Table of Contents

Abstract

Strings are widely used in modern programming languages. Unfortunately, the current approach leads to bug-prone software, raising the need for static analysis focusing on string values. In this thesis we design a suite of abstract semantics for strings and we discuss the tradeoff between efficiency and accuracy when using them to catch the properties of interest.

1 Introduction

Strings are nowadays widely used in programs developed using modern programming languages. First of all, the developers can query a database only submitting a string. In addition, reflection (i.e. "*the process by which a computer program can observe and modify its own structure and behavior*") strongly relies on string values. For instance, a PHP program can execute the code contained in a string (e.g. `$"$a=10"` will assign `10` to variable `$a`). Another example is the Java package `java.lang.reflect` that provides some classes in order to access information about the structure of classes at runtime. Finally, a large variety of programs manipulates strings (e.g. through concatenation) in order to build up the final output.

This approach leads to bug-prone software, especially when dealing with the input of the user. For instance, in Java the invocation `var.substring(var.indexOf('a'))` will throw an exception if the string stored in `var` does not contain any `'a'` character (as `var.indexOf('a')` would return `-1`, and so `substring` would throw an `IndexOutOfBoundsException`). Similarly, if method `replaceFirst` is called passing as first parameter a string not containing a regular expression, it would throw a `PatternSyntaxException`.

Another major concern regards security issues, in particular when using information provided by inputs of the user. For instance, consider a webpage that receives a value `id` as input, and uses it to delete the row identified by `id`. Intuitively, this would be implemented executing the string `"DELETE FROM Table WHERE ID = " + id`. But what happens if the value of `id` is `"10 OR TRUE"`? All the rows of the table would be permanently erased!

All these considerations show the need for static analysis applied to string values. The idea of static analysis is to compute an over-approximation of all the possible runtime behaviors of a program. In this way we can prove at compile time properties respected by all the possible executions. Applying it to string values may allow us to check some properties in order to reject programs that may contain some of the errors explained above.

Abstract interpretation is a mathematical framework aimed at formalizing the concept of approximation, and it has been widely applied to static analysis of programs. In this context, the first step is to define a concrete semantics that usually formalizes the runtime behaviors of a programming language. This semantics cannot be efficiently computed (e.g. because it requires to execute a program for all the possible inputs). Then abstract interpretation allows us to approximate it in order to build up an abstract semantics. The ideal goal is to obtain an abstract semantics that:

- can be efficiently computed (e.g. with quadratic/cubic complexity with regards to the number of analyzed statements);
- is precise enough in order to catch the properties of interest.

2 Preliminaries

In this chapter we describe the mathematical background used throughout the thesis. In particular, we introduce some basic notation and some well-known theoretical results on lattices and abstract interpretation theory. In addition, to better understand how abstract interpretation works, we present a brief case study.

2.1 Sets

A *set* is a collection of distinct objects. We denote sets with capital letters (i.e. $A, K, S, \ldots$) and elements with lower case characters (i.e. a). With $a \in A$ we denote the fact that a is a member of A.

We employ the usual symbols for *union* (∪), *intersection* (∩) and *complement* (\). With $A \subseteq B$ we denote the fact that A is a *subset* of B (every member of A is also member of B). The *cardinality* of a set A (i.e. the number of elements it contains) will be denoted with $|A|$.

The *power set* of a set S is the set of all subsets of S. This includes the subsets composed by all the members of S and the empty set. The power set can be written as $\mathcal{P}(S)$. Thus, $S \in \mathcal{P}(S)$ and $\emptyset \in \mathcal{P}(S)$. If a finite set S has cardinality n ($|S| = n$) then the power set has cardinality 2^n ($|\mathcal{P}(S)| = 2^n$).

Let $\mathbb{N}$ be the set of natural numbers (where $0 \in \mathbb{N}$). Let $\mathbb{Z}$ be the set of integer numbers, and let $[a \ldots b]$ be the set $\{ i \in \mathbb{Z} : i \geq a \wedge i \leq b \}$. Let $[-\infty \ldots b]$ be the set $\{ i \in \mathbb{Z} : i \leq b \}$ and $[a \ldots + \infty]$ be the set $\{ i \in \mathbb{Z} : i \geq a \}$.

