from the sub-segmentation lower bound $i = [\![\texttt{S.low}_{\texttt{str1}}]\!]\rho + |\texttt{str2}|$ to the sub-segmentation upper bound $j = [\![\texttt{S.high}_{\texttt{str1}}]\!]\rho$, as presented in Definition 4.9.

   b. If $|\texttt{str2}| \geqslant |\textit{string}(\texttt{str1})| \wedge |\texttt{str2}| \leqslant |\texttt{str1}|$:

   - $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low}', \texttt{str1.high}', S'_1, \mathsf{T'}_{\texttt{str1}})$

$$\texttt{str1.low}' = \texttt{str1.low}$$
$$\texttt{str1.high}' = \texttt{str1.high}$$

$$S'_1 : \begin{cases} \forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.low}]\!]\rho + |\texttt{str2}|), \\ \qquad \texttt{str1'}[i] \to (i, v) \mid (n, v) \in \mathit{codom}(S_2) \wedge \\ \qquad\qquad [\![\texttt{str2.low}]\!]\rho \leqslant n < [\![\texttt{str2.high}]\!]\rho \\ \forall i \in [[\![\texttt{str1.low}]\!]\rho + |\texttt{str2}|, [\![\texttt{str1.high}]\!]\rho), \\ \qquad \texttt{str1'}[i] \to (i, v) \mid (m, v) \in \mathit{codom}(S_1) \\ \qquad\qquad \wedge\ m = i \end{cases}$$

$$\mathsf{T'}_{\texttt{str1}} = \{i \in \mathsf{T}_{\texttt{str1}} : i \geqslant [\![\texttt{str1.low}]\!]\rho + |\texttt{str2}|\}$$

- $\mathsf{T'}_{\texttt{str1}} = \emptyset \Rightarrow \sigma'[\texttt{s}_{\texttt{str1}}] = \emptyset$

  $\mathsf{T'}_{\texttt{str1}} \neq \emptyset \Rightarrow \sigma'[\texttt{s}_{\texttt{str1}}] = \sigma[\texttt{ns}_{\texttt{str2}}] \dotplus \sigma[\texttt{ns}_{\texttt{str1}}](i, j)$

  $i = [\![\texttt{S.low}_{\texttt{str1}}]\!]\rho + |\texttt{str2}|$
  $j = \mathit{min}\{k \mid \texttt{str1}[k] = \text{‘}\backslash 0\text{’} \wedge k \in [[\![\texttt{NS.low}_{\texttt{str1}}]\!]\rho,$
  $[\![\texttt{NS.high}_{\texttt{str1}}]\!]\rho) \wedge k \geqslant [\![\texttt{S.low}_{\texttt{str1}}]\!]\rho + |\texttt{str2}|\}$

c. If $|\texttt{str2}| \geqslant |\mathit{string}(\texttt{str1})| \wedge |\texttt{str2}| > |\texttt{str1}|$:

- $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low}', \texttt{str1.high}', S'_1, \mathsf{T'}_{\texttt{str1}})$

  $\texttt{str1.low}' = \texttt{str1.low}$

  $\texttt{str1.high}' = \texttt{str1.low} + |\texttt{str2}|$

  $S'_1 : \forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.low}]\!]\rho + |\texttt{str2}|),$
  $\texttt{str1'}[i] \to (i, v) \mid (n, v) \in \mathit{codom}(S_2) \wedge$
  $[\![\texttt{str2.low}]\!]\rho \leqslant n < [\![\texttt{str2.high}]\!]\rho$

  $\mathsf{T'}_{\texttt{str1}} = \emptyset$

  $\mathit{min}(\mathsf{T'}_{\texttt{str1}})$ does not exist

- $\sigma'[\texttt{s}_{\texttt{str1}}] = \emptyset$

4. $\mathit{string}(\texttt{str1}) = \texttt{undef} \wedge \mathit{string}(\texttt{str2}) = \texttt{undef}$

   `undefined behaviour`

   In this case we assume that all the elements in `str2`, explicitly declared or not ($\top$), are copied in `str1`.

   a. If $|\texttt{str2}| \leqslant |\texttt{str1}|$:

   - $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low}', \texttt{str1.high}', S'_1, \mathsf{T'}_{\texttt{str1}})$

     $\texttt{str1.low}' = \texttt{str1.low}$

$$\texttt{str1.high'} = \texttt{str1.high}$$

$$S'_1 : \begin{cases} \forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.low}]\!]\rho + |\texttt{str2}|), \\ \qquad \texttt{str1'}[i] \rightarrow (i, v) \mid (n, v) \in codom(S_2) \wedge \\ \qquad\qquad [\![\texttt{str2.low}]\!]\rho \leqslant n < [\![\texttt{str2.high}]\!]\rho \\ \forall i \in [[\![\texttt{str1.low}]\!]\rho + |\texttt{str2}|, [\![\texttt{str1.high}]\!]\rho), \\ \qquad \texttt{str1'}[i] \rightarrow (i, v) \mid (m, v) \in codom(S_1) \\ \qquad\qquad \wedge\ m = i \end{cases}$$

$$\mathsf{T'}_{\texttt{str1}} = \emptyset$$

$min(\mathsf{T'}_{\texttt{str1}})$ does not exist

- $\sigma'[\mathsf{s}_{\texttt{str1}}] = \emptyset$

b. If $|\texttt{str2}| > |\texttt{str1}|$:

- $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low'}, \texttt{str1.high'}, S'_1, \mathsf{T'}_{\texttt{str1}})$

$$\texttt{str1.low'} = \texttt{str1.low}$$

$$\texttt{str1.high'} = \texttt{str1.low} + |\texttt{str2}|$$

$$S'_1 : \forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.low}]\!]\rho + |\texttt{str2}|), \\ \texttt{str1'}[i] \rightarrow (i, v) \mid (n, v) \in codom(S_2) \wedge \\ [\![\texttt{str2.low}]\!]\rho \leqslant n < [\![\texttt{str2.high}]\!]\rho$$

$$\mathsf{T'}_{\texttt{str1}} = \emptyset$$

$min(\mathsf{T'}_{\texttt{str1}})$ does not exist

- $\sigma'[\mathsf{s}_{\texttt{str1}}] = \emptyset$

Notice that, in any case, the concrete value of `str2`, after the application of the string copy function on it, remain unchanged.

$\diamondsuit$

Example **4.5.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char dest[10] = ``aaaaaaa'';
/* 1: */ char src[5] = ``bb'';
/* 2: */ strcpy(dest,src);
/* 3: */ return 0;
/* 4: */ }
```

The concrete values of `dest` and `src` at the program point 2 are respectively given by the quintuple $dest = (\rho, \texttt{dest.low}, \texttt{dest.high}, D, \mathsf{T}_{\texttt{dest}})$ and the quintuple $src = (\rho, \texttt{src.low}, \texttt{src.high}, S, \mathsf{T}_{\texttt{src}})$. Entering into the detail, we have that:

- $dest = (\rho, \texttt{dest.low}, \texttt{dest.high}, D, \mathsf{T}_{\texttt{dest}})$

$$\begin{aligned}
&[\![\texttt{dest.low}]\!]\rho = 0\\
&[\![\texttt{dest.high}]\!]\rho = 10\\
&codom(D) = \{(0,\text{`}\texttt{a}\text{'}), (1,\text{`}\texttt{a}\text{'}), (2,\text{`}\texttt{a}\text{'}), (3,\text{`}\texttt{a}\text{'}), (4,\text{`}\texttt{a}\text{'}), (5,\text{`}\texttt{a}\text{'}),\\
&\qquad (6,\text{`}\texttt{a}\text{'}), (7,\text{`}\texttt{\textbackslash 0}\text{'}), (8,\texttt{T}), (9,\texttt{T})\}\\
&\mathsf{T}_{\texttt{dest}} = \{7\}\\
&min(\mathsf{T}_{\texttt{dest}}) = 7
\end{aligned}$$

$string(\texttt{dest}) = < \texttt{a a a a a a a \textbackslash 0} >$

- $src = (\rho, \texttt{src.low}, \texttt{src.high}, S, \mathsf{T}_{\texttt{src}})$

$$\begin{aligned}
&[\![\texttt{src.low}]\!]\rho = 0\\
&[\![\texttt{src.high}]\!]\rho = 5\\
&codom(S) = \{(0,\text{`}\texttt{b}\text{'}), (1,\text{`}\texttt{b}\text{'}), (2,\text{`}\texttt{\textbackslash 0}\text{'}), (3,\texttt{T}), (4,\texttt{T}), (5,\texttt{T})\}\\
&\mathsf{T}_{\texttt{src}} = \{2\}\\
&min(\mathsf{T}_{\texttt{src}}) = 2
\end{aligned}$$

$string(\texttt{src}) = < \texttt{b b \textbackslash 0} >$

At this point it is possible to compute the concrete value of the `strcpy(dest,src)` function result or, in other words, the concrete value of the char array `dest` at the program point 4. From the Definition 4.10 we have that $\mathfrak{S}[\![\textbf{strcpy(}\texttt{dest},\texttt{src}\textbf{)}]\!]\sigma = \sigma'$, where:

- $\sigma'(\texttt{dest}) = (\rho, \texttt{dest.low}', \texttt{dest.high}', D', \mathsf{T}'_{\texttt{dest}})$

$$\begin{aligned}
&[\![\texttt{dest.low}']\!]\rho = [\![\texttt{dest.low}]\!]\rho = 0\\
&[\![\texttt{dest.high}']\!]\rho = [\![\texttt{dest.high}]\!]\rho = 10\\
&codom(D') = \{(0,\text{`}\texttt{b}\text{'}), (1,\text{`}\texttt{b}\text{'}), (2,\text{`}\texttt{\textbackslash 0}\text{'}), (3,\text{`}\texttt{a}\text{'}), (4,\text{`}\texttt{a}\text{'}), (5,\text{`}\texttt{a}\text{'}),\\
&\qquad (6,\text{`}\texttt{a}\text{'}), (7,\text{`}\texttt{\textbackslash 0}\text{'}), (8,\texttt{T}), (9,\texttt{T})\}\\
&\mathsf{T}'_{\texttt{dest}} = \{2\} \cup \{7\} = \{2, 7\}
\end{aligned}$$

$$\min(\mathsf{T'}_{\texttt{dest}}) = 2 \ (= min(\mathsf{T}_{\texttt{src}}))$$

$string(\texttt{dest}) = < \texttt{b b \textbackslash 0} >$

- $\sigma'[\mathtt{s}_{\texttt{dest}}] = \sigma[\mathtt{s}_{\texttt{src}}]$

$$(\mathtt{s}_{\texttt{dest:p4}} = \{0\} \ \text{`}\texttt{b}\text{'} \ \{2\}) = (\mathtt{s}_{\texttt{src:p2}} = \{0\} \ \text{`}\texttt{b}\text{'} \ \{2\})$$

At the end of the analysis we have the certainty that the program array `dest` is a char array representing a string since $string(\texttt{dest:p4})$ exists and it is correct with respect to the string copy function result (case 1.a.) presented in the Definition 4.10. The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `dest:p1 = split(dest):` $(\{0\}$ ‘a’ $\{7\}$, $\{8\}$ T $\{10\})$
- `dest:p2 = split(dest):` $(\{0\}$ ‘a’ $\{7\}$, $\{8\}$ T $\{10\}) \wedge$ `src:p2 = split(src):` $(\{0\}$ ‘b’ $\{2\}$, $\{3\}$ T $\{5\})$
  Check($string$(`dest`)) = TRUE, Check($string$(`src`)) = TRUE
- `dest:p3 = split(dest):` $(\{0\}$ ‘b’ $\{2\}, \{3\}$ ‘a’ $\{7\}$ ‘\0’ $\{8\}$ T $\{10\}) \wedge$ `src:p3 = split(src):` $(\{0\}$ ‘b’ $\{2\}$, $\{3\}$ T $\{5\})$
- `dest:p4 = dest:p3` $\wedge$ `src:p4 = src:p3`

△

Example **4.6.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char dest[5] = ``aaaa'';
/* 1: */ char src[6] = ``bbbbb'';
/* 2: */ strcpy(dest,src);
/* 3: */ return 0;
/* 4: */ }
```

The concrete values of `dest` and `src` at the program point 2 are respectively given by the quintuple $dest = (\rho, \texttt{dest.low}, \texttt{dest.high}, D, \mathsf{T}_{\texttt{dest}})$ and the quintuple $src = (\rho, \texttt{src.low}, \texttt{src.high}, S, \mathsf{T}_{\texttt{src}})$. Entering into the detail, we have that:

- $dest = (\rho, \texttt{dest.low}, \texttt{dest.high}, D, \mathsf{T}_{\texttt{dest}})$

$$\begin{aligned}
&[\![\texttt{dest.low}]\!]\rho = 0\\
&[\![\texttt{dest.high}]\!]\rho = 5\\
&codom(D) = \{(0,\text{‘}\texttt{a}\text{’}), (1,\text{‘}\texttt{a}\text{’}), (2,\text{‘}\texttt{a}\text{’}), (3,\text{‘}\texttt{a}\text{’}), (4,\text{‘}\texttt{\textbackslash 0}\text{’})\}\\
&\mathsf{T}_{\texttt{dest}} = \{4\}\\
&min(\mathsf{T}_{\texttt{dest}}) = 4
\end{aligned}$$

$string(\texttt{dest}) = <\texttt{a a a a \textbackslash 0}>$

- $src = (\rho, \texttt{src.low}, \texttt{src.high}, S, \mathsf{T}_{\texttt{src}})$

$$\begin{aligned}
&[\![\texttt{src.low}]\!]\rho = 0\\
&[\![\texttt{src.high}]\!]\rho = 6\\
&codom(S) = \{(0,\text{‘}\texttt{b}\text{’}), (1,\text{‘}\texttt{b}\text{’}), (2,\text{‘}\texttt{b}\text{’}), (3,\text{‘}\texttt{b}\text{’}), (4,\text{‘}\texttt{b}\text{’}), (5,\text{‘}\texttt{\textbackslash 0}\text{’})\}\\
&\mathsf{T}_{\texttt{src}} = \{5\}\\
&min(\mathsf{T}_{\texttt{src}}) = 5
\end{aligned}$$

$string(\texttt{src}) = <\texttt{b b b b b \textbackslash 0}>$

The concrete value of the char array `dest` at the program point `4` is given by $\mathfrak{S}[\![\textbf{strcpy}(\texttt{dest}, \texttt{src})]\!]\sigma = \sigma'$, where:

- $\sigma'(\texttt{dest}) = (\rho, \texttt{dest.low}', \texttt{dest.high}', D', \mathsf{T}'_{\texttt{dest}})$

$$\begin{aligned}
&[\![\texttt{dest.low'}]\!]\rho = [\![\texttt{dest.low}]\!]\rho = 0\\
&[\![\texttt{dest.high'}]\!]\rho = [\![\texttt{dest.high}]\!]\rho + |string(\texttt{src})| = 6\\
&codom(D') = \{(0,\text{‘}\texttt{b}\text{’}), (1,\text{‘}\texttt{b}\text{’}), (2,\text{‘}\texttt{b}\text{’}), (3,\text{‘}\texttt{b}\text{’}), (4,\text{‘}\texttt{b}\text{’}), (5,\text{‘}\texttt{\textbackslash 0}\text{’})\}\\
&\mathsf{T}'_{\texttt{dest}} = \{5\}\\
&\text{min}(\mathsf{T}'_{\texttt{dest}}) = 5\ (= min(\mathsf{T}_{\texttt{src}}))
\end{aligned}$$

$string(\texttt{src}) = <\texttt{b b b b b \textbackslash 0}>$

- $\sigma'[\texttt{s}_{\texttt{dest}}] = \sigma[\texttt{s}_{\texttt{src}}]$

$$(\texttt{s}_{\texttt{dest:p4}} = \{0\}\ \text{‘}\texttt{b}\text{’}\ \{5\}) = (\texttt{s}_{\texttt{src:p2}} = \{0\}\ \text{‘}\texttt{b}\text{’}\ \{5\})$$

At the end of the analysis we have the certainty that the program array `dest` is a char array representing a string since $\mathit{string}(\texttt{dest:p4})$ exists and it is correct with respect to the string copy function result (case 1.b.) presented in the Definition 4.10. The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `dest:p1 = split(dest):` $(\{0\}$ `'a'` $\{4\}, \emptyset)$
- `dest:p2 = split(dest):` $(\{0\}$ `'a'` $\{4\}, \emptyset) \wedge$
  `src:p2 = split(src):` $(\{0\}$ `'b'` $\{5\}, \emptyset)$
  $\mathrm{Check}(\mathit{string}(\texttt{dest})) = \mathrm{TRUE}$, $\mathrm{Check}(\mathit{string}(\texttt{src})) = \mathrm{TRUE}$
- `dest:p3 = split(dest):` $(\{0\}$ `'b'` $\{5\}, \emptyset) \wedge$
  `src:p3 = split(src):` $(\{0\}$ `'b'` $\{5\}, \emptyset)$
- `dest:p4 = dest:p3` $\wedge$ `src:p4 = src:p3`

$\triangle$

Example **4.7.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char dest[5] = ``aaaa'';
/* 1: */ char src[7] = ``bbbbbbb'';
/* 2: */ strcpy(dest,src);
/* 3: */ return 0;
/* 4: */ }
```

The concrete values of `dest` and `src` at the program point 2 are respectively given by the quintuple $\mathit{dest} = (\rho, \texttt{dest.low}, \texttt{dest.high}, D, \mathsf{T}_{\texttt{dest}})$ and the quintuple $\mathit{src} = (\rho, \texttt{src.low}, \texttt{src.high}, S, \mathsf{T}_{\texttt{src}})$. Entering into the detail, we have that:

- $\mathit{dest} = (\rho, \texttt{dest.low}, \texttt{dest.high}, D, \mathsf{T}_{\texttt{dest}})$

$$[\![\texttt{dest.low}]\!]\rho = 0$$
$$[\![\texttt{dest.high}]\!]\rho = 5$$
$$\mathit{codom}(D) = \{(0,\text{`a'}), (1,\text{`a'}), (2,\text{`a'}), (3,\text{`a'}), (4,\text{`\textbackslash 0'})\}$$

$$\mathsf{T}_{\texttt{dest}} = \{4\}$$
$$min(\mathsf{T}_{\texttt{dest}}) = 4$$

$string(\texttt{dest}) = <\texttt{a a a a} \backslash 0>$

- $src = (\rho, \texttt{src.low}, \texttt{src.high}, S, \mathsf{T}_{\texttt{src}})$

$$[\![\texttt{src.low}]\!]\rho = 0$$
$$[\![\texttt{src.high}]\!]\rho = 7$$
$$codom(S) = \{(0,\text{`}\texttt{b}\text{'}), (1,\text{`}\texttt{b}\text{'}), (2,\text{`}\texttt{b}\text{'}), (3,\text{`}\texttt{b}\text{'}), (4,\text{`}\texttt{b}\text{'}), (5,\text{`}\texttt{b}\text{'}), (6,\text{`}\texttt{b}\text{'})\}$$
$$\mathsf{T}_{\texttt{src}} = \emptyset$$
$min(\mathsf{T}_{\texttt{src}})$ does not exist

$string(\texttt{src}) = \texttt{undef}$

The concrete value of the char array `dest` at the program point `4` is given by $\mathfrak{S}[\![\textbf{strcpy}(\texttt{dest}, \texttt{src})]\!]\sigma = \sigma'$, where:

- $\sigma'(\texttt{dest}) = (\rho, \texttt{dest.low}', \texttt{dest.high}', D', \mathsf{T}'_{\texttt{dest}})$

$$[\![\texttt{dest.low}']\!]\rho = [\![\texttt{dest.low}]\!]\rho = 0$$
$$[\![\texttt{dest.high}']\!]\rho = [\![\texttt{dest.high}]\!]\rho + |src| = 7$$
$$codom(D') = \{(0,\text{`}\texttt{b}\text{'}), (1,\text{`}\texttt{b}\text{'}), (2,\text{`}\texttt{b}\text{'}), (3,\text{`}\texttt{b}\text{'}), (4,\text{`}\texttt{b}\text{'}), (5,\text{`}\texttt{b}\text{'}), (6,\text{`}\texttt{b}\text{'})\}$$
$$\mathsf{T}'_{\texttt{dest}} = \emptyset$$
$\min(\mathsf{T}'_{\texttt{dest}})$ does not exists

$string(\texttt{dest}) = \texttt{undef}$

- $\sigma'[\mathsf{s}_{\texttt{dest}}] = \emptyset$

At the end of the analysis we have the certainty that the program array `dest` is a char array that does not contain a string since $string(\texttt{dest:p4})$ does not exist, $\mathsf{T}'_{\texttt{dest}} = \emptyset$ and, of course, it has not a minimum element, as presented in the Definition 4.10 (case 3.c.). The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `dest:p1 = split(dest):` $\big(\{0\}\ \text{`a'}\ \{4\}, \emptyset\big)$

- `dest:p2 = split(dest):` $\big(\{0\}$ `‘a’` $\{4\}, \emptyset\big) \wedge$
  `src:p2 = split(src):` $\big(\emptyset, \{0\}$ `‘b’` $\{7\}\big)$
  Check($string$(`dest`)) = TRUE, Check($string$(`src`)) = FALSE
- `dest:p3 = split(dest):` $\big(\emptyset, \{0\}$ `‘b’` $\{7\}\big) \wedge$
  `src:p3 = split(src):` $\big(\emptyset, \{0\}$ `‘b’` $\{7\}\big)$
- `dest:p4 = dest:p3` $\wedge$ `src:p4 = src:p3`

$\triangle$

4.2.2 The string concatenation statement

strcat(dest,src) is a library function that allows one memory block to be appended to another memory block. Both memory blocks are required to be null-terminated. Since, in C, strings are not first-class datatypes, and are implemented as blocks of ASCII bytes in memory, strcat() will effectively append one string to another given two pointers to blocks of allocated memory. The formal declaration for string concatenation function is:

char *strcat(char *dest, const char *src)

dest is the pointer to the destination array, which should contain a C string, and should be large enough to contain the concatenated resulting string and src is the string to be appended. This should not overlap the destination. Implementation:

```
char *strcat(char *dest, const char *src)
{
        char *ret = dest;
        while (*dest)
           dest++;
        while (*dest++ = *src++);
        return ret;
}
```

The returned value is the pointer dest; the function has no failure mode and no error return [13]. As before, the analysis will be performed on static array.

The string concatenation original statement result - from the string.h header in the C standard library - may be misleading, or even erroneous, in the case in which one of the two char arrays function parameters, or both, had the string of interest equal to `undef`. Below, we stigmatize the behaviour of the string concatenation function, also the ambiguous one.

Definition 4.11 (*semantics of the string concatenation function*)**.** Let `str1` and `str2` be two char arrays declared in a program ($\mathtt{str1}, \mathtt{str2} \in$ ACVar), with concrete values respectively represented by the quintuples $str1 = (\rho, \mathtt{str1.low}, \mathtt{str1.high}, S_1, \mathsf{T}_{\mathtt{str1}})$ and $str2 = (\rho, \mathtt{str2.low}, \mathtt{str2.high}, S_2, \mathsf{T}_{\mathtt{str2}})$ ($str1, str2 \in$ ACVal), then:

$$\mathfrak{S}[\![\mathbf{strcat(}\mathtt{str1}, \mathtt{str2}\mathbf{)}]\!]\sigma = \sigma' \in \text{ACState}$$

1. $string(\mathtt{str1}) \neq \mathtt{undef} \wedge string(\mathtt{str2}) \neq \mathtt{undef}$

   a. If $|string(\mathtt{str1})| + |string(\mathtt{str2})| - 1 \leqslant |\mathtt{str1}|$:

   - $\sigma'(\mathtt{str1}) = (\rho, \mathtt{str1.low}', \mathtt{str1.high}', S'_1, \mathsf{T}'_{\mathtt{str1}})$

$$\mathtt{str1.low}' = \mathtt{str1.low}$$
$$\mathtt{str1.high}' = \mathtt{str1.high}$$
$$S'_1 : \begin{cases} \forall i \in [[\![\mathtt{str1.low}]\!]\rho, min(\mathsf{T}_{\mathtt{str1}})) \\ \qquad \mathtt{str1'}[i] \to (i, v) \mid (m, v) \in codom(S_1) \\ \qquad\qquad \wedge\ m = i \\ \forall i \in [min(\mathsf{T}_{\mathtt{str1}}), min(\mathsf{T}_{\mathtt{str1}})+ \\ \qquad |string(\mathtt{str2})|), \mathtt{str1'}[i] \to (i, v) \mid \\ \qquad\qquad (n, v) \in codom(S_2)\ \wedge \\ \qquad\qquad [\![\mathtt{S.low}_{\mathtt{str2}}]\!]\rho \leqslant n \leqslant [\![\mathtt{S.high}_{\mathtt{str2}}]\!]\rho \\ \forall i \in [min(\mathsf{T}_{\mathtt{str1}}) + |string(\mathtt{str2})|, \\ \qquad [\![\mathtt{str1.high}]\!]\rho), \mathtt{str1'}[i] \to (i, v) \mid \\ \qquad\qquad (m, v) \in codom(S_1) \wedge m = i \end{cases}$$
$$\mathsf{T}' = \{min(\mathsf{T}_{\mathtt{str1}}) + |string(\mathtt{str2})| - 1\} \cup \{i \in \mathsf{T}_{\mathtt{str1}} : i \geqslant min(\mathsf{T}_{\mathtt{str1}}) + |string(\mathtt{str2})| - 1\}$$
$$min(\mathsf{T}'_{\mathtt{str1}}) = min(\mathsf{T}_{\mathtt{str1}}) + |string(\mathtt{str2})| - 1$$

   - $\sigma'[\mathtt{s}_{\mathtt{str1}}] = \sigma[\mathtt{s}_{\mathtt{str1}}] \dot{+} \sigma[\mathtt{s}_{\mathtt{str2}}]$

b. If $|string(\texttt{str1})| + |string(\texttt{str2})| - 1 > |\texttt{str1}|$:

- $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low}', \texttt{str1.high}', S'_1, \mathsf{T}'_{\texttt{str1}})$

$$\begin{aligned}
&\texttt{str1.low}' = \texttt{str1.low}\\
&\texttt{str1.high}' = \texttt{str1.low} \;+\; |string(\texttt{str1})| \;+\\
&\qquad\qquad |string(\texttt{str2})| \;-\; 1\\
&S'_1 : \begin{cases}
\forall i \in [[\![\texttt{str1.low}]\!]\rho, min(\mathsf{T}_{\texttt{str1}}))\\
\qquad \texttt{str1'}[i] \to (i,v) \mid (m,v) \in codom(S_1)\\
\qquad\qquad \wedge\ m = i\\
\forall i \in [min(\mathsf{T}_{\texttt{str1}}), [\![\texttt{str1.low}]\!]\rho\ +\\
\qquad |string(\texttt{str1})| + |string(\texttt{str2})| - 1),\\
\qquad \texttt{str1'}[i] \to (i,v) \mid (n,v) \in codom(S_2)\ \wedge\\
\qquad\qquad [\![\texttt{S.low}_{\texttt{str2}}]\!]\rho \leqslant n \leqslant [\![\texttt{S.high}_{\texttt{str2}}]\!]\rho
\end{cases}\\
&\mathsf{T}'_{\texttt{str1}} = \{min(\mathsf{T}_{\texttt{str1}}) + |string(\texttt{str2})| - 1\}\\
&min(\mathsf{T}'_{\texttt{str1}}) = min(\mathsf{T}_{\texttt{str1}}) + |string(\texttt{str2})| - 1
\end{aligned}$$

- $\sigma'[\mathsf{s}_{\texttt{str1}}] = \sigma[\mathsf{s}_{\texttt{str1}}] \dotplus \sigma[\mathsf{s}_{\texttt{str2}}]$

2. $string(\texttt{str1}) = \texttt{undef} \wedge string(\texttt{str2}) \neq \texttt{undef}$

   `undefined behaviour`

   In this case we assume that the string of interest of `str2` is concatenated to `str1`.

   - $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low}', \texttt{str1.high}', S'_1, \mathsf{T}'_{\texttt{str1}})$

$$\begin{aligned}
&\texttt{str1.low}' = \texttt{str1.low}\\
&\texttt{str1.high}' = \texttt{str1.high} + |string(\texttt{str2})|\\
&S'_1 : \begin{cases}
\forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.high}]\!]\rho)\\
\qquad \texttt{str1'}[i] \to (i,v) \mid (m,v) \in codom(S_1)\\
\qquad\qquad \wedge\ m = i\\
\forall i \in [[\![\texttt{str1.high}]\!]\rho, [\![\texttt{str1.high}]\!]\rho\ +\\
\qquad |string(\texttt{str2})|), \texttt{str1'}[i] \to (i,v) \mid\\
\qquad\qquad (n,v) \in codom(S_2)\ \wedge\\
\qquad\qquad [\![\texttt{S.low}_{\texttt{str2}}]\!]\rho \leqslant n \leqslant [\![\texttt{S.high}_{\texttt{str2}}]\!]\rho
\end{cases}
\end{aligned}$$

$$\mathsf{T'}_{\mathtt{str1}} = \{[\![\mathtt{str1.high'}]\!]\rho - 1\}$$
$$min(\mathsf{T'}_{\mathtt{str1}}) = [\![\mathtt{str1.high'}]\!]\rho - 1$$

- $\sigma'[\mathsf{s}_{\mathtt{str1}}] = \sigma[\mathtt{ns}_{\mathtt{str1}}] \dotplus \sigma[\mathsf{s}_{\mathtt{str2}}]$

3. $string(\mathtt{str1}) \neq \mathtt{undef} \wedge string(\mathtt{str2}) = \mathtt{undef}$

   `undefined behaviour`

   In this case we assume that `str2` is concatenated to the string of interest of `str1`.

   a. If $|string(\mathtt{str1})| + |\mathtt{str2}| - 1 \leqslant |\mathtt{str1}|$

   - $\sigma'(\mathtt{str1}) = (\rho, \mathtt{str1.low'}, \mathtt{str1.high'}, S'_1, \mathsf{T'}_{\mathtt{str1}})$

     $\mathtt{str1.low'} = \mathtt{str1.low}$
     $\mathtt{str1.high'} = \mathtt{str1.high}$

$$S'_1 : \begin{cases} \forall i \in [[\![\mathtt{str1.low}]\!]\rho, min(\mathsf{T}_{\mathtt{str1}})) \\ \qquad \mathtt{str1'}[i] \to (i, v) \mid (m, v) \in codom(S_1) \\ \qquad\qquad \wedge\ m = i \\ \forall i \in [min(\mathsf{T}_{\mathtt{str1}}), min(\mathsf{T}_{\mathtt{str1}}) + |\mathtt{str2}|), \\ \qquad \mathtt{str1'}[i] \to (i, v) \mid (n, v) \in codom(S_2) \wedge \\ \qquad\qquad [\![\mathtt{str2.low}]\!]\rho \leqslant n < [\![\mathtt{str2.high}]\!]\rho \\ \forall i \in [min(\mathsf{T}_{\mathtt{str1}}) + |\mathtt{str2}|, [\![\mathtt{str1.high}]\!]\rho), \\ \qquad \mathtt{str1'}[i] \to (i, v) \mid (m, v) \in codom(S_1) \\ \qquad\qquad \wedge\ m = i \end{cases}$$

$$\mathsf{T'}_{\mathtt{str1}} = \{i \in \mathsf{T}_{\mathtt{str1}} : i \geqslant min(\mathsf{T}_{\mathtt{str1}}) + |\mathtt{str2}|\}$$

   - $\mathsf{T'}_{\mathtt{str1}} = \emptyset \Rightarrow \sigma'[\mathsf{s}_{\mathtt{str1}}] = \emptyset$

     $\mathsf{T'}_{\mathtt{str1}} \neq \emptyset \Rightarrow \sigma'[\mathsf{s}_{\mathtt{str1}}] = \sigma[\mathsf{s}_{\mathtt{str1}}] \dotplus \sigma[\mathtt{ns}_{\mathtt{str2}}] \dotplus \sigma[\mathtt{ns}_{\mathtt{str1}}](i, j)$

     $i = min(\mathsf{T}_{\mathtt{str1}}) + |\mathtt{str2}|$
     $j = min\{k \mid \mathtt{str1}[k] = \text{‘}\backslash 0\text{’} \wedge k \in [[\![\mathtt{NS.low}_{\mathtt{str1}}]\!]\rho, [\![\mathtt{NS.high}_{\mathtt{str1}}]\!]\rho) \wedge k \geqslant [\![\mathtt{S.low}_{\mathtt{str1}}]\!]\rho + |\mathtt{str2}|\}$

   b. If $|string(\mathtt{str1})| + |\mathtt{str2}| - 1 > |\mathtt{str1}|$

   - $\sigma'(\mathtt{str1}) = (\rho, \mathtt{str1.low'}, \mathtt{str1.high'}, S'_1, \mathsf{T'}_{\mathtt{str1}})$

$$\texttt{str1.low'} = \texttt{str1.low}$$
$$\texttt{str1.high'} = \texttt{S.high}_{\texttt{str1}} + |\texttt{str2}|$$
$$S'_1 : \begin{cases} \forall i \in [[\![\texttt{str1.low}]\!]\rho, min(\mathsf{T}_{\texttt{str1}})) \\ \qquad \texttt{str1'}[i] \rightarrow (i,v) \mid (m,v) \in codom(S_1) \\ \qquad\qquad \wedge\ m = i \\ \forall i \in [min(\mathsf{T}_{\texttt{str1}}), min(\mathsf{T}_{\texttt{str1}}) + |\texttt{str2}|), \\ \qquad \texttt{str1'}[i] \rightarrow (i,v) \mid (n,v) \in codom(S_2)\ \wedge \\ \qquad\qquad [\![\texttt{str2.low}]\!]\rho \leqslant n < [\![\texttt{str2.high}]\!]\rho \end{cases}$$
$$\mathsf{T'}_{\texttt{str1}} = \emptyset$$
$min(\mathsf{T'}_{\texttt{str1}})$ does not exist

- $\sigma'[\mathsf{s}_{\texttt{str1}}] = \emptyset$

4. $string(\texttt{str1}) = \texttt{undef} \wedge string(\texttt{str2}) = \texttt{undef}$

`undefined behaviour`

In this case we assume that `str2` is concatenated to `str1`.

- $\sigma'(\texttt{str1}) = (\rho, \texttt{str1.low'}, \texttt{str1.high'}, S'_1, \mathsf{T'}_{\texttt{str1}})$

$$\texttt{str1.low'} = \texttt{str1.low}$$
$$\texttt{str1.high'} = \texttt{str1.high} + |\texttt{str2}|$$
$$S'_1 : \begin{cases} \forall i \in [[\![\texttt{str1.low}]\!]\rho, [\![\texttt{str1.high}]\!]\rho) \\ \qquad \texttt{str1'}[i] \rightarrow (i,v) \mid (m,v) \in codom(S_1) \\ \qquad\qquad \wedge\ m = i \\ \forall i \in [[\![\texttt{str1.high}]\!]\rho, [\![\texttt{str1.high}]\!]\rho + |\texttt{str2}|), \\ \qquad \texttt{str1'}[i] \rightarrow (i,v) \mid (n,v) \in codom(S_2)\ \wedge \\ \qquad\qquad [\![\texttt{str2.low}]\!]\rho \leqslant n < [\![\texttt{str2.high}]\!]\rho \end{cases}$$
$$\mathsf{T'}_{\texttt{str1}} = \emptyset$$
$min(\mathsf{T'}_{\texttt{str1}})$ does not exist

- $\sigma'[\mathsf{s}_{\texttt{str1}}] = \emptyset$

Notice that, in any case, the concrete value of `str2`, after the application of the string concatenation function on it, remain unchanged.

$\Diamond$

Example **4.8.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char dest[9] = ''aaaa'';
/* 1: */ char src[5] = ''bbbb'';
/* 2: */ strcat(dest,src);
/* 3: */ return 0;
/* 4: */ }
```

The concrete values of `dest` and `src` at the program point 2 are respectively given by the quintuple $dest = (\rho, \texttt{dest.low}, \texttt{dest.high}, D, \mathsf{T}_{\texttt{dest}})$ and the quintuple $src = (\rho, \texttt{src.low}, \texttt{src.high}, S, \mathsf{T}_{\texttt{src}})$. Entering into the detail, we have that:

- $dest = (\rho, \texttt{dest.low}, \texttt{dest.high}, D, \mathsf{T}_{\texttt{dest}})$

  $$\begin{aligned}
  &[\![\texttt{dest.low}]\!]\rho = 0\\
  &[\![\texttt{dest.high}]\!]\rho = 9\\
  &codom(D) = \{(0,\text{‘}\texttt{a}\text{’}), (1,\text{‘}\texttt{a}\text{’}), (2,\text{‘}\texttt{a}\text{’}), (3,\text{‘}\texttt{a}\text{’}), (4,\text{‘}\texttt{\textbackslash 0}\text{’}), (5,\texttt{T}),\\
  &\qquad\qquad (6,\texttt{T}), (7,\texttt{T}), (8,\texttt{T})\}\\
  &\mathsf{T}_{\texttt{dest}} = \{4\}\\
  &min(\mathsf{T}_{\texttt{dest}}) = 4
  \end{aligned}$$

  $string(\texttt{dest}) = <\texttt{a a a a \textbackslash 0}>$

- $src = (\rho, \texttt{src.low}, \texttt{src.high}, S, \mathsf{T}_{\texttt{src}})$

  $$\begin{aligned}
  &[\![\texttt{src.low}]\!]\rho = 0\\
  &[\![\texttt{src.high}]\!]\rho = 5\\
  &codom(S) = \{(0,\text{‘}\texttt{b}\text{’}), (1,\text{‘}\texttt{b}\text{’}), (2,\text{‘}\texttt{b}\text{’}), (3,\text{‘}\texttt{b}\text{’}), (4,\text{‘}\texttt{\textbackslash 0}\text{’})\}\\
  &\mathsf{T}_{\texttt{src}} = \{4\}\\
  &min(\mathsf{T}_{\texttt{src}}) = 4
  \end{aligned}$$

  $string(\texttt{src}) = <\texttt{b b b b \textbackslash 0}>$

At this point it is possible to compute the concrete value of the `strcat(dest,src)` function result or, in other words, the concrete value of the char array `dest` at the program point 4. From the Definition 4.11 we have that $\mathfrak{S}[\![\textbf{strcat}(\texttt{dest}, \texttt{src})]\!]\sigma = \sigma'$, where:

- $\sigma'(\texttt{dest}) = (\rho, \texttt{dest.low'}, \texttt{dest.high'}, D', \mathsf{T}')$

$$\begin{aligned}
&[\![\texttt{dest.low'}]\!]\rho = [\![\texttt{dest.low}]\!]\rho = 0\\
&[\![\texttt{dest.high'}]\!]\rho = [\![\texttt{dest.high}]\!]\rho = 9\\
&codom(D') = \{(0,\text{‘a’}), (1,\text{‘a’}), (2,\text{‘a’}), (3,\text{‘a’}), (4,\text{‘b’}), (5,\text{‘b’}),\\
&\qquad\qquad (6,\text{‘b’}), (7,\text{‘b’}), (8,\text{‘\textbackslash 0’})\}\\
&\mathsf{T}' = \{8\} \cup \emptyset = \{8\}\\
&\min(\mathsf{T}') = min(\mathsf{T}_{\texttt{dest}}) + |string(\texttt{src})| - 1 = 4 + 5 - 1 = 8
\end{aligned}$$

$string(\texttt{dest}) = <\texttt{a a a a b b b b \textbackslash 0}>$

- $\sigma'[\mathtt{s}_{\texttt{dest}}] = \sigma[\mathtt{s}_{\texttt{dest}}] \dotplus \sigma[\mathtt{s}_{\texttt{src}}]$

$$\begin{aligned}
\mathtt{s}_{\texttt{dest:p4}} &= (\mathtt{s}_{\texttt{dest:p2}} = \{0\}\ \text{‘a’}\ \{4\}) \ \dotplus\ (\mathtt{s}_{\texttt{src:p2}} = \{0\}\ \text{‘b’}\ \{4\})\\
&= \{0\}\ \text{‘a’}\ \{4\}\ \text{‘b’}\ \{8\}
\end{aligned}$$

In this case $string(\texttt{dest:p4})$ exists and it is correct with respect to the string concatenation function result presented in the Definition 4.11 (case 1.a.). The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `dest:p1 = split(dest):` $(\{0\}\ \text{‘a’}\ \{4\},\ \{5\}\ \mathtt{T}\ \{9\})$
- `dest:p2 = split(dest):` $(\{0\}\ \text{‘a’}\ \{4\},\ \{5\}\ \mathtt{T}\ \{9\}) \wedge$
  `src:p2 = split(src):` $(\{0\}\ \text{‘b’}\ \{4\}, \emptyset)$
  Check($string$(`dest`)) = TRUE, Check($string$(`src`)) = TRUE
- `dest:p3 = split(dest):` $(\{0\}\ \text{‘a’}\ \{4\}\ \text{‘b’}\ \{8\}, \emptyset) \wedge$
  `src:p3 = split(src):` $(\{0\}\ \text{‘b’}\ \{4\}, \emptyset)$
- `dest:p4 = dest:p3` $\wedge$ `src:p4 = src:p3`