Given two sets X and Y, their *Cartesian product* is denoted by $X \times Y$. This set contains all the possible pairs composed by an element in X as first component and by an element in Y as second component. Formally: $X \times Y = \{ (x, y) : x \in X \wedge y \in Y \}$.

2.2 Lists

Given a set S, a list l is a partial function $[\mathbb{N} \to S]$ such that

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

This definition implies that the domain of all non-empty lists is a segment of $\mathbb{N}$. Intuitively, a list is an ordered sequence of elements such that it is defined on the first k elements. The empty list (i.e. the list l such that $dom(l) = \emptyset$) is denoted by ϵ^{l}. Let S be a generic set of elements, we denote by $\overrightarrow{S^{+}}$ the set of all finite lists composed of elements in S.

$len : [\overrightarrow{S^{+}} \rightarrow \mathbb{N}]$ is the function that, given a list, returns its length. Formally: $len(l) = i + 1 : i \in dom(l) \wedge i + 1 \notin dom(l)$. If $l = \epsilon^{l}$, then $len(l) = 0$.

We define an operator of *truncation* on lists:

$$trunc(l, n) = \{\, [i \longmapsto s(i)] : i \in [0, n-1] \wedge n < len(l) \,\}$$

Given a list l such that $len(l) = n$, then $trunc(l, n-1)$ corresponds to the same lists deprived of its last element.

We also define an operator which, given a list, returns the same list without its first character:

$$withoutFirst(l) = \{\, [i \longmapsto s(i+1)] : i \in [0, len(l) - 2] \,\}$$

We define an operator which, given two lists l_1 and l_2 such that $len(l_1) = n$ and $len(l_2) = m$, concatenates them:

$$concatList(l_1, l_2) = \{ \left[i \longmapsto \begin{cases} l_1(i) \text{ if } i < n \\ l_2(i-n) \text{ if } i \geq n \wedge i < n + m \end{cases} \right] \}$$

2.3 Strings

A *character* is a unit of information that roughly corresponds to a grapheme, grapheme-like unit, or symbol, such as in an alphabet or syllabary in the written form of a natural language. Examples of characters include letters ($a - z, A - Z$), numerical digits ($0 - 9$), and common punctuation marks (such as '.' or '$-$'). The concept also includes control characters, which do not correspond to symbols in a particular natural language, but rather to other bits of information used to process text in one or more languages. Examples of control characters include carriage return or tab. Computers and communication equipments represent characters using an encoding that relates each character to something — an integer quantity represented by a sequence of bits, typically. This representation can be stored or transmitted through a network. Two examples of popular encodings are ASCII and the UTF-8 encoding for Unicode.

Characters are typically combined into strings. In fact, a *string* can be defined as an ordered sequence of characters that are chosen from a particular set or alphabet.

More formally, let $\mathcal{C}$ be an alphabet, a non-empty finite set. Elements of $\mathcal{C}$ are called symbols or characters. A string (or word) over $\mathcal{C}$ is any finite sequence of characters from $\mathcal{C}$. For example, if $\mathcal{C} = \{0, 1\}$, then 0101 is a string over $\mathcal{C}$.

The length of a string is the number of characters in the string (the length of the sequence) and can be any non-negative integer. The empty string is the unique string over $\mathcal{C}$ of length 0, and is denoted by ϵ. The set of all strings over $\mathcal{C}$ of length n is denoted $\mathcal{C}^n$. For example, if $\mathcal{C} = \{0, 1\}$, then $\mathcal{C}^2 = \{00, 01, 10, 11\}$. Note that $\mathcal{C}^0 = \{\epsilon\}$ for any alphabet $\mathcal{C}$.

The *Kleene star* (or Kleene operator or Kleene closure) is a unary operation, either on set of strings or on sets of symbols or characters. The application of the Kleene star to a set $\mathcal{C}$ is written as $\mathcal{C}^*$. The definition is the following:

- If $\mathcal{C}$ is a set of strings then $\mathcal{C}^*$ is defined as the smallest superset of $\mathcal{C}$ that contains the empty string and is closed under the string concatenation operation. This set can also be described as the set of strings that can be made by concatenating zero or more strings from $\mathcal{C}$.
- If $\mathcal{C}$ is a set of symbols or characters then $\mathcal{C}^*$ is the set of all strings over symbols in $\mathcal{C}$, including the empty string.

An example of Kleene star applied to set of characters:

$\{\,'a','b','c'\,\}^*$
$= \{\,\lambda, "a", "b", "c", "aa", "ab", "ac", "ba", "bb", "bc", "ca", "cb", "cc", \ldots\}$

In terms of $\mathcal{C}^n$:

$$\mathcal{C}^* = \bigcup_{n \in \mathbb{N}} \mathcal{C}^n$$

Although $\mathcal{C}^*$ itself is countably infinite, all elements of $\mathcal{C}^*$ have finite length.

Concatenation is an important binary operation on $\mathcal{C}^*$. For any two strings s and t in $\mathcal{C}^*$, their concatenation is defined as the sequence of characters in s followed by the sequence of characters in t, and is denoted st or $s + t$. Formally:

$$s + t = \left\{ \left[i \longmapsto c : \begin{cases} s(i) \text{ if } i < len(s) \\ t(i - len(s)) \text{ if } len(s) \le i < len(t) \end{cases} \right] \right\}$$

For example, if $\mathcal{C} = \{a, b, \ldots, z\}$, $s = "straw"$, and $t = "berry"$, then $st = s + t = "strawberry"$ and $ts = t + s = "berrystraw"$. String concatenation is an associative, but non-commutative operation. The empty string serves as the identity element; for any string s, $\epsilon s = s\epsilon = s$.

A string s is said to be a *substring* or *factor* of t if there exist (possibly empty) strings u and v such that $t = usv$. The relation "is a substring of" defines a partial order on $\mathcal{C}^*$, the least element of which is the empty string.

Given two strings s and t, their longest common prefix can be defined as:

$$longestCommonPrefix(s,t) = \{\,[i \longmapsto s(i)] : i < \min\big(len(s), len(t)\big) \wedge \forall j \le i : s(j) = t(j)\,\}$$

We can also define the longest common suffix between two strings:

$$longestCommonSuffix(s,t) = \{\,[i \longmapsto s(i)] : i < \min\big(len(s), len(t)\big) \wedge \forall j \ge i : s(j) = t\big(j + len(t) - len(s)\big)\,\}$$

2.4 Graphs

A *directed graph* G is a pair (N, A), where N is a finite nonempty set, and A is a relation on N (A is any subset of $N \times N$). Each element in N is called a *node*, and each pair in A is called an *arc*. The arc (n, m) *leaves* the node n and *enters* the node m. We say that n is a *predecessor* of m, and m is a *successor* of n. $PRED(n)$ denotes the set of predecessors of node n, and $SUCC(n)$ denotes the successors of node n. The *indegree* of a node n is $\#PRED(n)$, and the *outdegree* of n is $\#SUCC(n)$, where $\#S$ denotes the cardinality of a set S. The arcs entering a node n are called the *incoming* arcs of n, and the arcs leaving a node n are called the *outgoing* arcs of n. A *path* is a finite sequence of one or more nodes such that, if the sequence contains two or more nodes and is denoted by $(n_1, \dots, n_k)$ with $k \geq 2$, then $(n_i, n_{i+1}) \in A \; \forall i \in [1, k-1]$. A *simple* path is a path with distinct nodes. If $k \geq 2$, then $(n_1, \dots,, n_k)$ is a path from n_1 to n_k with length $k-1$, and n_k is said to be *accessible* from n_k. As a special case, a single node denotes a path of length 0 from itself to itself. A *cycle* is a path $(n_1, \dots, n_k)$ where $n_1 = n_k$. A *simple* cycle is an arc from a node to itself. A path is *acyclic* or *noncircular* if it does not contain a cycle.