△

EXAMPLE **4.9.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char dest[8] = ''aaaa'';
/* 1: */ char src[5] = ''bbbbb'';
/* 2: */ strcat(dest,src);
/* 3: */ return 0;
/* 4: */ }
```

The concrete value of the char array `dest` at the program point 4 is given by $\mathfrak{S}[\![\mathbf{strcat}(\mathtt{dest},\mathtt{src})]\!]\sigma = \sigma'$, where:

- $\sigma'(\mathtt{dest}) = (\rho, \mathtt{dest.low'}, \mathtt{dest.high'}, D', \mathsf{T}')$

  $$[\![\mathtt{dest.low'}]\!]\rho = [\![\mathtt{dest.low}]\!]\rho = 0$$
  $$[\![\mathtt{dest.high'}]\!]\rho = [\![\mathtt{S.high}_{\mathtt{dest}}]\!]\rho + |\mathtt{src}| = 9$$
  $$\begin{aligned} codom(D)' = \{&(0,\text{`}\mathtt{a}\text{'}), (1,\text{`}\mathtt{a}\text{'}), (2,\text{`}\mathtt{a}\text{'}), (3,\text{`}\mathtt{a}\text{'}), (4,\text{`}\mathtt{b}\text{'}), (5,\text{`}\mathtt{b}\text{'}),\\ &(6,\text{`}\mathtt{b}\text{'}), (7,\text{`}\mathtt{b}\text{'}), (8,\text{`}\mathtt{b}\text{'})\}\end{aligned}$$
  $$\mathsf{T}' = \emptyset$$
  $\min(\mathsf{T}')$ does not exist.

  $string(\mathtt{dest}) = \mathtt{undef}$

- $\sigma'[\mathtt{s}_{\mathtt{dest}}] = \emptyset$

At the end of the analysis we have the certainty that the program array `dest` is a char array that does not contain a string since $string(\mathtt{dest:p4})$ does not exist, $\mathsf{T}'_{\mathtt{dest}} = \emptyset$ and, of course, it has not a minimum element, as presented in the Definition 4.11 (case 3.b.). The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `dest:p1 = split(dest):` $\big(\{\mathtt{0}\}$ `'a'` $\{\mathtt{4}\}$, $\{\mathtt{5}\}$ `T` $\{\mathtt{8}\}\big)$

- `dest:p2 = split(dest):` $\big(\{\mathtt{0}\}$ `'a'` $\{\mathtt{4}\}$, $\{\mathtt{5}\}$ `T` $\{\mathtt{8}\}\big) \wedge$
  `src:p2 = split(src):` $\big(\emptyset, \{\mathtt{0}\}$ `'b'` $\{\mathtt{5}\}\big)$
  $\mathrm{Check}(string(\mathtt{dest})) = \mathrm{TRUE}$, $\mathrm{Check}(string(\mathtt{src})) = \mathrm{FALSE}$

- `dest:p3 = split(dest):` $\big(\emptyset, \{\mathtt{0}\}$ `'a'` $\{\mathtt{4}\}$ `'b'` $\{\mathtt{9}\}\big) \wedge$
  `src:p3 = split(src):` $\big(\emptyset, \{\mathtt{0}\}$ `'b'` $\{\mathtt{5}\}\big)$

- `dest:p4 = dest:p3` $\wedge$ `src:p4 = src:p3`

$\triangle$

4.2.3 The string length statement

strlen(str) is a string function that determines the length of a C character array, computing the number of bytes in the string to which the null-terminated char array points, not including the terminating null byte. The formal declaration for string length function is:

size_t strlen(const char *str)

str is the string whose length has to be found. Implementation:

```
#include <stddef.h>
size_t strlen(const char *str)
{
        size_t i;
        for (i = 0; str[i] != '\0'; i++) ;
        return i;
}
```

The returned value is the length of str; the function has no failure mode and no error return [13]. The analysis will be performed on static array.

The string length original statement result - from the string.h header in the C standard library - may be misleading, or even erroneous, in the case in which the char array function parameter had the string of interest equal to `undef`. Below, we stigmatize the behaviour of the string length function, also the ambiguous one.

Definition 4.12 (*semantics of the string length function*)**.** Let `str` be a char array declared in a program (`str` $\in$ ACVar), with concrete value represented by the quintuples $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$ $(str1 \in \text{ACVal})$ and $\mathsf{T}_{\texttt{str}} \neq \emptyset$, then:

$$\mathfrak{S}[\![\textbf{strlen(str)}]\!]\sigma = n \in \mathbb{Z}_+$$

1. $string(\texttt{str}) \neq \texttt{undef} \Rightarrow n = min(\mathsf{T}_{\texttt{str}}) - [\![\texttt{S.low}_{\texttt{str}}]\!]\rho$
2. $string(\texttt{str}) = \texttt{undef} \Rightarrow n = \top$

   `undefined behaviour`

Notice that, in any case, the concrete value of `str`, after the application of the string length function on it, remain unchanged.

◊

Example **4.10.** Given the program:

```
#include <stdio.h>
#include <string.h>
#include <stddef.h>
         int main() {
/* 0: */ char str[8] = ``aaaa'';
/* 1: */ size_t l = strlen(str);
/* 2: */ return l;
/* 3: */ }
```

The concrete value of the char array `str` throughout the program is given by the quintuple $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$. Entering into the detail, we have that:

- $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$

$$\llbracket \texttt{str.low} \rrbracket \rho = 0$$
$$\llbracket \texttt{str.high} \rrbracket \rho = 8$$
$$\begin{aligned} codom(S) = \{&(0,\text{`a'}), (1,\text{`a'}), (2,\text{`a'}), (3,\text{`a'}), (4,\text{`\textbackslash 0'}), (5,\texttt{T}),\\ &(6,\texttt{T}), (7,\texttt{T})\} \end{aligned}$$
$$\mathsf{T}_{\texttt{str}} = \{4\}$$
$$min(\mathsf{T}_{\texttt{str}}) = 4$$

$$string(\texttt{str}) = < \texttt{a a a a \textbackslash 0} >$$

At the program point 2 we obtain that $\mathfrak{S}\llbracket \textbf{strlen(str)} \rrbracket \sigma = min(\mathsf{T}_{\texttt{str}}) - 0 = 4$. The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `str:p1 = split(str):` $\big(\{0\}$ ‘a’ $\{4\}$, $\{5\}$ T $\{8\}\big)$
  Check($string$(`str`)) = TRUE
- `str:p2 = split(str):` $\big(\{0\}$ ‘a’ $\{4\}$, $\{5\}$ T $\{8\}\big)$ $\wedge$ `l.p2:` 4
- `str:p3 = str:p2` $\wedge$ `l.p3 = l.p2`

△

EXAMPLE **4.11.** Given the program:

```
#include <stdio.h>
#include <string.h>
#include <stddef.h>
        int main() {
/* 0: */ char str[5] = ''aaaaa'';
/* 1: */ size_t l = strlen(str);
/* 2: */ return l;
/* 3: */ }
```

The concrete value of the char array `str` throughout the program is given by the quintuple $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$. Entering into the detail, we have that:

- $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$

$$\llbracket \texttt{str.low} \rrbracket \rho = 0$$
$$\llbracket \texttt{str.high} \rrbracket \rho = 5$$
$$codom(S) = \{(0,\text{'a'}), (1,\text{'a'}), (2,\text{'a'}), (3,\text{'a'}), (4,\text{'a'})\}$$
$$\mathsf{T}_{\texttt{str}} = \emptyset$$
$$min(\mathsf{T}_{\texttt{str}}) \text{ does not exist}$$

$string(\texttt{str}) = \texttt{undef}$

At the program point 2 we obtain that $\mathfrak{S}\llbracket \textbf{strlen}(\texttt{str}) \rrbracket \sigma = \top$. The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `str:p1 = split(str):` $\big(\emptyset, \{0\}$ `‘a’` $\{5\}\big)$

  $\text{Check}(string(\texttt{src})) = \text{FALSE}$

- `str:p2 = split(str):` $\big(\emptyset, \{0\}$ `‘a’` $\{5\}\big) \wedge$ `l.p2:`$\top$

- `str:p3 = str:p2` $\wedge$ `l.p3 = l.p2`

△

4.2.4 The string character statement

strchr(str,c) is a function in the C standard library that 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 formal declaration for string character function is:

char *strchr(const char *str, int c);

str is the C string to be scanned and c is the character to be searched in str. Implementation:

```
char *strchr(const char *str, int c)
{
        while (*str != (char)c)
           if (!*str++)
              return 0;
        return (char *)str;
}
```

Observe that there is no check if str is null. strchr() returns a pointer to the first occurrence of character c located within str. If character c does not occur in the string, strchr() returns a null pointer [13]. Notice that the subsequent analysis will be performed on static arrays.

The string character original statement result - from the string.h header in the C standard library - may be misleading, or even erroneous, in the case in which the char array function parameter had the string of interest equal to `undef`. Below, we stigmatize the behaviour of the string character function, also the ambiguous one.

Definition 4.13 (*semantics of the string character function*)**.** Let `str` be a char array declared in a program (`str` ∈ ACVar), with concrete value represented by the quintuples $str = (\rho, \texttt{str.low}, \texttt{str.high}, S,$ $\mathsf{T}_{\texttt{str}})$ ($str \in$ ACVal), then:

$$\mathfrak{S}[\![\textbf{strchr}(\texttt{str}, c)]\!]\sigma = \sigma' \in \text{ACState}$$

1. $string(\texttt{str}) \neq \texttt{undef}$

   In this case the string character function result is a suffix of the string of interest of `str` ($string(\texttt{str}) \equiv \texttt{str[i,j]}$ such that $\texttt{i} \in [[\![\texttt{S.low}_{\texttt{str}}]\!]\rho, [\![\texttt{S.high}_{\texttt{str}}]\!]\rho] \;\wedge\; \texttt{j} = [\![\texttt{S.high}_{\texttt{str}}]\!]\rho$). A suffix is a sub-string of the string of interest of `str` and as such its concrete value is represented by its own quintuple, as presented in the Definition 4.5.

a. If $c \in string(\texttt{str})$:

- $\sigma'(\texttt{str}) = (\rho, \texttt{str.low}', \texttt{str.high}', S', \texttt{T'}_{\texttt{str}})$

$$\begin{aligned}
&\texttt{str.low}' = min\{\texttt{i} \mid \texttt{str}[\texttt{i}] = c \land \\
&\qquad\qquad [\![\texttt{S.low}_{\texttt{str}}]\!]\rho \leqslant \texttt{i} \leqslant [\![\texttt{S.high}_{\texttt{str}}]\!]\rho\} \\
&\texttt{str.high}' = \texttt{S.high}_{\texttt{str}} + 1 \\
&S'\colon \forall i \in [[\![\texttt{str.low'}]\!]\rho, [\![\texttt{str.high'}]\!]\rho), \\
&\qquad \texttt{str1'}[i] \rightarrow (i, v) \mid (m, v) \in codom(S) \\
&\qquad\qquad \land\, m = i \\
&\texttt{T'}_{\texttt{str}} = \{min(\texttt{T}_{\texttt{str}})\} \\
&min(\texttt{T'}_{\texttt{str}}) = min(\texttt{T}_{\texttt{str}})
\end{aligned}$$

- $\sigma'[\texttt{s}_{\texttt{str}}] = \sigma[\texttt{s}_{\texttt{str}}](i, j)$

$$\begin{aligned}
&i = min\{i \mid \texttt{str}[i] = c \land \\
&\qquad [\![\texttt{S.low}_{\texttt{str}}]\!]\rho \leqslant i \leqslant [\![\texttt{S.high}_{\texttt{str}}]\!]\rho\} \\
&j = [\![\texttt{S.high}_{\texttt{str}}]\!]\rho
\end{aligned}$$

b. If $c \notin string(\texttt{str}) \land c \in \texttt{str}$:

- $\sigma'(\texttt{str}) = \texttt{null}$
- $\sigma'[\texttt{s}_{\texttt{str}}] = \emptyset$

c. If $c \notin string(\texttt{str}) \land c \notin \texttt{str}$:

- $\sigma'(\texttt{str}) = \texttt{null}$
- $\sigma'[\texttt{s}_{\texttt{str}}] = \emptyset$

2. $string(\texttt{str}) = \texttt{undef}$

undefined behaviour

In this case we assume that the string character function result is a suffix of `str` ($\texttt{str}[\texttt{i}, \texttt{j}]$ such that $\texttt{i} \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho) \land \texttt{j} = [\![\texttt{str.high}]\!]\rho - 1$), that is a sub-array of `str` and as such its concrete value is represented by its own quintuple, as presented in the Definition 4.5.

a. If $c \in \texttt{str}$:

- $\sigma'(\texttt{str}) = (\rho, \texttt{str.low'}, \texttt{str.high'}, S', \mathsf{T'}_{\texttt{str}})$

$$\begin{aligned}
&\texttt{str.low'} = \mathit{min}\{\texttt{i} \mid \texttt{str[i]} = c \wedge \\
&\qquad\qquad [\![\texttt{str.low}]\!]\rho \leqslant \texttt{i} < [\![\texttt{str.high}]\!]\rho\} \\
&\texttt{str.high'} = \texttt{str.high} \\
&S'\text{: } \forall i \in [[\![\texttt{str.low'}]\!]\rho, [\![\texttt{str.high'}]\!]\rho), \\
&\qquad \texttt{str1'}[i] \rightarrow (i, v) \mid (m, v) \in \mathit{codom}(S) \\
&\qquad\qquad \wedge\, m = i \\
&\mathsf{T'}_{\texttt{str}} = \emptyset \\
&\mathit{min}(\mathsf{T'}_{\texttt{str}}) \text{ does not exist}
\end{aligned}$$

- $\sigma'[\texttt{s}_{\texttt{str}}] = \emptyset$

b. If $c \notin \texttt{str}$:

- $\sigma'(\texttt{str}) = \texttt{null}$
- $\sigma'[\texttt{s}_{\texttt{str}}] = \emptyset$

Notice that $\sigma'(\texttt{str}) \equiv \sigma(\texttt{str[i, j]})$

◊

EXAMPLE **4.12.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char str[10] = ‘‘aabbcc’’;
/* 1: */ char *p = NULL;
/* 2: */ p = strchar(str,‘b’);
/* 3: */ printf(‘‘%s’’,p);
/* 4: */ return 0;
/* 5: */ }
```

The concrete value of the char array `str` at the program point `1` is given by the quintuple $\mathit{str} = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$. Entering into the detail, we have that:

- $\mathit{str} = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$

$$[\![\texttt{str.low}]\!]\rho = 0$$
$$[\![\texttt{str.high}]\!]\rho = 10$$

$$\begin{aligned}
codom(S) &= \{(0,\text{‘}\mathtt{a}\text{’}), (1,\text{‘}\mathtt{a}\text{’}), (2,\text{‘}\mathtt{b}\text{’}), (3,\text{‘}\mathtt{b}\text{’}), (4,\text{‘}\mathtt{c}\text{’}), (5,\text{‘}\mathtt{c}\text{’}),\\
&\quad (6,\text{‘}\backslash\mathtt{0}\text{’}), (7,\mathtt{T}), (8,\mathtt{T}), (9,\mathtt{T})\}\\
\mathsf{T}_{\mathtt{str}} &= \{6\}\\
min(\mathsf{T}_{\mathtt{str}}) &= 6
\end{aligned}$$

$$string(\mathtt{str}) =< \mathtt{a}\ \mathtt{a}\ \mathtt{b}\ \mathtt{b}\ \mathtt{c}\ \mathtt{c}\ \backslash\mathtt{0} >$$

At the program point 3 we obtain that $\mathfrak{S}[\![\textbf{strchr}(\mathtt{str}, \mathtt{b})]\!]\sigma = \sigma'$, where:

- $\sigma'(\mathtt{str}) = (\rho, \mathtt{str.low'}, \mathtt{str.high'}, S', \mathsf{T'}_{\mathtt{str}})$

$$\begin{aligned}
&[\![\mathtt{str.low'}]\!]\rho = 2\\
&[\![\mathtt{str.high'}]\!]\rho = 7\\
&codom(S') = \{(2,\text{‘}\mathtt{b}\text{’}), (3,\text{‘}\mathtt{b}\text{’}), (4,\text{‘}\mathtt{c}\text{’}), (5,\text{‘}\mathtt{c}\text{’}), (6,\text{‘}\backslash\mathtt{0}\text{’})\}\\
&\mathsf{T'}_{\mathtt{str}} = 6\\
&min(\mathsf{T'}_{\mathtt{str}}) = 6
\end{aligned}$$

- $\sigma'[\mathtt{s}_{\mathtt{str}}] = \sigma[\mathtt{s}_{\mathtt{str}}](2, 6) = \{2\}\ \text{‘}\mathtt{b}\text{’}\ \{4\}\ \text{‘}\mathtt{c}\text{’}\ \{6\}$

At the end of the analysis we have the certainty that the program result is a char sub-array that contain a string corresponding to a specific suffix of the string of interest of `str`, the char array to which the string character function has been applied in the program above, as presented in the Definition 4.13 (case 1.a.). The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `str:p1 = split(str):` $\big(\{0\}\ \text{‘}\mathtt{a}\text{’}\ \{2\}\ \text{‘}\mathtt{b}\text{’}\ \{4\}\ \text{‘}\mathtt{c}\text{’}\ \{6\}, \{7\}\ \mathtt{T}\ \{10\}\big)$
- `str:p2 = str:p1` $\wedge$ `p:p2 = split(p):` $(\emptyset, \emptyset)$
  Check($string(\mathtt{str})$) = TRUE, Check($string(\mathtt{p})$) = FALSE
- `str:p3 = split(str):` $\big(\{0\}\ \text{‘}\mathtt{a}\text{’}\ \{2\}\ \text{‘}\mathtt{b}\text{’}\ \{4\}\ \text{‘}\mathtt{c}\text{’}\ \{6\}, \{7\}\ \mathtt{T}\ \{10\}\big) \wedge$
  `p:p3 = str[2,6]:p3 = split(p):` $\big(\{2\}\ \text{‘}\mathtt{b}\text{’}\ \{4\}\ \text{‘}\mathtt{c}\text{’}\ \{6\}, \emptyset\big)$
- `str:p4 = str:p3` $\wedge$ `p:p4 = p:p3`
- `str:p5 = str:p4` $\wedge$ `p:p5 = p:p4`

△

Observe that, given a char array, named `C`, it is possible implement a string character statement that exploits the segmentation of the char array function parameter in order to search for the minimum segment in $Seg(\texttt{C})$ where $x_{seg(\texttt{C})_i} = c$, as proposed in the algorithm below. To avoid falling into wrong conclusions, the string character algorithm that we will present, has been implemented in order to work only on char arrays that certainly contain the string of interest. In the case in which $string(\texttt{C}) = \texttt{undef}$, our string character function produce an error. So, we are always in the case 1 of the Definition 4.13.

Algorithm 4.1. (*m-string character function*).

```
Given:

  • str:  a char array
  • split(str)
  • c:  the character to search in str

if $s_{str} \neq \emptyset$
    compute:
    $Seg(s_{str}) = \{seg(s_{str})_i \mid i = 1, \dots, n\}$
    then
        for all $i \in [1, n]$
            if $x_{seg(s_{str})_i} \equiv c$
                return $str[lb_{seg(s_{str})_i}, S.high_{str}]$
            otherwise i++
else STOP!
```

□

We propose another string character implementation that exploits the *suffix array* data structure [7]. Also in this case, in order to avoid falling into wrong conclusions and to avoid huge computations, the string character algorithm based on suffix array, that we will present, has been implemented in order to work only on char arrays that certainly contain the string of interest, otherwise our string character function produce an error. So, we are always in the case 1 of the Definition 4.13. Below, we provide a refined definition of the suffix domain [7] that inherits the features of our char array representation.

Definition 4.14 (*suffix array - a refinement*)**.** Consider a char array, named `str`, with associated concrete value and splitting and consider $string(\texttt{str}) \neq \texttt{undef}$.

Let $string(\texttt{str})[i, [\![\texttt{S.high}_{\texttt{str}}]\!]\rho]$ be the sub-string of $string(\texttt{str})$ ranging from $[\![\texttt{S.low}_{\texttt{str}}]\!]\rho \leqslant i \leqslant [\![\texttt{S.high}_{\texttt{str}}]\!]\rho$ to $[\![\texttt{S.high}_{\texttt{str}}]\!]\rho$, the suffix array `Pos` of $string(\texttt{C})$ is defined to be an array of integers providing the starting position of suffixes of $string(\texttt{str})$ in lexicographical order; namely, $\texttt{Pos}[k]$ contains the start position of the kth smallest suffix in the set:

$\{string(\texttt{str})[[\![\texttt{S.low}_{\texttt{str}}]\!]\rho, [\![\texttt{S.high}_{\texttt{str}}]\!]\rho], string(\texttt{str})[[\![\texttt{S.low}_{\texttt{str}}]\!]\rho + 1, [\![\texttt{S.high}_{\texttt{str}}]\!]\rho], \ \dots\ , string(\texttt{str})[[\![\texttt{S.high}_{\texttt{str}}]\!]\rho, [\![\texttt{S.high}_{\texttt{str}}]\!]\rho]\}$,

and we consider the partial order on suffixes satisfying:

$$string(\texttt{str})[\texttt{Pos}[k-1], [\![\texttt{S.high}_{\texttt{str}}]\!]\rho] \prec string(\texttt{str})[\texttt{Pos}[k], [\![\texttt{S.high}_{\texttt{str}}]\!]\rho]$$

where $\prec$ denotes the lexicographical order in the ASCII character domain.

◊

Example **4.13.** Let C: $<$ `m i n n i e \0 m o u s e \0` $>$ be a char array, then:

- $c = (\rho, 0, 13, C, \{6, 12\})$
- $string(\texttt{C}) = <$ `m i n n i e \0` $>$

i	0	1	2	3	4	5	6
$string(\texttt{C})[i]$	m	i	n	n	i	e	\0

- $split(\texttt{C}) = (\{0\}$ ‘m’ $\{1\}$ ‘i’ $\{2\}$ ‘n’ $\{4\}$ ‘i’ $\{5\}$ ‘e’ $\{6\}$, $\texttt{ns}_{\texttt{C}})$

The string of interest of `C` has the following suffixes:

	Suffix	i
$string(\texttt{C})[0,6]$	< m i n n i e \0 >	0
$string(\texttt{C})[1,6]$	< i n n i e \0 >	1
$string(\texttt{C})[2,6]$	< n n i e \0 >	2
$string(\texttt{C})[3,6]$	< n i e \0 >	3
$string(\texttt{C})[4,6]$	< i e \0 >	4
$string(\texttt{C})[5,6]$	< e \0 >	5
$string(\texttt{C})[6,6]$	< \0 >	6

These suffixes can be sorted in lexicographical order:

	Suffix	i
$string(\texttt{C})[6,6]$	$<\texttt{\textbackslash 0}>$	6
$string(\texttt{C})[5,6]$	$<\texttt{e}\ \texttt{\textbackslash 0}>$	5
$string(\texttt{C})[4,6]$	$<\texttt{i e \textbackslash 0}>$	4
$string(\texttt{C})[1,6]$	$<\texttt{i n n i e \textbackslash 0}>$	1
$string(\texttt{C})[0,6]$	$<\texttt{m i n n i e \textbackslash 0}>$	0
$string(\texttt{C})[3,6]$	$<\texttt{n i e \textbackslash 0}>$	3
$string(\texttt{C})[2,6]$	$<\texttt{n n i e \textbackslash 0}>$	2

The suffix array `Pos` contains the starting positions of these suffixes:

k	0	1	2	3	4	5	6
$\texttt{Pos}[k]$	6	5	4	1	0	3	2

So, for example, $\texttt{Pos}[5]$ contains the value 3, and therefore refers to the suffix starting at position 3 within $string(\texttt{C})$, which is the suffix $<\texttt{n i e \textbackslash 0}>$.

△

Algorithm 4.2. (*m-string character function - suffix version*).

```
Given:

  • str:  a char array

  • split(str)

  • c:  the character to search in str

if s_str ≠ ∅
    compute:
    Pos(string(str))
    then
        return Pos[k] = i → suffix(string(str))[i,j]
else STOP!
```

□

Lemma 4.1. *Given the char array* $\texttt{D}$*, with* $string(\texttt{D}) \neq \texttt{undef}$*. Let* $\texttt{Pos}[k]$ *be the result of the Algorithm 4.2.* $\texttt{Pos}[k]$ *contains the suffix value* $i = min\{l : [\![\texttt{S.low}_{\texttt{D}}]\!]\rho \leqslant l \leqslant [\![\texttt{S.high}_{\texttt{D}}]\!]\rho \ \wedge \ string(\texttt{D})[l] = c\}$ *that points to the suffix* $\texttt{D}[i,j]$*, whose concrete value is represented by* $\sigma' \in \text{ACState}$ *that is the result of the string character statement presented in the Definition 4.13 (case 1.a.).*

EXAMPLE **4.14.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char str[] = ‘‘minnie’’;
/* 1: */ char *p = NULL;
/* 2: */ p = strchr(str,‘n’);
/* 3: */ printf(‘‘%s’’,p);
/* 4: */ return 0;
/* 5: */ }
```

The concrete value of the char array `str` at program point `1` is given by the quintuple $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$. Entering into the detail, we have that:

$$[\![\texttt{str.low}]\!]\rho = 0$$

$$[\![\texttt{str.high}]\!]\rho = 7$$

$$codom(S) = \{(0,‘\texttt{m}’), (1,‘\texttt{i}’), (2,‘\texttt{n}’), (3,‘\texttt{n}’), (4,‘\texttt{i}’), (5,‘\texttt{e}’), (6,‘\backslash\texttt{0}’)\}$$

$$\mathsf{T}_{\texttt{str}} = \{6\}$$

$$min(\mathsf{T}_{\texttt{str}}) = 6$$

From the Definition 4.13 we know that $\mathfrak{S}[\![\textbf{strchr}(\texttt{str,n})]\!]\sigma = \sigma'$, where $\sigma'(\texttt{str}) \equiv \sigma(\texttt{str}[2,6]) = (\rho, 2, 7, codom(str_{[2,6]}) = \{(2,‘\texttt{n}’), (3,‘\texttt{n}’), (4,‘\texttt{i}’), (5,‘\texttt{e}’), (6,‘\backslash\texttt{0}’)\}, \{6\})$. Recovering the Example 4.13 and the Algorithm 4.2 it is possible to determine that $\sigma'(\texttt{str})$ is the concrete value of the $string(\texttt{str})$ suffix to which points the value in $\texttt{Pos}[6]$. As a matter of fact, $\texttt{Pos}[6] = 2$ (that is the minimum element of the set $\{2,3\}$) and therefore refers to the suffix starting at position 2 within $string(\texttt{str})$, which is the suffix $string(\texttt{str})[2,6] = <\texttt{n n i e} \backslash\texttt{0}>$. The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `str:p1 = split(str):` $(\{0\}\ ‘\texttt{m}’\ \{1\}\ ‘\texttt{i}’\ \{2\}\ ‘\texttt{n}’\ \{4\}\ ‘\texttt{i}’\ \{5\}\ ‘\texttt{e}’\ \{6\}, \emptyset)$
- `str:p2 = str:p1` $\wedge$ `p:p2 = split(p):` $(\emptyset, \emptyset)$

  $\text{Check}(string(\texttt{str})) = \text{TRUE}$, $\text{Check}(string(\texttt{p})) = \text{FALSE}$

- `str:p3 = str:p2` $\wedge$
  `p:p3 = str[2,6]:p3 = split(p):` ({2} ‘n’ {4} ‘i’ {5} ‘e’ {6}, ∅)
- `str:p4 = str:p3` $\wedge$ `p:p4 = p:p3`
- `str:p5 = str:p4` $\wedge$ `p:p5 = p:p4`

△

4.2.5 The string compare statement

strcmp(str1,str2) is a function in the C standard library that lexicographically compares two C strings. The formal declaration for string compare function is:

```
int strcmp(const char *str1, const char *str2);
```

str1 is the first string to be compared and str2 is the second string to be compared. Implementation:

```
int strcmp (const char *str1, const char *str2)
{
    while(*str1 && (*str1==*str2))
        str1++,str2++;
    return *(const unsigned char*)str1− *(const unsigned char*)str2;
}
```

The returned value is 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, considering a lexicographic order [13].

Notice that the subsequent analysis will be performed on static arrays.

The string compare original statement result - from the string.h header in the C standard library - may be misleading, or even erroneous, in the case in which one of the two char arrays function parameters, or both, had the string of interest equal to `undef`. Below, we stigmatize the behaviour of the string copy function, also the ambiguous one.

Definition 4.15 (*semantics of the string compare function*)**.** Let `str1` and `str2` be two char arrays declared in a program ($\texttt{str1}, \texttt{str2} \in$ ACVar), with concrete values respectively represented by the quintuples $str1 = (\rho, \texttt{str1.low}, \texttt{str1.high}, S_1, \mathsf{T}_{\texttt{str1}})$ and $str2 = (\rho, \texttt{str2.low}, \texttt{str2.high}, S_2, \mathsf{T}_{\texttt{str2}})$ ($str1, str2 \in$ ACVal), then:

$$\mathfrak{S}[\![\textbf{strcmp}(\texttt{str1}, \texttt{str2})]\!]\sigma = n \in \mathbb{Z}$$

- $string(\texttt{str1}) \neq \texttt{undef} \;\wedge\; (\texttt{str2}) \neq \texttt{undef}$

$$n \text{ is } \begin{cases} < 0 & \text{if } string(\texttt{str1}) \prec string(\texttt{str2}) \\ = 0 & \text{if } string(\texttt{str1}) = string(\texttt{str2}) \\ > 0 & \text{if } string(\texttt{str1}) \succ string(\texttt{str2}) \end{cases}$$

- $string(\texttt{str1}) = \texttt{undef} \;\wedge\; (\texttt{str2}) \neq \texttt{undef}$

  undefined behaviour

  $n = \top$

- $string(\texttt{str1}) \neq \texttt{undef} \;\wedge\; (\texttt{str2}) = \texttt{undef}$

  undefined behaviour

  $n = \top$

- $string(\texttt{str1}) = \texttt{undef} \;\wedge\; (\texttt{str2}) = \texttt{undef}$

  undefined behaviour

  $n = \top$

Notice that, in any case, the concrete values of `str1` and `str2`, after the application of the string compare function on them, remain unchanged.

$\Diamond$

We are going to introduce two new implementations of the string compare function. In order to avoid falling into wrong conclusions, both the string compare algorithms that we will present, have been implemented in order to work only on char arrays that certainly contain the string of interest.

Our first string compare procedure is based on segments comparison. In particular, given two char arrays, str1 and str2, we compare the two strings segment-wise.

Algorithm 4.3. (*m-string compare function*).

```
Given:

  • str1 and str2:  the char arrays to be compared

  • split(str1) and split(str2)

if $\mathtt{s_{str1}} \wedge \mathtt{s_{str2}} \neq \emptyset$
    compute:
    $\mathtt{Seg(s_{str1}) = \{seg(s_{str1})_i \mid i = 1, \dots, n\}}$
    $\mathtt{Seg(s_{str2}) = \{seg(s_{str2})_j \mid j = 1, \dots, m\}}$
    then
        for all $\mathtt{i \in [1,n] \wedge j \in [1,m]}$ such that $\mathtt{i = j}$
            if $\mathtt{x_{seg(s_{str1})_i} \not\equiv x_{seg(s_{str2})_j}}$
                return n
            otherwise
               if $\mathtt{ub_{seg(s_{str1})_i} - lb_{seg(s_{str1})_i} \not\equiv ub_{seg(s_{str2})_j} - lb_{seg(s_{str2})_j}}$
                   return n
               otherwise i++ $\wedge$ j++
        return n
else STOP!
```

□

From now on, we will refer to the result of the Algorithm 4.3 as $n = Alg_1(\text{str1}, \text{str2})$, where str1 and str2 are the compared char arrays.

Lemma 4.2. *Given two char arrays, $\mathtt{D}$ and $\mathtt{R}$, with string$(\mathtt{D}) \neq$ **undef** and string$(\mathtt{R}) \neq$ **undef**. Let $n = Alg_1(\mathtt{D}, \mathtt{R})$. Then:*

1. *$n < 0 \Leftrightarrow \mathtt{D} \prec \mathtt{R}$*

    a. *If string$(\mathtt{D})$ and string$(\mathtt{R})$ do not share any prefix the comparison regards just the first segments of $s_{\mathtt{D}}$ and $s_{\mathtt{R}}$. As a matter of fact the seg$(s_{\mathtt{D}})_1$ and the seg$(s_{\mathtt{R}})_1$ differ for the characters that they contain. Notice that the considered segments can also differ for the respectively upper bounds. In* *this case n corresponds to the difference between the corresponding ASCII constant value to the character in $seg(s_D)_1$ and the corresponding ASCII constant value to the character in $seg(s_R)_1$. Formally, given the char scalar variables $D[lb_{seg(s_D)_1}]$ and $R[lb_{seg(s_R)_1}]$ and the environment $\rho \in \mathcal{R}_v$ mapping the given variables to their ASCII constant domain value, then:*

$$n = [\![D[lb_{seg(s_D)_1}]]\!]\rho - [\![R[lb_{seg(s_R)_1}]]\!]\rho$$

b. *If $string(D)$ and $string(R)$ share a prefix of length $|string(D)|-1$ (we are in the case in which the length of the string of interest of D is strictly smaller than the length of the string of interest of R), assume that the s_D sub-segmentation has m segments then, for all i=j such that i=1,...,m-1 we have that:*

$$\begin{array}{rcl} x_{seg(s_D)_i} & = & x_{seg(s_R)_j} \\ ub_{seg(s_D)_i} - lb_{seg(s_D)_i} & = & ub_{seg(s_R)_j} - lb_{seg(s_R)_j} \end{array}$$

and, for i=j such that i=m:

$$\begin{array}{rcl} x_{seg(s_D)_{i=m}} & = & x_{seg(s_R)_{j=m}} \\ ub_{seg(s_D)_m} - lb_{seg(s_D)_m} & \leqslant & ub_{seg(s_R)_m} - lb_{seg(s_R)_m} \end{array}$$

The segments comparison procedure stops when it compares the last segment of s_D, that is $seg(s_D)_m$, in which the segment upper bound is equal to $S.high_D$, with the respective $seg(s_R)_m$. In this case we have two possibilities:

I. if $ub_{seg(s_D)_m} - lb_{seg(s_D)_m} = ub_{seg(s_R)_m} - lb_{seg(s_R)_m}$, n corresponds to the negative ASCII constant value to the character in $seg(s_R)_{m+1}$. Formally:

$$n = -[\![R[lb_{seg(s_R)_{m+1}}]]\!]\rho$$

II. if $ub_{seg(s_D)_m} - lb_{seg(s_D)_m} < ub_{seg(s_R)_m} - lb_{seg(s_R)_m}$, n corresponds to the negative ASCII constant value to the character in $seg(s_R)_m$. Formally:

$$n = -[\![R[lb_{seg(s_R)_m}]]\!]\rho$$

Notice that, in both cases, $string(D)$ and $string(R)$ share a prefix of length equal to $|ub_{seg(s_D)_m} - lb_{seg(s_D)_1}|$.

c. *If string(D) and string(R) share a prefix of length strictly smaller than $|string(\mathtt{D})| - 1$ means that, at least:*

$$\begin{array}{rcl} x_{seg(s_D)_1} & = & x_{seg(s_R)_1} \\ ub_{seg(s_D)_1} - lb_{seg(s_D)_1} & \lesseqgtr & ub_{seg(s_R)_1} - lb_{seg(s_R)_1} \end{array}$$

The procedure stops to the first comparison in which or the segments elements are not the same ($x_{seg(s_D)_{i>1}} \neq x_{seg(s_R)_{j>1}}$) or in which the differences between the upper bounds and the lower bounds of the compared segments are different ($ub_{seg(s_D)_{i\geqslant 1}} - lb_{seg(s_D)_{i\geqslant 1}} \neq ub_{seg(s_R)_{j\geqslant 1}} - lb_{seg(s_R)_{j\geqslant 1}}$). In the case in which the compared segments have different elements, n is equal to the difference between the ASCII constant value to the character in $seg(s_D)_{i>1}$ and the ASCII constant value to the character in $seg(s_R)_{j>1}$. Formally:

$$n = [\![\mathtt{D}[lb_{seg(s_D)_{i>1}}]]\!]\rho - [\![\mathtt{R}[lb_{seg(s_R)_{j>1}}]]\!]\rho$$

In the case in which the differences between the upper bounds and the lower bounds of the compared segments do not coincide we distinguish two cases:

I. if $ub_{seg(s_D)_{i\geqslant 1}} - lb_{seg(s_D)_{i\geqslant 1}} > ub_{seg(s_R)_{j\geqslant 1}} - lb_{seg(s_R)_{j\geqslant 1}}$, n is equal to the difference between the ASCII constant value to the character in $seg(s_D)_{i\geqslant 1}$ and the ASCII constant value to the character in $seg(s_R)_{(j\geqslant 1)+1}$. Formally:

$$n = [\![\mathtt{D}[lb_{seg(s_D)_{i\geqslant 1}}]]\!]\rho - [\![\mathtt{R}[lb_{seg(s_R)_{(j\geqslant 1)+1}}]]\!]\rho$$

II. if $ub_{seg(s_D)_{i\geqslant 1}} - lb_{seg(s_D)_{i\geqslant 1}} < ub_{seg(s_R)_{j\geqslant 1}} - lb_{seg(s_R)_{j\geqslant 1}}$, n is equal to the difference between the ASCII constant value to the character in $seg(s_D)_{(i\geqslant 1)+1}$ and the ASCII constant value to the character in $seg(s_R)_{j\geqslant 1}$. Formally:

$$n = [\![\mathtt{D}[lb_{seg(s_D)_{(i\geqslant 1)+1}}]]\!]\rho - [\![\mathtt{R}[lb_{seg(s_R)_{j\geqslant 1}}]]\!]\rho$$

Here, string(D) and string(R) share a prefix of length equal to $|min(ub_{seg(s_D)_{i\geqslant 1}} - lb_{seg(s_D)_1}, ub_{seg(s_R)_{i\geqslant 1}} - lb_{seg(s_R)_1})|$

Notice that in the case in which D lexicographically precedes R, s_D and s_R could not have the same number of segments.

2. $n = 0 \Leftrightarrow \mathtt{D} = \mathtt{R}$

D and R are lexicographically equal if string(D) and string(R) are of the same length, have the same number of segments and the differences between the segment bounds and the contained elements *coincides. Assume that the $\mathtt{s_D}$ and the $\mathtt{s_R}$ sub-segmentations have m segments then, for all i=j such that i=1,...,m we have that:*

$$\begin{aligned} x_{seg(\mathtt{s_D})_i} &= x_{seg(\mathtt{s_R})_j} \\ ub_{seg(\mathtt{s_D})_i} - lb_{seg(\mathtt{s_D})_i} &= ub_{seg(\mathtt{s_R})_j} - lb_{seg(\mathtt{s_R})_j} \end{aligned}$$

The segments comparison procedure stops when it compares the last segment of $\mathtt{s_D}$, $seg(\mathtt{s_D})_m$, in which the segment upper bound is equal to $\mathtt{S.high_D}$, with the respective $seg(\mathtt{s_R})_m$, in which the segment upper bound is equal to $\mathtt{S.high_R}$ ($[\![\mathtt{S.high_D}]\!]\rho = [\![\mathtt{S.high_R}]\!]\rho$). In this case n is equal to the difference between the corresponding ASCII constant value to the character in $seg(\mathtt{s_D})_m$ and the corresponding ASCII constant value to the character in $seg(\mathtt{s_R})_m$ (since the two segments contain the same element with the same ASCII constant value, n will be always equal to 0). Formally:

$$n = [\![\mathtt{D}[lb_{seg(\mathtt{s_D})_m}]]\!]\rho - [\![\mathtt{R}[lb_{seg(\mathtt{s_R})_m}]]\!]\rho$$

Furthermore, string($\mathtt{D}$) and string($\mathtt{R}$) share a prefix of length equal to $|ub_{seg(\mathtt{s_D})_m} - lb_{seg(\mathtt{s_D})_1}|$ ($= |ub_{seg(\mathtt{s_R})_m} - lb_{seg(\mathtt{s_R})_1}|$).

3. $n > 0 \Leftrightarrow \mathtt{D} \succ \mathtt{R}$

- ***a.*** *If string($\mathtt{D}$) and string($\mathtt{R}$) do not share any prefix, the considerations made in 1.**a.** hold.*
- ***b.*** *If string($\mathtt{D}$) and string($\mathtt{R}$) share a prefix of length |string($\mathtt{R}$)| -1 (we are in the case in which the length of the string of interest of $\mathtt{D}$ is strictly greater than the length of the string of interest of $\mathtt{R}$), assume that the $\mathtt{s_R}$ sub-segmentation has m segments then, for all i=j such that j=1,...m-1 we have that:*

$$\begin{aligned} x_{seg(\mathtt{s_D})_i} &= x_{seg(\mathtt{s_R})_j} \\ ub_{seg(\mathtt{s_D})_i} - lb_{seg(\mathtt{s_D})_i} &= ub_{seg(\mathtt{s_R})_j} - lb_{seg(\mathtt{s_R})_j} \end{aligned}$$

and, for i=j such that j=m:

$$\begin{aligned} x_{seg(\mathtt{s_D})_m} &= x_{seg(\mathtt{s_R})_m} \\ ub_{seg(\mathtt{s_D})_m} - lb_{seg(\mathtt{s_D})_m} &\geqslant ub_{seg(\mathtt{s_R})_m} - lb_{seg(\mathtt{s_R})_m} \end{aligned}$$

The segments comparison procedure stops when it compares the $seg(\mathtt{s_D})_m$ with the respective $seg(\mathtt{s_R})_m$, in which the segment upper bound is equal to $\mathtt{S.high_R}$. At this point, we have two possibilities:

I. if $ub_{seg(\mathtt{s}_D)_m} - lb_{seg(\mathtt{s}_D)_m} = ub_{seg(\mathtt{s}_R)_m} - lb_{seg(\mathtt{s}_R)_m}$, n corresponds to ASCII constant value to the character in $seg(\mathtt{s}_D)_{m+1}$. Formally:

$$n = [\![\mathtt{D}[lb_{seg(\mathtt{s}_D)_{m+1}}]]\!]\rho$$

II. if $ub_{seg(\mathtt{s}_D)_m} - lb_{seg(\mathtt{s}_D)_m} > ub_{seg(\mathtt{s}_R)_m} - lb_{seg(\mathtt{s}_R)_m}$, n corresponds to the ASCII constant value to the character in $seg(\mathtt{s}_D)_m$. Formally:

$$n = [\![\mathtt{D}[lb_{seg(\mathtt{s}_D)_m}]]\!]\rho$$

Notice that, in both cases, string(D) and string(R) share a prefix of length $|ub_{seg(\mathtt{s}_R)_m} - lb_{seg(\mathtt{s}_R)_1}|$.

c. *If string(D) and string(R) share a prefix of length strictly smaller than $|string(\mathtt{R})| - 1$, the considerations made in 1.**c.** hold.*

Notice that in the case in which D lexicographically follows R, $\mathtt{s}_D$ and $\mathtt{s}_R$ could not have the same number of segments.

This is suitable also to the case in which one of the two compared char array, or both, are sub-segmentations.

Example **4.15.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char D[5] = ''aaa'';
/* 1: */ char R[5] = ''abb'';
/* 2: */ int n = strcmp(D,R);
/* 3: */ return 0;
/* 4: */ }
```

The strings of interest of the two char arrays, D and R, declared in the program above, exist and correspond to $string(\mathtt{D}) = <\mathtt{a\ a\ a\ \backslash 0}>$ and $string(\mathtt{R}) = <\mathtt{a\ b\ b\ \backslash 0}>$. From the Definition 4.15, since D precedes R in lexicographic order, we know that $\mathfrak{S}[\![\mathbf{strcmp(}\mathtt{D},\mathtt{R}\mathbf{)}]\!]\sigma = -1$.