A *directed acyclic graph* (acronym DAG) is a directed graph without any cycles. A (rooted) *tree* is a DAG satisfying the following properties:

(1) There is exactly one node, called the *root*, with indegree 0.
(2) Every node except the root has indegree 1.
(3) There is a path from the root to each node.

If (u, v) is an arc in the tree, then u is called the *parent* of v, and v is called a *son* of u. The *ancestor* and *descendant* relations are reflexive and transitive closures of the respective parent and son relations. Node n is called a *proper* ancestor (descendant) of node m iff n is an ancestor (descendant) of m and $n \neq m$. A *leaf* is a node n with outdegree 0.

2.5 Regular expressions

Regular expressions (often called *regex* or *regexp*), provide a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification. The origins of regular expressions lie in automata theory and formal language theory, fields which study models of computation (automata) and ways to describe and classify formal languages. The use of regular expressions in structured information standards for document and database modeling started in the 1960s and expanded in the 1980s.

2.5.1 Basic concepts

A regular expression, also referred to as a *pattern*, is an expression that describes a set of strings. Usually, they are used to give a concise description of a set, without having to list all its elements. We can say that a particular set of strings is “described by a certain pattern”, or, alternatively, that such pattern “matches each of the strings in the considered set”. In most formalism, if there is any regex that matches a particular set then there are an infinite number of such expressions. Most formalism provides the following three basic operations to construct regular expressions:

- **Boolean "or"**; a vertical bar separates alternatives. For example, $cat|cut$ can match "cat" or "cut".
- **Grouping**; parentheses are used to define the scope and precedence of the operators (among other uses). For example, $cat|cut$ and $c(a|u)t$ are equivalent patterns which both describe the set of "cat" and "cut".
- **Quantification**; a quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark (?), the asterisk ($*$) (derived from the Kleene star we described before), and the plus sign ($+$).

- **?** The question mark indicates there is <u>zero or one</u> occurrences of the preceding element. For example, $brea?d$ matches both "$bred$" and "$bread$".
- * The asterisk indicates there are <u>zero or more</u> occurrences of the preceding element. For example, ca^*t matches "ct", "cat", "$caat$", "$caaat$", "$caaaat$" and so on.
- **+** The plus sign indicates that there is <u>one or more</u> occurrences of the preceding element. For example, ca^+t matches "cat", "$caat$", "$caaat$", "$caaaat$" and so on, but not "ct".

With these three simple construction blocks we can build arbitrarily complex expressions, much like we can construct arithmetical expressions from numbers and the four operations ($+$, $-$, $\times$, and $\div$). For example, $H(ae?|\text{ä})ndel$ and $H(a|ae|\text{ä})ndel$ are both valid patterns which match the same three strings as $H(\text{ä}|ae?)ndel$: $\{H\text{ä}ndel, Handel, Haendel\}$.

Now we can explore the definition of regular expressions with tools coming from formal language theory.

2.5.2 Formal language theory

Regular expressions can be defined in terms of formal language theory. Regular expressions consist of constants and operators that denote sets of strings and operations over these sets, respectively. Given a finite alphabet Σ, the following constants are defined:

- (<u>empty set</u>) ∅ denoting the set ∅
- (<u>empty string</u>) ϵ denoting the "empty" string, with no characters at all
- (<u>literal character</u>) a in Σ denoting a character in the language

The following operations are defined:

- (<u>concatenation</u>) RS denoting the set $\{\alpha\beta \mid \alpha \in R \wedge \beta \in S\}$. For example $\{"a", "bc"\}\{"de", "f"\} = \{"ade", "af", "bcde", "bcf"\}$.

- (alternation) $R \mid S$ denoting the set union of R and S. For example $\{"ab","c"\} \mid \{"ab","d","ef"\} = \{"ab","c","d","ef"\}$.
- (Kleene star) R^* denoting the smallest superset of R that contains ϵ and is closed under string concatenation. This is the set of all strings that can be made by concatenating zero or more strings in R. For example, $\{"ab","c"\}^* =$ $\{\varepsilon,"ab","c","abab","abc","cab","cc","ababab","abcab",\dots\}$.