Considering the Algorithm 4.3 and the Lemma 4.2, we notice that $string(\mathtt{D})$ and $string(\mathtt{R})$ share a prefix of length strictly smaller than $|string(\mathtt{D})| - 1$. Given: $\mathrm{Seg}(\mathtt{s_D}) = \{seg(\mathtt{s_D})_1 = (0,\text{‘}\mathtt{a}\text{’},3)\}$ and $\mathrm{Seg}(\mathtt{s_R}) = \{seg(\mathtt{s_R})_1 = (0,\text{‘}\mathtt{a}\text{’},1), seg(\mathtt{s_R})_2 = (1,\text{‘}\mathtt{b}\text{’},3)\}$, we initially compare the first segment of $\mathtt{s_D}$ with the first segment of $\mathtt{s_R}$. Since $\mathrm{Seg}(\mathtt{s_D})$ has just one element, we have that:

- $x_{seg(\mathtt{s_D})_1} = x_{seg(\mathtt{s_R})_1}$
- $ub_{seg(\mathtt{s_D})_1} - lb_{seg(\mathtt{s_D})_1} > ub_{seg(\mathtt{s_R})_1} - lb_{seg(\mathtt{s_R})_1}$

The bounds differences of the first compared segments do not coincide and n is equal to the difference between the ASCII constant value to the character in $seg(\mathtt{s_D})_1$ and the ASCII constant value to the character in $seg(\mathtt{s_R})_2$, that is: $n = 97 - 98 = -1$. So, `D` lexicographically precedes `R`, as stated before, and the strings of interest of the two program char arrays share a prefix of length equal to $min(ub_{seg(\mathtt{s_D})_1} - lb_{seg(\mathtt{s_D})_1}, ub_{seg(\mathtt{s_R})_1} - lb_{seg(\mathtt{s_R})_1}) = 1$ (case 1.**c.** of Lemma 4.2). The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `D:p1 = split(D):` $(\{0\}$ ‘a’ $\{3\}, \emptyset)$
- `D:p2 = split(D):` $(\{0\}$ ‘a’ $\{3\},\emptyset) \wedge$
  `R:p2 = split(R):` $(\{0\}$ ‘a’ $\{1\}$ ‘b’ $\{3\}, \emptyset)$
- `D:p3 = split(D):` $(\{0\}$ ‘a’ $\{3\},\emptyset) \wedge$
  `R:p3 = split(R):` $(\{0\}$ ‘a’ $\{1\}$ ‘b’ $\{3\}, \emptyset) \wedge$
  `n.p3:` -1
- `D:p3 = D:p4` $\wedge$ `R:p3 = R:p4` $\wedge$ `n.p3 = n.p4`

$\triangle$

Another possible string compare statement implementation, from a static analysis point of view, can be made using the concepts of *suffix array* and *longest common prefix array* [7]. The latter is an auxiliary data structure to the suffix array introduced in the string character statement section 4.2.4. Below, we provide a refined definition of the longest common prefix suffix domain [7] that inherits the features of our char array representation.

Definition 4.16 (*longest common prefix array - a refinement*)**.** Consider a char array `str` with associated concrete value and splitting and consider $string(\mathtt{str}) \neq \mathtt{undef}$.

Let $string(\mathtt{str})[i, [\![\mathtt{S.high_{str}}]\!]\rho]$ be the sub-string of the string of interest of `str` ranging from $[\![\mathtt{S.low_{str}}]\!]\rho \leqslant i \leqslant [\![\mathtt{S.high_{str}}]\!]\rho$ to

$[\![\mathtt{S.high_{str}}]\!]\rho$ and let `Pos` be the suffix array of $\mathit{string}(\mathtt{str})$, as presented in the Definition 4.14. Then the *lcp* array `H` is an array of size $|\mathit{string}(\mathtt{str})|$ storing in each element k, a value in $\mathbb{Z} \cup \{\bot\}$, where $\mathtt{H}[0] = \bot$ and for all $k \in [1, [\![\mathtt{S.high_{str}}]\!]\rho]$, $\mathtt{H}[k] = \mathit{lcp}(\mathit{string}(\mathtt{str})[\mathtt{Pos}[k-1], [\![\mathtt{S.high_{str}}]\!]\rho], \mathit{string}(\mathtt{str})[\mathtt{Pos}[k], [\![\mathtt{S.high_{str}}]\!]\rho])$ is the length of the longest common prefix between the lexicographically ith smallest suffix of $\mathit{string}(\mathtt{str})$ and its predecessor in the suffix array.

◊

Example 4.16. Recovering the Example 4.13 we have that the suffix array `Pos` of the string of interest of `C` is:

k	0	1	2	3	4	5	6
$\mathtt{Pos}[k]$	6	5	4	1	0	3	2

Then the *lcp* array `H` is constructed by comparing lexicographically consecutive suffixes to determine their longest common prefix:

k	0	1	2	3	4	5	6
$\mathtt{H}[k]$	$\bot$	0	0	1	0	0	1

For instance, $\mathtt{H}[3] = 1$ as the longest common prefix shared by the suffixes $\mathtt{Pos}[2] = <\mathtt{i\ e\ \backslash 0}>$ and $\mathtt{Pos}[3] = <\mathtt{i\ n\ n\ i\ e\ \backslash 0}>$ that is the string $<\mathtt{i}>$ whose length is equal to 1.

△

Definition 4.17 (*glue operator*). Consider two char arrays `str1` and `str2` with $\mathit{string}(\mathtt{str1}) \neq \mathtt{undef}$ and $\mathit{string}(\mathtt{str2}) \neq \mathtt{undef}$. The operator $\oplus$ glues the strings of interest of `str1` and `str2`, $\mathtt{G}_{(\mathtt{str1},\mathtt{str2})}$, maintaining the terminating null character between them as a separator.

◊

Example 4.17. Let `D` and `R` be two char arrays such that:

- $\mathtt{D} : <\mathtt{m\ i\ n\ n\ i\ e\ \backslash 0\ x\ x\ x}>$

  $\mathit{string}(\mathtt{D}) = <\mathtt{m\ i\ n\ n\ i\ e\ \backslash 0}> \wedge |\mathit{string}(\mathtt{D})| = 7$
- $\mathtt{R} : <\mathtt{m\ o\ u\ s\ e\ \backslash 0\ y\ y}>$

  $\mathit{string}(\mathtt{R}) = <\mathtt{m\ o\ u\ s\ e\ \backslash 0}> \wedge |\mathit{string}(\mathtt{R})| = 6$

Then the glue between $\mathit{string}(\mathtt{D})$ and $\mathit{string}(\mathtt{R})$ is:

$$\mathsf{G}_{(\mathtt{D},\mathtt{R})} = \mathit{string}(\mathtt{D}) \oplus \mathit{string}(\mathtt{R}) = < \mathtt{m\ i\ n\ n\ i\ e\ \backslash 0\ \ m\ o\ u\ s\ e\ \backslash 0} >$$

$$|\mathsf{G}_{(\mathtt{D},\mathtt{R})}| = |\mathit{string}(\mathtt{D})| + |\mathit{string}(\mathtt{R})|.$$

△

Algorithm 4.4. (*m-string compare function - suffix version*).

```
Given:

  • str1, str2:  two char arrays
  • split(str1) and split(str2)

if $\mathtt{s}_{\mathtt{str1}} \neq \emptyset$ and $\mathtt{s}_{\mathtt{str2}} \neq \emptyset$
  compute:
    $\mathtt{G}_{(\mathtt{str1},\mathtt{str2})}$, $\mathtt{Pos}(\mathtt{G}_{(\mathtt{str1},\mathtt{str2})})$ and $\mathtt{H}(\mathtt{Pos}(\mathtt{G}_{(\mathtt{str1},\mathtt{str2})}))$
  then
    given $\mathtt{i} \neq \mathtt{j}$ compare:
      $\mathtt{Pos}(\mathtt{G}_{(\mathtt{str1},\mathtt{str2})})[\mathtt{i}] \rightarrow$
        $\mathtt{suffix}(\mathtt{G}_{(\mathtt{str1},\mathtt{str2})})[0, |\mathtt{string}(\mathtt{str1})| + |\mathtt{string}(\mathtt{str2})| - 1]$
      and
      $\mathtt{Pos}(\mathtt{G}_{(\mathtt{str1},\mathtt{str2})})[\mathtt{j}] \rightarrow$
        $\mathtt{suffix}(\mathtt{G}_{(\mathtt{str1},\mathtt{str2})})[|\mathtt{string}(\mathtt{str1})|, |\mathtt{string}(\mathtt{str1})| +$
          $|\mathtt{string}(\mathtt{str2})| - 1]$
    return n and $\mathtt{lcp}(\mathtt{Pos}(\mathtt{G}_{(\mathtt{str1},\mathtt{str2})})[\mathtt{i}], \mathtt{Pos}(\mathtt{G}_{(\mathtt{str1},\mathtt{str2})})[\mathtt{j}])$
else STOP!
```

□

From now on, we will refer to the result of the Algorithm 4.4 as $n = \mathit{Alg}_2(\mathrm{str1}, \mathrm{str2})$, where str1 and str2 are the compared char arrays.

Lemma 4.3. *Given two char arrays, $\mathtt{D}$ and $\mathtt{R}$, with $\mathit{string}(\mathtt{D}) \neq$ **`undef`** and $\mathit{string}(\mathtt{R}) \neq$ **`undef`**, let $n = \mathit{Alg}_2(\mathtt{D}, \mathtt{R})$, Pos be the suffix array of $\mathsf{G}_{(\mathtt{D},\mathtt{R})}$, that is the glue between the strings of interest of $\mathtt{D}$ and $\mathtt{R}$, and let H be the longest common prefix array of Pos.*

Then:

- $n = -1 \Leftrightarrow \mathtt{D} \prec \mathtt{R}$

  $\mathtt{D}$ lexicographically precedes $\mathtt{R}$ if there exist k_1, k_2 such that $k_1 < k_2$ and:

  - $\mathtt{Pos}[k_1] = \mathsf{G}_{(\mathtt{D},\mathtt{R})}[0, |string(\mathtt{D})| + |string(\mathtt{R})| - 1]$
  - $\mathtt{Pos}[k_2] = \mathsf{G}_{(\mathtt{D},\mathtt{R})}[|string(\mathtt{D})|, |string(\mathtt{D})| + |string(\mathtt{R})| - 1]$

  $\mathtt{D}$ and $\mathtt{R}$ share a prefix if, given $k_2 - k_1 = m$, $\mathtt{H}[k_1 + 1] \neq 0 > \cdots > \mathtt{H}[k_1 + m] \neq 0$. In this particular case $\mathtt{H}[k_1 + m]$ $(= \mathtt{H}[k_2])$ also corresponds to the longest common prefix between $\mathtt{Pos}[k_1]$ and $Pos[k_2]$.

- $n = 0 \Leftrightarrow \mathtt{D} = \mathtt{R}$

  $\mathtt{D}$ and $\mathtt{R}$ are lexicographically equal if $|string(\mathtt{D})| = |string(\mathtt{R})|$ and if there exist k_1, k_2 such that $k_1 = k_2 - 1$ and:

  - $\mathtt{Pos}[k_1] = \mathsf{G}_{(\mathtt{D},\mathtt{R})}[|string(\mathtt{D})|, |string(\mathtt{D})| + |string(\mathtt{R})| - 1]$
  - $\mathtt{Pos}[k_2] = \mathsf{G}_{(\mathtt{D},\mathtt{R})}[0, |string(\mathtt{D})| + |string(\mathtt{R})| - 1]$

  $\mathtt{H}[k_2] = lcp(\mathtt{Pos}[k_1], \mathtt{Pos}[k_2]) = |string(\mathtt{D})| (= |string(\mathtt{R})|)$

- $n = 1 \Leftrightarrow \mathtt{D} \succ \mathtt{R}$

  $\mathtt{D}$ lexicographically follows $\mathtt{R}$ if there exist k_1, k_2 such that $k_1 < k_2$ and:

  - $\mathtt{Pos}[k_1] = \mathsf{G}_{(\mathtt{D},\mathtt{R})}[|string(\mathtt{D})|, |string(\mathtt{D})| + |string(\mathtt{R})| - 1]$
  - $\mathtt{Pos}[k_2] = \mathsf{G}_{(\mathtt{D},\mathtt{R})}[0, |string(\mathtt{D})| + |string(\mathtt{R})| - 1]$

  $\mathtt{D}$ and $\mathtt{R}$ share a prefix if, given $k_2 - k_1 = m$, $\mathtt{H}[k_1 + 1] \neq 0 > \cdots > \mathtt{H}[k_1 + m] \neq 0$. In this particular case $\mathtt{H}[k_1 + m]$ $(= \mathtt{H}[k_2])$ also corresponds to the longest common prefix between $\mathtt{Pos}[k_1]$ and $Pos[k_2]$.

EXAMPLE **4.18.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char D[5] = ‘‘aaa’’;
/* 1: */ char R[5] = ``abb'';
/* 2: */ int n = strcmp(D,R);
/* 3: */ return 0;
/* 4: */ }
```

Given the C program of the Example 4.15, the strings of interest of the two char arrays, D and R, exist and correspond to $string(\texttt{D}) = <$ `a a a \0` $>$ and $string(\texttt{R}) = <$ `a b b \0` $>$. Furthermore, from the Definition 4.15, since D precedes R in lexicographic order (as before), we know that $\mathfrak{S}[\![\textbf{strcmp}(\texttt{D},\texttt{R})]\!]\sigma = -1$.

Consider now the concatenation between the strings of interest of D and R: $\texttt{G}_{(\texttt{D},\texttt{R})} = string(\texttt{D}) \oplus string(\texttt{R}) = <$ `a a a \0 a b b \0` $>$. $\texttt{G}_{(\texttt{D},\texttt{R})}$ has the following suffixes:

	Suffix	i
$\texttt{G}_{(\texttt{D},\texttt{R})}[0,7]$	< `a a a \0 a b b \0` >	0
$\texttt{G}_{(\texttt{D},\texttt{R})}[1,7]$	< `a a \0 a b b \0` >	1
$\texttt{G}_{(\texttt{D},\texttt{R})}[2,7]$	< `a \0 a b b \0` >	2
$\texttt{G}_{(\texttt{D},\texttt{R})}[3,7]$	< `\0 a b b \0` >	3
$\texttt{G}_{(\texttt{D},\texttt{R})}[4,7]$	< `a b b \0` >	4
$\texttt{G}_{(\texttt{D},\texttt{R})}[5,7]$	< `b b \0` >	5
$\texttt{G}_{(\texttt{D},\texttt{R})}[6,7]$	< `b \0` >	6
$\texttt{G}_{(\texttt{D},\texttt{R})}[7,7]$	< `\0` >	7

The suffixes can be sorted in lexicographical order:

	Suffix	i
$\texttt{G}_{(\texttt{D},\texttt{R})}[7,7]$	< `\0` >	7
$\texttt{G}_{(\texttt{D},\texttt{R})}[3,7]$	< `\0 a b b \0` >	3
$\texttt{G}_{(\texttt{D},\texttt{R})}[2,7]$	< `a \0 a b b` >	2
$\texttt{G}_{(\texttt{D},\texttt{R})}[1,7]$	< `a a \0 a b b \0` >	1
$\texttt{G}_{(\texttt{D},\texttt{R})}[0,7]$	< `a a a \0 a b b \0` >	0
$\texttt{G}_{(\texttt{D},\texttt{R})}[4,7]$	< `a b b \0` >	4
$\texttt{G}_{(\texttt{D},\texttt{R})}[6,7]$	< `b \0` >	6
$\texttt{G}_{(\texttt{D},\texttt{R})}[5,7]$	< `b b \0` >	5

The suffix array Pos of $\texttt{G}_{(\texttt{D},\texttt{R})}$ contains the starting positions of these suffixes:

k	0	1	2	3	4	5	6	7
Pos[k]	7	3	2	1	0	4	6	5

Then the *lcp* array `H` is constructed:

k	0	1	2	3	4	5	6	7
`H`$[k]$	$\perp$	1	0	1	2	1	0	1

Taking into account the Algorithm 4.4 and the Lemma 4.3, we have that, there exist $(k_1 = 4) < (k_2 = 5)$ such that:

- $Pos[4] = \mathtt{G}_{(\mathtt{D},\mathtt{R})}[0, 7]$
- $Pos[5] = \mathtt{G}_{(\mathtt{D},\mathtt{R})}[4, 7]$
- $5 - 4 = 1$ then $\mathtt{H}[5] = 1$

So, `D` $\prec$ `R` in lexicographical order (as stated before) and they share a prefix of length equal to 1. The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `D:p1 = split(D):` $(\{0\}$ ‘`a`’ $\{3\}, \emptyset)$
- `D:p2 = split(D):` $(\{0\}$ ‘`a`’ $\{3\},\emptyset) \wedge$
  `R:p2 = split(R):` $(\{0\}$ ‘`a`’ $\{1\}$ ‘`b`’ $\{3\}, \emptyset)$
- `D:p3 = split(D):` $(\{0\}$ ‘`a`’ $\{3\},\emptyset) \wedge$
  `R:p3 = split(R):` $(\{0\}$ ‘`a`’ $\{1\}$ ‘`b`’ $\{3\}, \emptyset) \wedge$
  `n.p3:` -1
- `D:p3 = D:p4` $\wedge$ `R:p3 = R:p4` $\wedge$ `n.p3 = n.p4`

$\triangle$

Theorem 4.1. *Given two char arrays, D and R, let:*

- $n_1 = Alg_1(D, R)$
- $n_2 = Alg_2(D, R)$

Then: $n_1 = 0 \Leftrightarrow n_2 = 0$ *and if* $n_1 \neq 0$, $\dfrac{n_1}{|n_1|} = n_2$.

Proof. by Lemma 4.2 and Lemma 4.3.

$\triangledown$

EXAMPLE **4.19.** Let n_1 be the result of the Example 4.15 and n_2 be the result of the Example 4.18. Since $n_1 \neq 0$, $\frac{n_1}{|n_1|} = n_2 = \frac{-1}{|-1|} = -1$ and the Theorem 4.1 holds.

$\triangle$

4.2.6 The string modification statement

strmdf(str,c,i) is a C function opportunely constructed that, given a char array and a specified position, substitutes the character in that position with another one. The formal declaration for strmdf() function is:

```
char *strmdf(char *str, int c, int i);
```

str is the C string to be modified, c (converted to a char) is the character to be inserted in str and i is the position in str of the character to be replaced with c. Implementation:

```
char *strmdf(char *str, int c, int i)
{
        for(k = 0; k < length; k++) {
           if(str[k] == '\0') {
              str[i] = (char)c;
           }
        }
        return (char *)str;
}
```

where "length" corresponds to the length of the str char array. strmdf() return a pointer to the first cell of the modified str. In the case in which $\mathit{string}(\text{str}) = \texttt{undef}$, our string modification statement returns an error.

Notice that the subsequent analysis will be performed on static arrays.

Definition 4.18. Consider a char array `str`, its concrete representation $\mathit{str} = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$ and $\mathit{codom}(S) = \{(i, v) \mid i \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)\}$. We define $m : \mathit{codom}(S) \times \wp(\mathbb{N}, \text{ASCII}_{ch}) \to \mathit{codom}(S')$ as the function that maps the value of the couple (i, v) to the new value (i, v') and returns the set $\mathit{codom}(S)$ with the appropriate modifications.

$\Diamond$

EXAMPLE **4.20.** Given the char array `C`: $<$ `m i n n i e` $\backslash 0>$

- $\texttt{C} = (\rho, \texttt{C.low}, \texttt{C.high}, C, \mathsf{T}_{\texttt{C}})$

  $[\![\texttt{C.low}]\!]\rho = 0$

$$[\![\texttt{C.high}]\!]\rho = 7$$
$$C = \{(0,\text{‘}\texttt{m}\text{’}), (1,\text{‘}\texttt{i}\text{’}), (2,\text{‘}\texttt{n}\text{’}), (3,\text{‘}\texttt{n}\text{’}), (4,\text{‘}\texttt{i}\text{’}), (5,\text{‘}\texttt{e}\text{’}), (6,\text{‘}\texttt{\textbackslash 0}\text{’})\}$$
$$\mathsf{T}_{\texttt{C}} = \{6\}$$

$$m(C, \{(1,\text{‘}\texttt{m}\text{’})\}) = \{(0,\text{‘}\texttt{m}\text{’}), (1,\text{‘}\texttt{m}\text{’}), (2,\text{‘}\texttt{n}\text{’}), (3,\text{‘}\texttt{n}\text{’}), (4,\text{‘}\texttt{i}\text{’}), (5,\text{‘}\texttt{e}\text{’}), (6,\text{‘}\texttt{\textbackslash 0}\text{’})\}$$
$$m(C, \{(1,\text{‘}\texttt{m}\text{’}), (2,\text{‘}\texttt{m}\text{’})\}) = \{(0,\text{‘}\texttt{m}\text{’}), (1,\text{‘}\texttt{m}\text{’}), (2,\text{‘}\texttt{m}\text{’}), (3,\text{‘}\texttt{n}\text{’}), (4,\text{‘}\texttt{i}\text{’}), (5,\text{‘}\texttt{e}\text{’}), (6,\text{‘}\texttt{\textbackslash 0}\text{’})\}$$

△

Definition 4.19 (*semantics of the string modification function*)**.** Let `str` be a char array declared in a program (`str` $\in$ ACVar), with concrete value represented by the quintuple $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$ ($str \in$ ACVal) and $string(\texttt{str}) \neq \texttt{undef}$. Given an index $\texttt{j} \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)$ and a generic character `x`, then:

$$\mathfrak{S}[\![\textbf{strmdf(}\texttt{str,x,j}\textbf{)}]\!]\sigma = \sigma' \in \text{ACState}$$

1. $\texttt{x} = \text{‘}\texttt{\textbackslash 0}\text{’} \wedge \texttt{j} \in \mathsf{T}_{\texttt{str}}$

   - $\sigma'(\texttt{str}) = (\rho, \texttt{str.low}', \texttt{str.high}', S', \mathsf{T}'_{\texttt{str}})$

     $\texttt{str.low}' = \texttt{str.low}$

     $\texttt{str.high}' = \texttt{str.high}$

     S': $\forall i \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)$
     $\quad \texttt{str}'[i] \to (i, v') \mid (i, v') \in m(S, \{(\texttt{j},\text{‘}\texttt{\textbackslash 0}\text{’})\})$

     $\mathsf{T}'_{\texttt{str}} = \mathsf{T}_{\texttt{str}}$

     $min(\mathsf{T}'_{\texttt{str}}) = min(\mathsf{T}_{\texttt{str}})$

   - $\sigma'[\texttt{s}_{\texttt{str}}] = \sigma[\texttt{s}_{\texttt{str}}]$

2. $\texttt{x} = \text{‘}\texttt{\textbackslash 0}\text{’} \wedge \texttt{j} \notin \mathsf{T}_{\texttt{str}} \wedge \texttt{j} < min(\mathsf{T}_{\texttt{str}})$

   - $\sigma'(\texttt{str}) = (\rho, \texttt{str.low}', \texttt{str.high}', S', \mathsf{T}'_{\texttt{str}})$

     $\texttt{str.low}' = \texttt{str.low}$

     $\texttt{str.high}' = \texttt{str.high}$

     S': $\forall i \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)$
     $\quad \texttt{str}'[i] \to (i, v') \mid (i, v') \in m(S, \{(\texttt{j},\text{‘}\texttt{\textbackslash 0}\text{’})\})$

     $\mathsf{T}'_{\texttt{str}} = \texttt{j} \cup \mathsf{T}_{\texttt{str}}$

     $min(\mathsf{T}'_{\texttt{str}}) = \texttt{j}$

- $\sigma'[\mathtt{s_{str}}] = \sigma[\mathtt{s_{str}}](i, j)$

  $i = [\![\mathtt{S.low_{str}}]\!]\rho$
  j corresponds to the index specified in the strmdf() statement.

3. $\mathtt{x} = `\backslash 0\text{'} \wedge \mathtt{j} \notin \mathsf{T}_{\mathtt{str}} \wedge \mathtt{j} > min(\mathsf{T}_{\mathtt{str}})$

   - $\sigma'(\mathtt{str}) = (\rho, \mathtt{str.low}', \mathtt{str.high}', S', \mathsf{T'}_{\mathtt{str}})$

     $\mathtt{str.low'} = \mathtt{str.low}$
     $\mathtt{str.high'} = \mathtt{str.high}$
     S': $\forall i \in [[\![\mathtt{str.low}]\!]\rho, [\![\mathtt{str.high}]\!]\rho)$
     $\quad \mathtt{str'}[i] \rightarrow (i, v') \mid (i, v') \in m(S, \{(\mathtt{j}, `\backslash 0\text{'})\})$
     $\mathsf{T'}_{\mathtt{str}} = \mathtt{j} \cup \mathsf{T}_{\mathtt{str}}$
     $min(\mathsf{T'}_{\mathtt{str}}) = min(\mathsf{T}_{\mathtt{str}})$

   - $\sigma'[\mathtt{s_{str}}] = \sigma[\mathtt{s_{str}}]$

4. $\mathtt{x} \neq `\backslash 0\text{'} \wedge \mathtt{j} \in \mathsf{T}_{\mathtt{str}} \wedge \mathtt{j} = min(\mathsf{T}_{\mathtt{str}})$

   If $\mathsf{T}_{\mathtt{str}}$ has only one element, then:

   - $\sigma'(\mathtt{str}) = (\rho, \mathtt{str.low}', \mathtt{str.high}', S', \mathsf{T'}_{\mathtt{str}})$

     $\mathtt{str.low'} = \mathtt{str.low}$
     $\mathtt{str.high'} = \mathtt{str.high}$
     S': $\forall i \in [[\![\mathtt{str.low}]\!]\rho, [\![\mathtt{str.high}]\!]\rho)$
     $\quad \mathtt{str'}[i] \rightarrow (i, v') \mid (i, v') \in m(S, \{(\mathtt{j}, \mathtt{x})\})$
     $\mathsf{T'}_{\mathtt{str}} = \emptyset$
     $min(\mathsf{T'}_{\mathtt{str}}) = \nexists$

   - $\sigma'[\mathtt{s_{str}}] = \emptyset$

     Observe that the resulting value does not represent a string and the system may raise a type error warning.

   If $\mathsf{T}_{\mathtt{str}}$ has more than one element, then:

   - $\sigma'(\mathtt{str}) = (\rho, \mathtt{str.low}', \mathtt{str.high}', S', \mathsf{T'}_{\mathtt{str}})$

     $\mathtt{str.low'} = \mathtt{str.low}$

$\texttt{str.high'} = \texttt{str.high}$
S': $\forall i \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)$
  $\texttt{str'}[i] \to (i, v') \mid (i, v') \in m(S, \{(\texttt{j}, \texttt{x})\})$
$\mathsf{T'}_{\texttt{str}} = \mathsf{T}_{\texttt{str}} \setminus \{\texttt{j}\}$
$min(\mathsf{T'}_{\texttt{str}}) = min(\mathsf{T}_{\texttt{str}} \setminus \{\texttt{j}\})$

- $\sigma'[\texttt{s}_{\texttt{str}}] = \sigma[\texttt{str}_{|(\texttt{j},v) \to (\texttt{j},\texttt{x})}](i, j)$

  $i = [\![\texttt{S.low}_{\texttt{str}}]\!]\rho$
  $j = min(\mathsf{T}_{\texttt{str}} \setminus \{\texttt{j}\})$

5. ‘x’ $\neq$ ‘\0’ $\wedge$ $\texttt{j} \in \mathsf{T}_{\texttt{str}} \wedge \texttt{j} > min(\mathsf{T}_{\texttt{str}})$

- $\sigma'(\texttt{str}) = (\rho, \texttt{str.low'}, \texttt{str.high'}, S', \mathsf{T'}_{\texttt{str}})$

  $\texttt{str.low'} = \texttt{str.low}$
  $\texttt{str.high'} = \texttt{str.high}$
  S': $\forall i \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)$
    $\texttt{str'}[i] \to (i, v') \mid (i, v') \in m(S, \{(\texttt{j}, \texttt{x})\})$
  $\mathsf{T'}_{\texttt{str}} = \mathsf{T}_{\texttt{str}} \setminus \{\texttt{j}\}$
  $min(\mathsf{T'}_{\texttt{str}}) = min(\mathsf{T}_{\texttt{str}})$

- $\sigma'[\texttt{s}_{\texttt{str}}] = \sigma[\texttt{s}_{\texttt{str}}]$

6. ‘x’ $\neq$ ‘\0’ $\wedge$ $\texttt{j} \notin \mathsf{T}_{\texttt{str}} \wedge \texttt{j} < min(\mathsf{T}_{\texttt{str}})$

- $\sigma'(\texttt{str}) = (\rho, \texttt{str.low'}, \texttt{str.high'}, S', \mathsf{T'}_{\texttt{str}})$

  $\texttt{str.low'} = \texttt{str.low}$
  $\texttt{str.high'} = \texttt{str.high}$
  S': $\forall i \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)$
    $\texttt{str'}[i] \to (i, v') \mid (i, v') \in m(S, \{(\texttt{j}, \texttt{x})\})$
  $\mathsf{T'}_{\texttt{str}} = \mathsf{T}_{\texttt{str}}$
  $min(\mathsf{T'}_{\texttt{str}}) = min(\mathsf{T}_{\texttt{str}})$

- $\sigma'[\texttt{s}_{\texttt{str}}] = \sigma[\texttt{s}_{\texttt{str}_{|(\texttt{j},v) \to (\texttt{j},\texttt{x})}}]$

7. ‘x’ $\neq$ ‘\0’ $\wedge$ $\texttt{j} \notin \mathsf{T}_{\texttt{str}} \wedge \texttt{j} > min(\mathsf{T}_{\texttt{str}})$

- $\sigma'(\texttt{str}) = (\rho, \texttt{str.low'}, \texttt{str.high'}, S', \mathsf{T'}_{\texttt{str}})$

$$\begin{aligned}
&\texttt{str.low}' = \texttt{str.low}\\
&\texttt{str.high}' = \texttt{str.high}\\
&S'\text{: } \forall i \in [[\![\texttt{str.low}]\!]\rho, [\![\texttt{str.high}]\!]\rho)\\
&\qquad \texttt{str}'[i] \to (i, v') \mid (i, v') \in m(S, \{(\texttt{j}, \texttt{x})\})\\
&\mathsf{T}'_{\texttt{str}} = \mathsf{T}_{\texttt{str}}\\
&min(\mathsf{T}'_{\texttt{str}}) = min(\mathsf{T}_{\texttt{str}})
\end{aligned}$$

- $\sigma'[\mathsf{s}_{\texttt{str}}] = \sigma[\mathsf{s}_{\texttt{str}}]$

◊

Example **4.21.** Given the program:

```
#include <stdio.h>
#include <string.h>
        int main() {
/* 0: */ char str[10] = ‘‘aabbcc’’;
/* 1: */ strmdf(str,‘b’,1);
/* 2: */ return 0;
/* 3: */ }
```

The concrete value of the char array `str` at the program point `1` is given by the quintuple $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$. Entering into the detail, we have that:

- $str = (\rho, \texttt{str.low}, \texttt{str.high}, S, \mathsf{T}_{\texttt{str}})$

$$\begin{aligned}
&[\![\texttt{str.low}]\!]\rho = 0\\
&[\![\texttt{str.high}]\!]\rho = 10\\
&codom(S) = \{(0,\text{‘}\texttt{a}\text{’}), (1,\text{‘}\texttt{a}\text{’}), (2,\text{‘}\texttt{b}\text{’}), (3,\text{‘}\texttt{b}\text{’}), (4,\text{‘}\texttt{c}\text{’}), (5,\text{‘}\texttt{c}\text{’}),\\
&\qquad\qquad (6,\text{‘}\backslash\texttt{0}\text{’}), (7,\texttt{T}), (8,\texttt{T}), (9,\texttt{T})\}\\
&\mathsf{T}_{\texttt{str}} = \{6\}\\
&min(\mathsf{T}_{\texttt{str}}) = 6
\end{aligned}$$

$string(\texttt{str}) = < \texttt{a a b b c c} \ \backslash 0 >$

At the program point `2` we obtain that $\mathfrak{S}[\![\textbf{strmdf}(\texttt{str}, \texttt{b}, 1)]\!]\sigma = \sigma'$, where:

- $\sigma'(\texttt{str}) = (\rho, \texttt{str.low}', \texttt{str.high}', S', \mathsf{T}'_{\texttt{str}})$

$$\llbracket \texttt{str.low'} \rrbracket \rho = 0$$
$$\llbracket \texttt{str.high'} \rrbracket \rho = 10$$
$$S' = m(S, \{(1, \texttt{b})\})$$
$$\mathsf{T'}_{\texttt{str}} = \mathsf{T}_{\texttt{str}}$$
$$min(\mathsf{T'}_{\texttt{str}}) = min(\mathsf{T}_{\texttt{str}})$$

- $\sigma'[\mathtt{s}_{\texttt{str}}] = \{0\}$ ‘a’ $\{1\}$ ‘b’ $\{4\}$ ‘c’ $\{6\}$

At the end of the analysis we have the certainty that the program array is a char array containing a string, as presented in the Definition 4.19 (case 6). The char array M-String segmentation analysis, applied on the program above, produces the following abstract predicates:

- `str:p1 = split(str):` $\big(\{0\}$ ‘a’ $\{2\}$ ‘b’ $\{4\}$ ‘c’ $\{6\}, \{7\}$ `T` $\{10\}\big)$
  Check($string$(`str`)) = TRUE
- `str:p2 = split(str):` $\big(\{0\}$ ‘a’ $\{1\}$ ‘b’ $\{4\}$ ‘c’ $\{6\}, \{7\}$ `T` $\{10\}\big)$
- `str:p3 = str:p2`

△

Chapter 5

M-String library

The main C header file string.h functions have been re-implemented and collected in the M-String library, exploiting the split char array segmentation and the char array semantics presented in Section 4.1 and the string.h statements semantics defined in Section 4.2, in order to test the correctness of our *ad hoc* char array representation.

The M-String library C code is presented in the Appendix A

5.1 Header files

The M-String library header files contain the prototypes of the library functions and the definitions of the types that the library user can use. We defined three header files:

1. my_array.h

   The definition of the `my_array` type is presented. Furthermore, the functions prototypes `initArray` and `fillArray` are introduced.

2. m-string.h

   The definitions of the `quintuple`, `segment` and `m_split` types are presented. Furthermore, the functions prototypes `f_rho`, `m_string`, `split`, `m_strcpy`, `m_strcat`, `m_strlen`, `m_strchr`, `m_strcmp` and `m_strmdf` are introduced.

   Notice that the `m_string` function returns the quintuple of a given `my_array` type, that is the concrete value of a C program char array, defined in 4.1.1. The `split` function returns the splitting of a given quintuple type, that is the abstract representation of a C program char array, presented in the Definition 4.2. For the quintuple type and for the f_rho and split outputs, a print prototype function is provided.

3. suffix.h

   The functions prototypes SuffixArray, m_strchr_suffix and m_strcmp_suffix are presented.

For all the dynamic structures, a free() prototype function is declared. The code of the provided header files is available in Section A.1 of the Appendix.

5.2 Body files

The M-String library body files contain the body of the functions and the basic types definition used by the functions. In agreement with the presented header files, we have defined three body files:

1. my_array.c

   The body of the initArray and fillArray functions is presented.

2. m-string.c

   The body of the f_rho, m_string, split, m_strcpy, m_strcat, m_strlen, m_strchr, m_strcmp, m_strmdf and of other useful functions (like the print functions) is defined.

   Notice that the m_strcpy, m_strcat and the m_strmdf functions have as input two quintuple types and they exploit the (index,value) pairs to compute the output. The m_strlen function has as input a quintuple type and it exploits the T set (the fifth quintuple parameter) to compute the output. The m_strchr function has as input a quintuple type and an int type (converted to a char), the m_strcmp has as input two quintuple types; both the m_strchr and the m_strcmp exploit the notion of segment, presented in the Definition 4.6, to compute the output.

3. suffix.c

   The body of the SuffixArray, m_strchr_suffix and m_strcmp_suffix functions is presented. The sorting algorithm used to sort the suffix array is the *bubble sort* [22].

Searching for the string character and the string compare functions results using the suffix array is a positive attempt to connect the segmentation representation of a char array to the main structure of the bioinformatics analysis. Furthermore, opportunely manipulating the longest common prefix array structure we are able to determine if two compared strings share a prefix and how it is long.

For all the dynamic structures, the body of the `free()` functions is declared. The code of the provided body files is available in Section A.2 of the Appendix.

Chapter 6

Results

In this chapter we compare our *ad hoc* char array representation to the one introduced by Cousot in [1], the advantages and disadvantages of our strings abstract domain and the possible practical applications, by discussing some preliminary experimental results.

6.1 Comparisons

The M-String Segmentation adds precision to the static analysis of C strings with respect to the general C-Segmentation framework applied on C char arrays, in terms of information loss.

We introduce an important formalization, relating to our M-String abstract representation.

Definition 6.1 (*segmentation mapping*)**.** Let ACVal, presented in the sub-section 4.1.1, be our concrete domain and let M-String be our abstract domain. Then:

$$f_1 : \text{ACVal} \rightarrow \text{M-String}$$

$$f_2 : \text{M-String} \rightarrow \text{ACVal}$$

f_1 and f_2 are two monotonic functions between partial orders, (ACVal, $\subseteq$) and (M-String,$\sqsubseteq$). Let `str` be a char array in C, and given $\sigma(\texttt{str}) \in$ ACVal and $\mathit{split}(\texttt{str}) \in$ M-String, the two functions are defined by:

- $f_1(\sigma(\texttt{str})) = \mathit{split}(\texttt{str})$
- $f_2(\mathit{split}(\texttt{str})) = \sigma(\texttt{str})$

◊

We may define a projection operator π, from the M-String segmentation to the Cousot segmentation, that merges the two split parameters into a single segmentation, proving the soundness and the precision of our C strings abstract domain.

Definition 6.2 (*projection operator*)**.** Given the M-String char array segmentation abstract domain and given the Cousot segmentation (C-Seg $\equiv \overline{\mathcal{A}}$) abstract domain then, π is the projection operator, such that π : M-String $\rightarrow$ C-Seg. Formally, let `str` be a program char array, then:

- $\pi((\emptyset, \{\texttt{str.low}\} \ \ldots \ \{\texttt{str.high}\}?)) =$
  $\{\texttt{str.low}\} \ \ldots \ \{\texttt{str.high}\}?$
- $\pi((\{\texttt{str.low}\}_{\sqcup}, \{\texttt{str.low+1}\} \ \ldots \ \{\texttt{str.high}\}?)) =$
  $\{\texttt{str.low}\}$ '\0' $\{\texttt{str.low+1}\} \ \ldots \ \{\texttt{str.high}\}?$
- $\pi((\{\texttt{str.low}\} \ \ldots \ \{\texttt{str.high-1}\}, \emptyset)) =$
  $\{\texttt{str.low}\} \ \ldots \ \{\texttt{str.high-1}\}$ '\0' $\{\texttt{str.high}\}$
- $\pi((\{\texttt{str.low}\} \ \ldots \ \{\texttt{i}\}, \{\texttt{i+1}\} \ \ldots \ \{\texttt{str.high}\})) =$
  $\{\texttt{str.low}\} \ \ldots \ \{\texttt{i}\}$ '\0' $\{\texttt{i+1}\} \ \ldots \ \{\texttt{str.high}\}$

◊

Lemma 6.1. *Consider the segmentation unification presented in the sub-section 3.2.6, the splitting unification introduced in Definition 4.3 and the projection operator defined in Definition 6.2. Let* `str1` *and* `str2` *be two char arrays, then:* $\pi(unify(split(\texttt{str1}), split(\texttt{str2}))) \subseteq unify(\pi(split(\texttt{str1})), \pi(split(\texttt{str2})))$.

The M-String abstract domain is a more precise representation of a C char array than the C-Segmentation abstract domain.

Lemma 6.2. *Let* `str` *be a C char array,* α_{Seg} *be the function defined in 3.1,* f_1 *be the function defined in 6.1 and given the projection operator defined in 6.2 then,* $\pi(f_1(\sigma(\texttt{str}))) \subseteq \alpha_{Seg}(\sigma(\texttt{str}))$.

Theorem 6.1 (*correctness*)**.** *The M-String segmentation semantics is a sound refinement of the Cousot parametric segmentation semantics.*

Proof: by applying Lemma 6.2 to the semantics of each pair of corresponding functions defined on the abstract domains C-Seg and M-String.

▽

6.2 Advantages and disadvantages

The M-String segmentation is a refined abstract domain for static analysis of C programs. In particular, given a C program managing strings, the M-String representation is able to identify, at each program point, the correctness of a string, highlighting the so-called *string of interest* of a char array. With the M-String abstract domain we can approximate the important information related to a string managed as a char array, without information loss.

On the other end, the M-String representation is an *ad hoc* C strings abstraction, and as such, limited to C programs. Appropriate extensions are required in order to detect the correctness of strings managed by other programming language.

6.3 Preliminary experimental results

We now introduce some possible applications of our char array representation.

6.3.1 Italian fiscal code

The Italian fiscal code is similar to a Social Security Number in the United States or the National Insurance Number in the United Kingdom. The Italian fiscal code is an alphanumeric code of 16 characters and uniquely identifying individuals in the health system, or natural persons who act as parties in private contracts [14].

Given a person, the first three letters of his Italian fiscal code belong to the surname, the second three letters belong to the first name; then we find the last two birth year digits, a letter associated to the month of birth, the two birthday digits (if the person is a woman, 40 is added), one letter and three numbers corresponding to the town of birth (notice that each single Italian town - comune - has its own pre-determined code) and lastly, a check character.

Example **6.1.** Consider these generality:

- **Name:** Martina
- **Surname:** Olliaro
- **Birthday:** December 31, 1991
- **Birth town:** Novara, Italy

The corresponding fiscal code is:

LLRMTN91T71F952L

$\triangle$

Notice that, in general, in a fiscal code we have six characters, two numbers, one character, two numbers, one character, three numbers and one character.

Consider now the *alphanumeric domain* ($\mathtt{AN}$), where $\mathtt{AN} = \{\perp, c, n, \perp\}$ and $\prec = \{(\perp, c), (\perp, n), (c, \top), (n, \top)\}$, where c stands for character (a character is any element belonging to the considered alphabet) and n stands for number (a number is any digit belonging to the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$) and assume to have a C program that manages fiscal codes as strings. We may abstract the array elements with the $\mathtt{AN}$ domain and represents the results with the M-String segmentation, as usual.

Example **6.2.** Let $\mathtt{F}$ be a char array containing the fiscal code presented in the Example 6.1, such that:

$$\mathtt{F} = < \mathtt{L\ L\ R\ M\ T\ N\ 9\ 1\ T\ 7\ 1\ F\ 9\ 5\ 2\ L\ \backslash 0} >$$

It is possible abstract each array element $\mathtt{AN}(\llbracket \mathtt{F}[i] \rrbracket \bar{\rho})$, $\forall i \in [0, 16]$ $(\equiv \mathtt{F}^{\bar{\rho}_{\mathtt{AN}}})$, using the *alphanumeric domain.*

$$\mathtt{F}^{\bar{\rho}_{\mathtt{AN}}} = < c\ c\ c\ c\ c\ c\ n\ n\ c\ n\ n\ c\ n\ n\ n\ c\ \perp >$$

Notice that the terminating null character is abstracted by the symbol $\perp$, since it does not represent nor a character nor a number. We consider it as the terminating character of the char array $\mathtt{F}^{\bar{\rho}_{\mathtt{AN}}}$, whose elements were abstracted, inheriting the properties in the M-String representation.

At this point we can represent the array $\mathtt{F}^{\bar{\rho}_{\mathtt{AN}}}$ with the appropriate M-String segmentation. In particular, the concrete value of $\mathtt{F}^{\bar{\rho}_{\mathtt{AN}}}$ is given by the quintuple $f^{\bar{\rho}_{\mathtt{AN}}} = (\rho, 0, 17, F^{\bar{\rho}_{\mathtt{AN}}}, \{16\})$ and its splitting is defined as follows:

$$split(\mathtt{F}^{\bar{\rho}_{\mathtt{AN}}}) = (\{\mathtt{0}\}\ c\ \{\mathtt{6}\}\ n\ \{\mathtt{8}\}\ c\ \{\mathtt{9}\}\ n\ \{\mathtt{11}\}\ c\ \{\mathtt{12}\}\ n\ \{\mathtt{15}\}\ c\ \{\mathtt{16}\}, \emptyset)$$

△

It is important to note that the representation given in the Example 6.2 for a specific fiscal code is, to all effects, equal for any fiscal code. So, combining the *alphanumeric abstract domain* and the M-String representation on C programs that manage fiscal codes as char arrays, we are able to track the correctness of the information that flows in those programs. Furthermore we are able to identify cases of "homocode - omocodia", or when two people share the same personal data and consequently they have the same fiscal code. It is possible replace one or more of the seven code numbers, starting with the one on the right, with corresponding letters in order to avoid "fiscal twins". In these particular cases, the corresponding split segmentation will opportunely change.

It is necessary define a way to distinguish between a case of homocode and an error in a fiscal code definition.

6.3.2 Italian car number plates

The current numbering process for Italian car number plates requires that the same should be composed by two letters, three number and two letters again [15]. So, assuming to have a C program that manage car licence plates as char arrays, we can track the correctness of the information that flows in that program by combining the *alphanumeric abstract domain* and the M-String representation, as proposed in the sub-section 6.3.1. In particular, the string of interest of a char array, `str`, which contains a generic current Italian car number plates plus the terminating null character, can be abstracted as follows: $\mathtt{s}_{\mathtt{str}^{\bar{\rho}_{\mathtt{AN}}}} = \{\mathtt{0}\}\ c\ \{\mathtt{2}\}\ n\ \{\mathtt{5}\}\ c\ \{\mathtt{7}\}$.

6.3.3 Product identifier code - Google Merchant Center

Google Merchant Center is a tool that lets you upload your store and product data to Google and make them available on Google Shopping and other Google services [16].

Product identifier codes are not limited to the `id[id]` attribute to define the product sold on the global marketplace. Common product identifier codes include GTIN (Global Trade Item Number) `gtin[gtin]`, MPN (Manufacturer Part Number) `mpn[mpn]`, and brand names `brand [brand]`. There exist different types of Global Trade Item Number of different lengths, like the UCP (Universal Product Code), the EAN (European Article Number), the ISBN (International Standard Book Number) and so on. Therefore, in general, given a C char array, `str`, containing a product identifier code plus the terminating null character and combining the *alphanumeric abstract domain* with the M-String representation, we have that the string of interest of `str`, or the product Id, is as follows: $\mathtt{s}_{\mathtt{str}^{\bar{\rho}_{\mathtt{AN}}}} = \mathrm{B}_1\ n\ \mathrm{B}_2\ c\ \mathrm{B}_3\ \top\ \mathrm{B}_4$. Furthermore, if we consider the reduced product of the *prefix* [11] and the *alphanumeric* domains, combined with the M-String representation we are able to be more precise in detecting information and an abstracted product Id appears, for example, as follows: $\mathtt{s}_{\mathtt{str}^{\bar{\rho}_{\mathtt{AN}}}} = \mathrm{B}_1\ (s_1 s_2^*, n)\ \mathrm{B}_2\ (s_i s_{i+1}^*, c)\ \mathrm{B}_3\ (s_n\ s_{n+1}^*, \top)\ \mathrm{B}_4$, where s denotes an alphanumeric element of a product Id field (or segment).

6.3.4 Text analysis

Text plagiarism: consider a text, contained in a C char array, if we perform an abstraction only on the non-significant words, like the conjunctions, we obtain a text segmentation that highlights the position of these words. If we perform this analysis on two different texts and assuming to have an algorithm able to match the bounds of the segments of equally non-significant words of the two considered texts, we may suppose an application able to detect the text plagiarism.

DNA sequencing: consider a sequence of DNA, contained in a C char array, if we perform an abstraction only on particular sub-sequences of characters, we obtain a DNA sequence segmentation that highlights the presence of specific patterns in it. If we perform this analysis on two different DNA sequences and assuming to have an algorithm able to match the bounds of the abstract segments of these patterns, we may suppose an application able to identify and to take trace of patterns similarities between DNA strings.

Chapter 7

Conclusions

We designed M-String, a refinement of the segmentation array domain for string analysis. The proposed domain allows to better capture the string manipulation operators, detecting errors due to the absence of the string terminating null character in the array representation.

The M-String abstract domain is an *ad hoc* segmentation approach for strings as char array in C programming language. Given a C program managing strings, the M-String representation is able to identify, at each program point, the correctness of a string, highlighting the so-called *string of interest* of a char array. With the M-String abstract domain we can approximate the important information related to a string managed as a char array, without information loss.

7.1 Future works

- Extend the M-String Segmentation abstract domain to other programming languages and create a general method able to formalize the concept of string and to verify the correctness of strings, as program variables.

- Integrate the M-String Segmentation abstract domain, by applying composition operators, with different string abstract domains, like the character inclusion domain, the prefix-suffix domain, the bricks domain, the string set domain, the constant string domain and the string hash domain.

- Customize the M-String Segmentation abstract domain for intent string analysis in Android application.

The inter-components communication (ICC) in Android applications is through intents exchange, i.e. particular text strings that are created and broadcasted to other apps in the same environment which codify to capture them. In particular, intents strings may carry device and user sensitive data. In this context, an information leakage analysis can be based on taint analysis, in which undesired information flows are tracked together with information on the string value in Android's messaging data structure. There is the need to refine string analysis in order to improve the precision of the Android information loss detection, to determine when there is actually information leakage and what kind of string manipulations have been performed during the Android apps execution.

- Customize the M-String Segmentation abstract domain for the identification of malicious strings after the realization of a security incident as the *SQL injection* attack on data management applications. Since the *SQL injection* attack takes advantage of the bad practice in concatenating strings, meant to be used by a server database, the M-String domain could be opportunely extended in order to detect potentially SQL strings predisposed to possibly concatenations and to model them.

Bibliography

[1] Patrick Cousot, Radhia Cousot, Francesco Logozzo (2011). *A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis.* Austin, Texas (USA).

[2] Patrick Cousot (2014). *Introduction to Abstract Interpretation.* London, England.

[3] Gary A. Kildall (1973). *A unified approach to global program optimization.* Monterey, California.

[4] Patrick Cousot and Radhia Cousot (1977). *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* Grenoble, France.

[5] Gilberto Filè and Francesco Ranzato (1999). *The powerset operator on abstract interpretations.* Padova, Italy.

[6] Agostino Cortesi, Giulia Costantini and Pietro Ferrara (2013). *A Survey on Product Operators in Abstract Interpretation.* Journal reference: EPTCS 129, 2013, pp. 325-336. DOI: 10.4204/EPTCS.129.19

[7] Udi Manber and Gene Myers (1989). *Suffix arrays: A new method for on-line string searches.* Department of Computer Science, University of Arizona, Tucson (AZ).

[8] Robert Seacord (2013). *Secure Coding in C and C++ (2nd edition).* Publisher: Addison-Wesley Professional.

[9] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin (1999). *Principles of Program Analysis.* Publisher: Springer.

[10] Anders Møller and Michael I. Schwartzbach (2017). *Static Program Analysis*. Department of Computer Science, Aarhus University, Denmark.

[11] Giulia Costantini, Pietro Ferrara, Agostino Cortesi (2013). *A suite of abstract domains for static analysis of string values*. Published online in Wiley InterScience. DOI: 10.1002/spe.

[12] Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang (2017). *Combining String Abstract Domains for JavaScript Analysis: An Evaluation*. In: Legay A., Margaria T. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2017. Lecture Notes in Computer Science, vol 10205. Springer, Berlin, Heidelberg.

[13] http://clc-wiki.net/wiki/C_standard_library:string

[14] https://en.wikipedia.org/wiki/Italian_fiscal_code_card

[15] https://en.wikipedia.org/wiki/Vehicle_registration_plates_of_Italy #1994-present

[16] https://support.google.com/merchants/answer/160161?hl=it

[17] Bertrand Meyer, Jim Woodcock (2005). *Verified Software: Theories, Tools, Experiments*. First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions.

[18] Patrick Cousot. *Abstract Interpretation Based Formal Methods and Future Challenges*. École normale supérieure, Département d'informatique, 45 rue d'Ulm, 75230 Paris cedex 05, France.

[19] Agostino Cortesi (2016). Software Correctness, Security and Reliability course's material. Ca' Foscari University, Venice.

[20] Magnus Madsen and Esben Andreasen (2014). *String analysis for dynamic field access*. In A. Cohen, editor, Compiler Construction, volume 8409 of Lecture Notes in Computer Science, pages 197-217. Springer.

[21] Patrick Cousot and Radhia Cousot (1992). *Abstract Interpretation Framework.* This article is reprinted from the "Special Issue on Abstract Interpretation" of: The Journal of Logic and Computation, Volume 2, Number 4, Pages 511–547. © Oxford University Press.

[22] Robert Sedgewick (1990). *Algorithms in C.* © Addison-Wesley Publishing Company, Inc.

Appendix A

M-String library - Implementation

The code of the M-String library, in C, is provided.

A.1 The library header files

```
        /* ------------- MY_ARRAY.H -------------- */
typedef struct stmy_array {
  char *text;
  int counter;
  int dim;
} my_array;
my_array *initArray(int dim);
my_array *fillArray(char *text, int counter, int dim);
void freeArray(my_array *C);
        /* ------------- M-STRING.H -------------- */
typedef struct stindex_value {
  int index;
  char value;
} index_value;
        /* ---------------------------------------- */
typedef struct stquintuple {
  char *(*rho) (struct stquintuple q);
  int lower_bound;
  int upper_bound;
  index_value *IV;
  int *T;
} quintuple;
        /* --------------------------------------- */
typedef struct stsegment {
  int lb;
  char c;
  int ub;
  } segment;
        /* --------------------------------------- */
typedef struct stm_split{
  int ssize;
  int nssize;
  segment *s;
  segment *ns;
} m_split;
        /* --------------------------------------- */
char *f_rho(quintuple q);
void print_rho(const quintuple q);
quintuple m_string(my_array *C);
void freeM_string(quintuple *q);
        /* --------------------------------------- */
m_split *split(quintuple q);
void freeSplit(m_split *split);
void print_split(m_split *split);
        /* --------------------------------------- */
void print_quintuple(const quintuple q);
int m_strlen(const quintuple q);
void m_strcpy(quintuple *dest, const quintuple src);
```

```
quintuple *m_strcat(quintuple *dest, const quintuple src);

quintuple *splitTOquintuple(const m_split *split, int index);

quintuple *m_strchr(const quintuple q, int c);

int m_strcmp(const quintuple q1, const quintuple q2);

void m_strmdf(quintuple *q, char c, int j);

        /* -------------- SUFFIX.H ---------------- */

int *SuffixArray(const quintuple q);

quintuple *m_strchr_suffix (const quintuple q, int c);

int m_strcmp_suffix(const quintuple q1, const quintuple q2,
        int *LCPlen);
```

A.2 The library body files

```
        /* ------------- MY_ARRAY.C -------------- */

#include <stdio.h>

#include <stdlib.h>

#include "my_array.h"

        /* --------------------------------------- */

my_array *initArray(int dim) {

  int i;

  my_array *newArray = malloc(sizeof(my_array));

  newArray -> text = malloc(dim * sizeof(char));

  newArray -> dim = dim;

  newArray -> counter = 0;

  for(i = 0; i < dim; i++) {

    newArray -> text[i] = '-';

  }

  return newArray;

}

        /* --------------------------------------- */

my_array *fillArray(char *text, int counter, int dim) {

  int i;

  my_array *C = initArray(dim);
  C -> counter = counter;
  for(i = 0; i < counter; i++) {
    C -> text[i] = text[i];
  }
  return C;
}
        /* -------------------------------------- */
void freeArray(my_array *C) {
  free(C->text);
  free(C);
}
        /* ------------- M-STRING.C -------------- */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "my_array.h"
#include "m-string.h"

const char endchar = '$';
const char emptyset = '@';
const int ERR_NO_INTEREST_STRING = -INT_MAX;
        /* --------- the rho environment --------- */
char *f_rho(quintuple q){
  int i = 0;
  char *s = NULL;
  if (q.T != NULL) {
    s = malloc((q.T[0]-q.lower_bound+1) * sizeof(char));
    for(i = q.lower_bound; i <= q.T[0]; i++) {
      s[i] = q.IV[i-q.lower_bound].value;
    }
  }
  return s;
}
```

```
        /* ---- the char array concrete value ---- */

        /* ------------ the quintuple ------------ */

quintuple m_string(my_array *C) {
  int i;
  int count = 0;
  quintuple q;
  q.rho = f_rho;
  q.lower_bound = 0;
  q.upper_bound = C -> dim;
  q.T = NULL;
  q.IV = malloc(C -> dim * sizeof(index_value));
  for(i = 0; i < q.upper_bound; i++) {
    q.IV[i].index = i;
    q.IV[i].value = C -> text[i];
    if( (C -> text[i] == '\0') ) {
      if(q.T == NULL) {
        q.T = malloc(sizeof(int));
      }
      else {
        q.T = realloc(q.T, (count+1) * sizeof(int));
      }
      q.T[count] = i;
      count++;
    }
  }
  return q;
}
        /* --------------------------------------- */
void freeM_string(quintuple *q) {
  if (q != NULL) {
    free(q -> T);
    free(q -> IV);
  }
}
        /* --------------------------------------- */
int upper_bound(const index_value *IV, char c, int lb, int
    limit){
  int i = lb + 1;
  while ((IV[i].value == c) && (i < limit)) {
    i++;
  }
  return i;
}
        /* ---- the char array abstract value ---- */
        /* ----------- the splitting ------------- */
m_split *split(quintuple q) {
  int i = 0;
  int k = 0;
  int j;
  int lb_ns;
  m_split *split = NULL;
  split = malloc(sizeof(m_split));
  split -> s = NULL;
  split -> ns = NULL;
  split -> ssize = 0;
  split -> nssize = 0;
  if(q.T != NULL){
    while (i + q.lower_bound < q.T[0]) {
      if (split -> s == NULL) {
        split -> s = malloc(sizeof(segment));
      }
      else {
        split -> s = realloc(split -> s,(k+1)*sizeof(segment));
      }
      split -> s[k].lb = i + q.lower_bound;

      split -> s[k].c = q.IV[i].value;

      split -> s[k].ub = upper_bound(q.IV, split -> s[k].c, i,
          q.upper_bound) + q.lower_bound;

      i = split -> s[k].ub - q.lower_bound;

      k++;

    }

    lb_ns = q.T[0] + 1;

  }

  else {

    lb_ns = q.lower_bound;

  }

  split -> ssize = k;

  j = lb_ns - q.lower_bound;

  k = 0;

  while (j + q.lower_bound < q.upper_bound) {

    if (split -> ns == NULL) {

      split -> ns = malloc(sizeof(segment));

    }

    else {

      split -> ns = realloc(split -> ns,(k+1)*sizeof(segment));

    }

    split -> ns[k].lb = j + q.lower_bound;

    split -> ns[k].c = q.IV[j].value;

    split -> ns[k].ub = upper_bound(q.IV, split -> ns[k].c, j,
        q.upper_bound) + q.lower_bound;

    j = split -> ns[k].ub - q.lower_bound;

    k++;

  }

  split -> nssize = k;

  return split;

}
```

```
        /* --------------------------------------- */
void freeSplit(m_split *split) {
  if (split !=NULL) {
    if (split -> s != NULL ) {
      free(split -> s);
    }
    if (split -> ns != NULL ) {
      free(split -> ns);
    }
    free(split);
  }
}
        /* ----------- the print functions ------- */
        /* ------------------ 1 ------------------ */
void print_rho(const quintuple q) {
  int i;
  char *s = q.rho(q);
  if (s != NULL) {
    for (i = q.lower_bound; i <= q.T[0]; i++) {
      if (s[i] == '\0') {
        printf("%c", endchar);
      }
      else {
        printf("%c", s[i]);
      }
    }
  }
  else {
    printf("undef");
  }
}
```