It is assumed that the Kleene star has the highest priority, then concatenation and then alternation. If there is no ambiguity, parentheses may be omitted. For example, $(ab)c$ can be written as abc, and $a|(b(c^*))$ can be written as $a|bc^*$.

Let us see some examples of regular expressions:

- $a|b^*$ denotes $\{\varepsilon, a, b, bb, bbb, \dots\}$
- $(a|b)^*$ denotes the set of all strings with no symbols other than a and b, including the empty string: $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$
- $ab^*(c|\varepsilon)$ denotes the set of strings starting with a, then zero or more bs and finally (optionally) a c: $\{a, ac, ab, abc, abb, abbc, \dots\}$

This formal definition of regular expressions is purposely parsimonious and avoids defining the redundant quantifiers $?$ and $+$, which can be expressed as follows: $a^+ = aa^*$, and $a? = (a|\epsilon)$. Sometimes the complement operator is added; R^C denotes the set of all strings over Σ^* that are not in R. In principle, the complement operator is redundant, as it can always be circumscribed by using the other operators. However, the process for computing such a representation is complex, and the result may require expressions of a size that is double exponentially larger.

As the example on $\{Händel, Handel, Haendel\}$ showed, different regular expressions can express the same set of strings. This happens because the formalism is redundant. It is possible to write an algorithm which, given two regular expressions, decides whether the described languages (set of strings) are essentially equal, reduces each expression to a minimal deterministic finite state machine, and determines whether they are isomorphic (equivalent). To what extent can this redundancy be eliminated? Does it exists an interesting subset of regular expressions that is still fully expressive? This turns out to be a surprisingly difficult problem. As simple as regular expressions are, it turns out there is no method to systematically rewrite them to some normal form.

2.5.3 Syntax

The precise syntax for regular expressions varies among tools and with context. In fact, traditional Unix regular expression syntax followed common conventions but often differed from tool to tool. The IEEE POSIX *Basic Regular Expressions* (**BRE**) standard (released alongside an alternative flavor called *Extended Regular Expressions* or ERE) was designed mostly for backward compatibility with the traditional (*Simple Regular Expression*) syntax. The BRE standard has provided a common standard which has since been adopted as the default syntax of many Unix regular expression tools. In the BRE syntax, most characters are treated as literals — they match only themselves (i.e., a matches "a"). The exceptions, listed below, are called *metacharacters* or *metasequences*.

Metacharacter	**Description**
.	Matches any single character. Within POSIX bracket expressions, the dot character matches a literal dot. For example, $a.c$ matches "abc", etc., but $[a.c]$ matches only "a", "$.$", or "c".
[ ]	A bracket expression. Matches a single character that is contained within the brackets. For example, $[abc]$ matches "a", "b", or "c". $[a-z]$ specifies a range which matches any lowercase letter from "a" to "z". These forms can be mixed: $[abcx-z]$ matches "a", "b", "c", "x", "y", or "z", as does $[a-cx-z]$. The $-$ character is treated as a literal character only if it is the last or the first character within the brackets, or if it is escaped with a backslash: $[abc-]$, $[-abc]$, or $[a\backslash-bc]$.
[^]	Matches a single character that is <u>not</u> contained within the brackets. For example, $[\hat{}abc]$ matches any character other than "a", "b", or "c". $[\hat{}a-z]$ matches any single character that is not a lowercase letter

	from "a" to "z". As above, literal characters and ranges can be mixed.
^	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
$	Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.
BRE: \(\) ERE: ( )	Defines a marked subexpression. The string matched within the parentheses can be recalled later (see the next entry, $\backslash\boldsymbol{n}$).
\n	Matches what the n^{th} marked subexpression matched, where n is a digit from 1 to 9. Some tools allow referencing more than nine capturing groups.
*	Matches the preceding element zero or more times. For example, $ab*c$ matches "ac", "abc", "$abbbc$", etc. $[xyz]*$ matches "", "x", "y", "z", "zx", "zyx", "$xyzzy$", and so on. $\backslash(ab\backslash)*$ matches "", "ab", "$abab$", "$ababab$", and so on.
BRE: \{m,n\} ERE: {m,n}	Matches the preceding element at least m and not more than n times. For example, $a\backslash\{3{,}5\backslash\}$ matches only "aaa", "$aaaa$", and "$aaaaa$". This is not found in a few, older instances of regular expressions.