```
        /* ------------------ 2 ------------------ */
void print_quintuple(const quintuple q) {
  int i;
  int counter = 0;
  printf("\n-----------------------------------------
        ---------------------------------------\n");
  printf("\nThe string of interest of the current char array is
      :\n");
  printf("\nstring(MyArray) =  ");
  print_rho(q);
  printf("\n\nThe concrete value of the current char array is:
        \n");
  printf("\n(rho, %d, %d, f, {NULL indexes}) \n\n",
        q.lower_bound, q.upper_bound);
  printf("where: \n\n");
  printf("codom(f) = {");
  for(i = 0; i < q.upper_bound-q.lower_bound; i++){
    if(q.IV[i].value == '\0'){
      printf("(%d, %c)", q.IV[i].index, endchar);
      counter++;
    }
    else {
      printf("(%d, %c)", q.IV[i].index, q.IV[i].value);
    }
    if(i != q.upper_bound - q.lower_bound - 1){
      printf(", ");
    }
  }
  printf("} \n\n");
  printf("{NULL indexes} = {");
  if (counter != 0) {
    for (i = 0; i < counter; i++) {
      if ( i < counter-1 ) {
        printf("%d, ", q.T[i]);
      }
      else {
        printf("%d", q.T[i]);
      }
    }
  }
  printf("} \n\n");
}
        /* ------------------ 3 ------------------ */
void print_split(m_split *split){
  if (split != NULL)     {
    int i = 0;
    segment *s = split -> s;
    segment *ns = split -> ns;
    printf("The splitting of the current char array is:\n\n");
    printf("split(MyArray) = ( ");
    if (s != NULL) {
      printf("{%d} ", s[i].lb);
      if (s[0].c == '\0'){
        printf("_ ");
      }
    for (i = 0; i < split->ssize; i++) {
      if (s[i].c != '\0') {
        printf("%c {%d} ", s[i].c, s[i].ub );
      }
    }
  }
  else {
    printf("%c ", emptyset);
  }
  printf (", ");
  i = 0;
  if (ns != NULL) {
    printf("{%d} ", ns[i].lb);
      for (i = 0; i < split->nssize; i++) {
        if (ns[i].c != '\0') {
          printf("%c {%d} ", ns[i].c, ns[i].ub );
        }
        else {
          printf("%c {%d} ", endchar, ns[i].ub );
        }
      }
    }
    else {
      printf("%c ", emptyset);
    }
  }
  printf("); \n\n");
}
        /* --------- m_strcpy function ----------- */
void m_strcpy(quintuple *dest, const quintuple src){
  int i;
  int k;
  index_value *srcIV;
  m_split *split_src = split(src);
  m_split *split_dest = split(*dest);
  segment *s = split_src -> s;
  if (s != NULL) {
    if (src.T[0] <= dest -> upper_bound) {
      for (i = src.lower_bound; i <= src.T[0]; i++) {
        dest -> IV[i-src.lower_bound].index = src.IV[i-
                src.lower_bound].index+dest->lower_bound-
                        src.lower_bound;
        dest -> IV[i-src.lower_bound].value = src.IV[i-
                src.lower_bound].value;

      };

      free(dest -> T);

      dest -> T = NULL;

      k = 0;

      for (i = dest -> lower_bound; i < dest -> upper_bound; i
          ++) {

        if( dest -> IV[i-dest->lower_bound].value == '\0') {

          if (dest -> T == NULL) {

            dest -> T = malloc(sizeof(int));

          }

          else {

            dest -> T = realloc(dest -> T, (k+1)*sizeof(int));

          }

          dest -> T[k] = i + dest -> lower_bound;

          k++;

        }

      }

      dest -> rho = src.rho;

    }

    else {

      dest -> upper_bound = src.T[0] + 1 + dest -> lower_bound;

      freeM_string(dest);

      dest -> T = NULL;

      dest -> IV = malloc((dest->upper_bound-dest->lower_bound)
          *sizeof(index_value));

      for (i = src.lower_bound; i <= src.T[0]; i++){

        dest -> IV[i-src.lower_bound].value = src.IV[i-
                src.lower_bound].value;

        dest -> IV[i-src.lower_bound].index = src.IV[i-
                src.lower_bound].index + dest -> lower_bound;

      }

      dest -> T = malloc(sizeof(int));
```

```
      dest -> T[0] = src.T[0] + dest -> lower_bound;
      dest -> rho = src.rho;
    }
  }
  else {
    srcIV = malloc((src.upper_bound) * sizeof(index_value));
    for (i = src.lower_bound; i <= src.upper_bound; i++) {
      srcIV[i-src.lower_bound] = src.IV[i-src.lower_bound];
    }
    if (src.upper_bound <= dest -> upper_bound) {
      for (i = src.lower_bound; i < src.upper_bound; i++) {
        dest -> IV[i-src.lower_bound].value = src.IV[i-
                src.lower_bound].value;
        dest -> IV[i-src.lower_bound].index = src.IV[i-
                src.lower_bound].index  + dest -> lower_bound;
      }
      k = 0;
      free(dest -> T);
      dest -> T = NULL;
      for (i = dest->lower_bound; i < dest->upper_bound; i++) {
        if ( dest->IV[i-dest->lower_bound].value == '\0') {
          if (dest -> T == NULL) {
            dest -> T = malloc(sizeof(int));
          }
          else {
            dest->T=realloc(dest->T, (k+1)*sizeof(int));
          }
          dest -> T[k] = i;
          k++;
        }
      }
    }
    else {
      freeM_string(dest);
      dest -> rho = src.rho;
      dest -> upper_bound=src.upper_bound+dest -> lower_bound;
      dest -> IV=malloc(dest->upper_bound*sizeof(index_value));
      dest -> T = NULL;
      for (i=src.lower_bound; i < src.upper_bound; i++) {
            dest -> IV[i-src.lower_bound].value = srcIV[i-
                src.lower_bound].value;
            dest -> IV[i-src.lower_bound].index = srcIV[i-
                src.lower_bound].index  + dest -> lower_bound;
      }
    }
    free(srcIV);
  }
  freeSplit(split_src);
  freeSplit(split_dest);
}
        /* --------- m_strcat function ----------- */
quintuple *m_strcat(quintuple *dest, const quintuple src) {
  int i;
  int *T = NULL;
  m_split *split_src = split(src);
  m_split *split_dest = split(*dest);
  if ((split_src -> s !=NULL) && (split_dest -> s !=NULL)) {
    int dest_minT = dest -> T[0];
    int src_minT = src.T[0];
    index_value *srcIV = NULL;
    index_value *destIV = NULL;
    T = malloc(sizeof(int));
    T[0] = src_minT - src.lower_bound + dest_minT ;
    if ((src_minT + dest_minT) <= dest -> upper_bound) {
      int j;
      srcIV = malloc((src_minT-src.lower_bound+1) * sizeof(
          index_value));

      for (i = src.lower_bound; i <= src_minT; i++) {

        srcIV[i-src.lower_bound] = src.IV[i-src.lower_bound];

      }

      for (i = 0; i <= src_minT-src.lower_bound; i++) {

        dest->IV[dest_minT+i].index = srcIV[i].index +
            dest_minT + dest -> lower_bound;

        dest->IV[dest_minT+i].value = srcIV[i].value;

      };

      for (j = i+1; j < dest -> upper_bound; j++) {

        dest -> IV[j].index = j + dest -> lower_bound;

      }

      i = 1;

      while (dest->T[i-1] != 0) {

        if (dest->T[i-1] > T[0]){

          T=realloc(T, (i+1)*sizeof(int));

          T[i] = dest->T[i-1];

        }

        i++;

      }

      free(srcIV);

      free(dest->T);

      dest->T=T;

    }

    else {

      dest -> upper_bound = src_minT - src.lower_bound +
          dest_minT + 1;

      destIV = malloc((dest -> upper_bound) * sizeof (
          index_value));

      for (i = dest -> lower_bound; i <= dest_minT; i++) {

        destIV[i-dest->lower_bound].index = dest -> IV[i-dest->
            lower_bound].index;

        destIV[i-dest->lower_bound].value = dest -> IV[i-dest->
            lower_bound].value;
      }

      for (i = 0; i <= src_minT-src.lower_bound; i++) {

        destIV[dest_minT+i-dest->lower_bound].index = i +
            dest_minT;

        destIV[dest_minT+i-dest->lower_bound].value =
            src.IV[i].value;

      }

      freeM_string(dest);

      dest -> T = T;

      dest -> IV = destIV;

    }

    return dest;

  }

  if ((split_src -> s !=NULL) && (split_dest -> s == NULL)) {

    int j;

    int src_minT = src.T[0];

    index_value *destIV = NULL;

    int upper_bound = src_minT - src.lower_bound + dest ->
        upper_bound + 1;

    destIV = malloc((upper_bound)*sizeof(index_value));

    for (i=dest -> lower_bound; i < dest -> upper_bound; i++){

      destIV[i-dest->lower_bound].index = dest -> IV[i-dest->
          lower_bound].index;

      destIV[i-dest->lower_bound].value = dest -> IV[i-dest->
          lower_bound].value;

    }

    for (j = i; j <= src_minT+i-src.lower_bound; j++) {

      destIV[j-dest->lower_bound].index = j;

      destIV[j-dest->lower_bound].value = src.IV[j-i].value;

    };

    dest -> upper_bound = upper_bound;

    T = malloc(sizeof(int));

    T[0] = src_minT+i-src.lower_bound;

    freeM_string(dest);
    dest->IV = destIV;
    dest->T=T;
  }
  if ((split_src -> s == NULL) && (split_dest -> s != NULL)) {
    int j, k;
    index_value *destIV = NULL;
    int dest_minT = dest -> T[0];
    int upper_bound;
    if ((dest_minT+src.upper_bound) > dest -> upper_bound) {
      upper_bound = dest_minT+src.upper_bound;
    }
    else {
      upper_bound = dest -> upper_bound;
    }
    destIV = malloc(upper_bound * sizeof(index_value));
    for (i = dest -> lower_bound; i < dest_minT; i++) {
      destIV[i-dest->lower_bound].index = dest -> IV[i-dest->
          lower_bound].index;
      destIV[i-dest->lower_bound].value = dest -> IV[i-dest->
          lower_bound].value;
    }
    for (j = i; j < i + src.upper_bound; j++) {
      destIV[j-dest->lower_bound].index = j;
      destIV[j-dest->lower_bound].value = src.IV[j-i-
           src.lower_bound].value;
    }
    for (k = j; k < upper_bound; k++) {
      destIV[k-dest->lower_bound].index = dest -> IV[k-dest->
          lower_bound].index;
      destIV[k-dest->lower_bound].value = dest -> IV[k-dest->
          lower_bound].value;
    }
    dest -> upper_bound = upper_bound;
    freeM_string(dest);
    dest->IV = destIV;
    dest->T = NULL;
  }
  if ((split_src -> s == NULL) && (split_dest -> s == NULL)) {
    int j;
    index_value *destIV = NULL;
    int upper_bound = dest -> upper_bound + src.upper_bound;
    destIV = malloc(upper_bound * sizeof(index_value));
    for (i=dest -> lower_bound; i < dest -> upper_bound; i++){
      destIV[i-dest->lower_bound].index = dest -> IV[i-dest->
          lower_bound].index;
      destIV[i-dest->lower_bound].value = dest -> IV[i-dest->
          lower_bound].value;
    }
    for (j = i; j < i + src.upper_bound; j++) {
    destIV[j-dest->lower_bound].index = j;
    destIV[j-dest->lower_bound].value = src.IV[j-i-
        src.lower_bound].value;
    }
    dest -> upper_bound = upper_bound;
    freeM_string(dest);
    dest->IV = destIV;
    dest->T = NULL;
  }
  freeSplit(split_src);
  freeSplit(split_dest);
  return dest;
}

        /* --------- m_strlen function ----------- */
int m_strlen(const quintuple q) {
  int len;
  if(q.T != NULL) {
    len = q.T[0]-q.lower_bound; /* + 1; */
  }
  else {
    len = -1;
  }
  return len;
}
        /* --------- m_strchr function ----------- */
quintuple *splitTOquintuple(const m_split *split, int index) {
  int i,j;
  int ivcounter = 0;
  int tcounter = 0;
  quintuple *res = malloc(sizeof(quintuple));
  segment *s = split -> s;
  if (index < 0) {
    return NULL;
  }
  res -> rho = f_rho;
  res -> lower_bound = s[index].lb;
  res -> upper_bound = s[split->ssize-1].ub + 1;
  res -> IV = malloc((res -> upper_bound) * sizeof(index_value)
     );
  res -> T = NULL;
  for(i = index; i <= split -> ssize; i++) {
    for(j = 0; j < (s[i].ub - s[i].lb); j++) {
      res -> IV[ivcounter].value = s[i].c;
      res -> IV[ivcounter].index = res -> lower_bound +
          ivcounter;
      ivcounter++;
    }
  }
  res -> IV[res -> upper_bound - res -> lower_bound-1].value =
     '\0';
  res -> IV[res -> upper_bound - res -> lower_bound-1].index =
     res -> upper_bound - 1;
  for (i = res -> lower_bound; i < res -> upper_bound; i++) {
    if (res -> IV[i - res -> lower_bound].value == '\0') {
      if (res -> T == NULL) {
        res -> T = malloc(sizeof(int));
      }
      else {
        res -> T = realloc(res -> T, (tcounter+1)*sizeof(int));
      }
      res -> T[tcounter] = i ;
      tcounter++;
    }
  }
 return res;
}
        /* --------------------------------------- */
quintuple *m_strchr(const quintuple q, int c) {
  int i = 0;
  int index = -1;
  m_split *split_q = split(q);
  segment *s = split_q -> s;
  if (s != NULL) {
    for (i = 0; i < split_q -> ssize; i++) {
      if(s[i].c == (char) c) {
        index = i;
        i = split_q -> ssize;
      }
    }
    if (index > -1) {
      return splitTOquintuple(split_q, index);
    }
  }
  else {
    printf("\nError: there is no string of interest!\n");
  }
  freeSplit(split_q);
  return NULL;
}
        /* --------- m_strcmp function ----------- */
int m_strcmp(const quintuple q1, const quintuple q2) {
  int i = 0;
  m_split *split_q1 = split(q1);
  m_split *split_q2 = split(q2);
  segment *sq1 =  split_q1 -> s;
  segment *sq2 =  split_q2 -> s;
  if ((sq1 != NULL) && (sq2 != NULL)) {
    for (i = 0; i < split_q1 -> ssize; i++) {
      if (sq1[i].c == sq2[i].c) {
        if (sq1[i].ub - q1.lower_bound > sq2[i].ub -
                q2.lower_bound) {
          return (sq1[i].c - sq2[i+1].c);
        }
        if (sq1[i].ub - q1.lower_bound < sq2[i].ub -
                q2.lower_bound ) {
          return (sq1[i+1].c - sq2[i].c);
        }
      }
      else if (sq1[i].c != sq2[i].c) {
        return (sq1[i].c-sq2[i].c);
      }
    }
  }
  else {
    printf("\nError: there is no string of interest!\n");
  }
  freeSplit(split_q1);
  freeSplit(split_q2);
  return ERR_NO_INTEREST_STRING;
}
        /* --------- m_strmdf function ----------- */
void m_strmdf(quintuple *q, char c, int j) {
  int *T = NULL;
  int i;
  int k = 0;
  int limit = q -> upper_bound;
  m_split *split_q = split(*q);
  if (split_q -> s != NULL) {
    if (j >= q -> lower_bound && j < limit ) {
      q -> IV[j].value = c;
      for (i = q -> lower_bound; i < limit; i++){
        if (q -> IV[i - q->lower_bound].value == '\0') {
          if (T == NULL) {
            T = malloc (sizeof(int));
          }
          else {
            T = realloc (T, (k+1) * sizeof(int));
          };
          T[k] = q->IV[i - q->lower_bound].index;
          k++;
        }
      }
      free(q->T);
      q->T = T;
    }
  }
  else {
    printf("\nError: there is no string of interest!\n");
  }
  freeSplit (split_q);
}
        /* -------------- SUFFIX.C --------------- */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "my_array.h"
#include "m-string.h"
        /* -------------------------------------- */
int ivcmp(const index_value *IV1, const index_value *IV2){
  while (IV1 && IV2 && (IV1->value == IV2->value)) {
    IV1++;
    IV2++;
  }
  return ((const unsigned char) IV1->value - (const unsigned
      char) IV2->value);
}
        /* --------- sorting algorithm ----------- */
void BubbleSortSuffix (int *A, const quintuple q, int len) {
  int sorted = 0;
  int i;
  for (i = 0; i <= len && !sorted; i++){
    int j;
    sorted = 1;
    for (j=len; j>i; j--) {
      if (ivcmp(&q.IV[A[j-1]], &q.IV[A[j]]) > 0) {
        int temp = A[j-1];
        A[j-1] = A[j];
        A[j] = temp;
        sorted = 0;
      }
    }
  }
```

```
}
        /* --------------------------------------- */
void SortSuffix (int *A, const quintuple q, int len) {
  BubbleSortSuffix(A, q, len);
}
        /* ----------- suffix array -------------- */
int *SuffixArray(const quintuple q, int len){
  if (q.T != NULL) {
    int i;
    int *sa = malloc((len + 1) * sizeof(int));
    if (sa) {
      for ( i = 0; i <= len; i++) {
        sa[i] = i;
      }
      SortSuffix(sa, q, len);
    }
    return sa;
  }
  return NULL;
}
        /* ------ m_strchr suffix function ------- */
quintuple * m_strchr_suffix (const quintuple q, int c) {
  int *sa;
  int i;
  m_split *split_q = split(q);
  segment *s = split_q -> s;
  if ( s != NULL) {
    sa = SuffixArray(q, q.T[0]);
    if (sa != NULL) {
      for (i = 0; i <= q.T[0]; i++) {
        if (q.IV[sa[i]].value == c) {
          int j = 0;
          while ((j <= split_q -> ssize)  && (s[j].c != c)) {
            j++;
          }
          return splitTOquintuple(split_q, j);
        }
      }
    }
    free(sa);
  }
  else {
  printf("\nError: there is no string of interest!\n");
  }
  return NULL;
}
        /* ------------ glue operator ------------ */
quintuple glue (const quintuple q1, const quintuple q2) {
  quintuple glue;
  int i;
  int k = 0;
  glue.lower_bound = q1.lower_bound;
  glue.upper_bound = q1.T[0] + q2.T[0] + 2;
  glue.IV = malloc (glue.upper_bound * sizeof(index_value));
  glue.T = NULL;
  for (i = 0; i < glue.upper_bound; i++) {
    glue.IV[i].index = i;
    if (i <= q1.T[0]) {
      glue.IV[i].value = q1.IV[i].value;
    }
    else {
      glue.IV[i].value = q2.IV[i-q1.T[0]-1].value;
    }
    if (glue.IV[i].value == '\0') {
      if (glue.T == NULL) {
        glue.T = malloc(sizeof(int));
      }
      else {
        glue.T = realloc(glue.T,(k+1) * sizeof(int));
      }
      glue.T[k] = i;
      k++;
    }
  }
  return glue;
}
        /* ------- longest common prefix  -------- */
int * Compute_LCP_Array (const quintuple G, int *sa, int len){
  int i;
  int *lcp = NULL;
  if (sa != NULL) {
    lcp = malloc ((len+1) * sizeof(int));
    lcp[0] = -1;
    for (i = 1; i <= len; i++) {
      int j = 0;
      index_value *IV1 = G.IV+sa[i];
      index_value *IV2 = G.IV+sa[i-1];
      while(IV1 && IV2 && (IV1->value == IV2->value)) {
        IV1++; IV2++; j++;
      };
      lcp[i] = j;
    }
  }
  return lcp;
}
```

```
        /* ------ m_strcmp suffix function ------- */

int m_strcmp_suffix(const quintuple q1, const quintuple q2, int
     *LCPlen) {

  int i, u, v;

  int *sa, *H;

  m_split *split_q1 = split(q1);

  m_split *split_q2 = split(q2);

  if ((split_q1 -> s != NULL) && (split_q2 -> s !=NULL)) {

    int posu = 0;

    int posv = q1.T[0] + 1;

    quintuple G = glue(q1, q2);

    sa = SuffixArray(G, G.upper_bound-1);

    H = Compute_LCP_Array(G, sa, G.upper_bound-1);

    for (i = G.lower_bound; i < G.upper_bound; i++){

      if (sa[i] == posu) { u = i; }

        if (sa[i] == posv) { v = i; }

        }

        if (u < v) {

          if (u == v-1) {

            *LCPlen = H[v];

            return -1;

          }

          if (u < v-1) {

            int j = 1;

            int existLCP = 1;

            while ((j <= v-u) && (existLCP==1)) {

              if ((H[u+j-1] == 0) || (H[u+j] ==0) || (H[u+j-1]
                  < H[u+j])) {

              existLCP = 0;

            }

            j++;

          }

          if (existLCP) {
```

```
          *LCPlen = H[v];
        }
        else {
          *LCPlen = 0;
        }
        return -1;
      }
    }
    else {
      if (v == u - 1) {
        if ((H[u] == q1.T[0] + 1) && (H[u] == q2.T[0] + 1)) {
          *LCPlen = q1.T[0] + 1;
          return 0;
        }
        else {
          *LCPlen = H[u];
          return 1;
        }
      }
      else {
        int j = 1;
        int existLCP = 1;
        while ((j <= u-v) && (existLCP==1)) {
          if ((H[v+j-1] == 0) || (H[v+j] ==0) || (H[v+j-1] <
              H[v+j])) {
          existLCP = 0;
        }
        j++;
      }
      if (existLCP) {
        *LCPlen = H[u];
      }
      else {
          *LCPlen = 0;
        }
        return 1;
      }
    }
    free(sa);
    free(H);
  }
  else {
    printf("\nError: there is no string of interest!\n");
  }
  return 0;
}
```