Some examples of regular expressions built through this BRE syntax:

- $.at$ matches any three-character string ending with "at", including "hat", "cat", and "bat"
- $c[ua]t$ matches "cut" and "cat"
- $[\hat{}h]at$ matches all strings matched by $.at$ except "hat"
- $\hat{}[bc]at$ matches "bat" and "cat", but only at the beginning of the string or line
- $[bc]at\$$ matches "bat" and "cat", but only at the end of the string or line
- \[.\] matches "[a]" and "[b]" since the brackets are escaped

In the POSIX *Extended Regular Expression* (**ERE**) syntax, support is removed for $\backslash n$ backreferences and the following metacharacters are added:

Metacharacter	**Description**
?	Matches the preceding element zero or one time. For example, $ba?$

	matches "b" or "ba".
+	Matches the preceding element one or more times. For example, ba^+ matches "ba", "baa", "$baaa$", and so on.
|	The choice (alternation or set union) operator matches either the expression before or the expression after the operator. For example, $abc\|def$ matches "abc" or "def".

Some examples of regular expressions built through ERE syntax:

- $[hc]^+at$ matches "hat", "cat", "$hhat$", "$chat$", "$hcat$", "$ccchat$", and so on, but not "at"
- $[hc]?at$ matches "hat", "cat", and "at"
- $cat|dog$ matches "cat" or "dog"

2.5.3.1 POSIX character classes

Since many ranges of characters depend on the chosen locale setting (i.e., in some settings letters are organized as $abc\ldots zABC\ldots Z$, while in some others as $aAbBcC\ldots zZ$), the POSIX standard defines some classes or categories of characters as shown in the following table:

POSIX	**ASCII**	**Description**
[:alnum:]	[A-Za-z0-9]	Alphanumeric characters
[:word:]	[A-Za-z0-9_]	Alphanumeric characters plus "_"
[:alpha:]	[A-Za-z]	Alphabetic characters
[:blank:]	[\t]	Space and tab
[:cntrl:]	[\x00-\x1F\x7F]	Control characters
[:digit:]	[0-9]	Digits
[:graph:]	[\x21-\x7E]	Visible characters
[:lower:]	[a-z]	Lowercase letters
[:print:]	[\x20-\x7E]	Visible characters and spaces
[:punct:]	[-!"#$%&'()*+,./:;<=>?@[\\\]^_`{|}~]	Punctuation characters
[:space:]	[\t\r\n\v\f]	Whitespace characters
[:upper:]	[A-Z]	Uppercase letters
[:xdigit:]	[A-Fa-f0-9]	Hexadecimal digits

2.5.4 Patterns for non-regular languages

Many features found in modern regular expression libraries provide an expressive power that far exceeds the regular languages. For example, many implementations allow grouping sub-expressions with parentheses and recalling the value they match in the same expression (**back-references**). This means that a pattern can match strings of repeated words like "$papa$" or "$WikiWiki$", called *squares* in formal language theory. The pattern for these strings is $(.*)\backslash 1$. The language of squares is not regular, nor is it context-free. Pattern matching with an unbounded number of back references, as supported by numerous modern tools, is NP-complete.

However, many tools, libraries, and engines that provide such constructions still use the term regular expression for their patterns. This has led to a nomenclature where the term regular expression has different meanings in formal language theory and pattern matching.

2.6 Partial orders

A *partial order* is a binary relation "$\leq_S$" over a set S which is reflexive, anti-symmetric and transitive, i.e., for all a, b and c in S, we have that:

- $a \leq a$ (reflexivity);
- if $a \leq b$ and $b \leq a$ then $a = b$ (anti-symmetry);
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity).

A set with a partial order is called *partially ordered set* (also called a *poset*). We denote it by $(S, \leq_S)$. Thus, a poset consists of a set together with a binary relation that indicates that, for certain pairs of elements in the set, one of the elements precedes the other. These relations are called partial order to reflect the fact that not every pair of elements of a poset need be related; for some pairs, it may be neither element precedes the other in the poset. Partial orders generalize the more familiar total orders, in which every pair is related. A finite poset can be visualized through its *Hasse diagram*, which depicts the ordering relation between certain pairs of elements and allows one to reconstruct the whole partial order structure.

For example, the power set of $\{ x, y, z \}$ partially ordered by set inclusion, has the Hasse diagram:

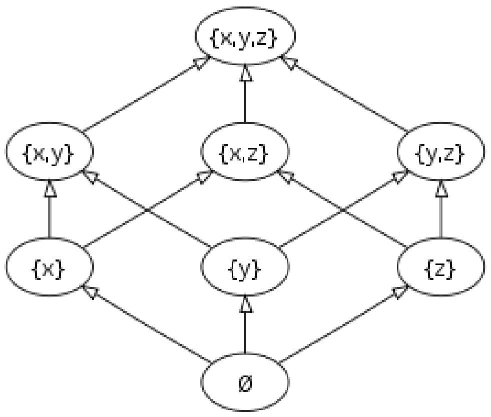


Figure 1 - Hasse diagram of {x,y,z}

The set $A = \{ 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60 \}$ of all divisors of 60, partially ordered by divisibility, has the Hasse diagram:

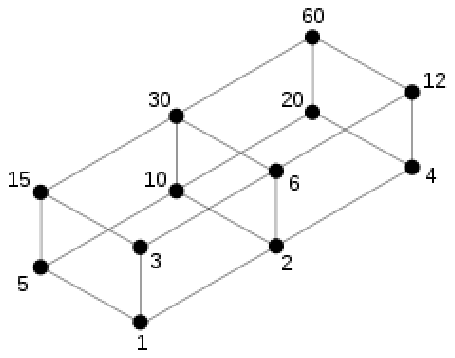


Figure 2 - Hasse diagram of divisors of 60

A poset $(S, \leq_S)$ has a *top element* $\top$ iff $\top \in S \wedge \forall s \in S : s \leq_S \top$. Dually, it has a *bottom element* $\bot$ iff $\bot \in S \wedge \forall s \in S : \bot \leq_S s$. In the example we saw before, we have:

- in the power set of $\{x, y, z\}$, $\top = \{x, y, z\}$ and $\bot = \emptyset$;
- in the divisors of 60, $\top = 60$ and $\bot = 1$.

An *upper bound* of a subset X of some partially ordered set $(S, \leq_S)$ is an element of S which is greater than or equal to every element of X. In other words: given $X \subseteq S$, $s \in S$ is an upper bound of X iff $\forall x \in X : x \leq_S s$. It is the *least upper bound* (lub) if $\forall s_1 \in S$ such that s_1 is an upper bound of X, then $s \leq_S s_1$; we denote it by $s = \sqcup X$. The term *lower bound* is defined dually as an element of S which is lesser than or equal to every element of X: $s \in S$ is a lower bound of X iff $\forall x \in X : s \leq_S x$. It is the *greatest lower bound* (glb) if $\forall s_1 \in S$ such that s_1 is a lower bound of X, then $s_1 \leq_S s$; we denote it by $s = \sqcap X$.

A *lattice* is a poset in which any two elements have a unique supremum (the elements' least upper bound also called their *join*) and an infimum (greatest lower bound, also called their *meet*). More formally, a poset $(S, \leq_S)$ is a lattice if it satisfies the following two axioms:

- existence of binary joins

    - for any two elements $a, b \in S$, the set $\{a, b\}$ has a join (also known as least upper bound or supremum);
- existence of binary meets
    - for any two elements $a, b \in S$, the set $\{a, b\}$ has a meet (also known as greatest lower bound or infimum).

A *complete lattice* is a poset such that every subset of S has a least upper bound and a greatest lower bound. A *bounded lattice* has a greatest (or maximum) and least (or minimum) element, also called top and bottom. Any lattice can be converted into a bounded lattice by adding a greatest and least element, and every non-empty finite lattice is bounded by taking the join (lub) and meet (glb) of all elements. A poset is a bounded lattice if and only if every finite set of elements (including the empty set $\emptyset$) has a lub and a glb.

A *chain* C in a poset $(S, \leq_S)$ is a subset of S such that $\forall c_1, c_2 \in C : c_1 \leq_S c_2 \vee c_2 \leq_S c_1$. An *ascending chain* is an ordered subset $\{ s_i : i \in [a \dots b]$ where $a, b \in \mathbb{N} \cup \{-\infty, +\infty\}\}$ of S such that $\forall j, k \in [a \dots b] : j \leq k \Rightarrow x_j \leq x_k$. Dually, a *descending chain* is an ordered subset of elements such that each element is less or equal than the previous ones.

A poset $(S, \leq_S)$ satisfies the *ascending chain condition* (ACC) if every ascending chain $c_1 \leq_S c_2 \leq \cdots$ of elements in S is eventually stationary, i.e. $\exists i \in \mathbb{N} : \forall j > i \; c_j = c_i$.

Given an ascending chain $d_0 \leq_S d_1 \leq_S d_2 \leq_S \dots$ in a poset $(S, \leq_S)$, a *widening operator* $\nabla : [S \to S]$ is an upper bound operator such that the chain

$$w_0 = d_0, w_1 = w_0 \nabla d_1, \dots, w_i = w_{i-1} \nabla d_i$$

is ultimately stationary, i.e. $\exists j \in \mathbb{N} : \forall k \in \mathbb{N} : k > j \Rightarrow w_j = w_k$.

2.7 Static analysis and verification

Static analysis is an automatic technique for finding out interesting properties (output values, termination, dependencies, etc.) of programs without running them.

To understand why static analysis is so important, we can cite the catastrophe of *Ariane 5*, an expendable launch system used to deliver payloads into geostationary transfer orbit or low Earth orbit. Ariane 5's first test flight on 4 June 1996 failed, with the rocket self-destructing 37 seconds after launch because of a malfunction in the control software, which was arguably one of the most expensive computer bugs in history. A data conversion from 64-bit floating point value to 16-bit signed integer value caused a processor trap (operand error) because the floating point value was too large to be represented by a 16-bit signed integer. The use of static analysis may have prevented such disaster.

The main techniques for static analysis are (Nielson, Nielson, & Hankin, 2005):

- *Abstract interpretation*
    - Since abstract interpretation is the technique that we will use throughout the thesis, we will expand this subject in the next paragraph.
- *Data-flow analysis (and Control-flow analysis)*
    - Data-flow analysis (Hecht, 1977) is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is *reaching definitions*. A simple way to perform data-flow analysis of programs is to set up data-flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it

reaches a fixpoint. This general approach was developed by Gary Kildall while teaching at the Naval Postgraduate School (Kildall, 1973).

- *Model checking*
    - Model checking (Clarke, 2008) refers to the following problem: given a finite model of a system, test automatically whether this model meets a given specification. Typically, the systems one has in mind are hardware or software systems, and the specification contains safety requirements such as the absence of inconsistent or critical states that can cause the system to crash. In order to solve such a problem algorithmically, both the model of the system and the specification are formulated in some precise mathematical language: to this end, it is formulated as a task in logic, namely to check whether a given structure satisfies a given logical formula. The concept is general and applies to all kinds of logics and suitable structures. A simple model-checking problem is verifying whether a given formula in the propositional logic is satisfied by a given structure. Model checking tools face a combinatorial blow up of the state-space (commonly known as the state explosion problem) that must be addressed to solve most real-world problems.
- *Type systems*
    - A type system may be defined as a tractable syntactic framework for classifying phrases according to the kinds of values they compute (Pierce, 2002). A type system associates types with each computed value. By examining the flow of these values, a type system attempts to prove that no type errors can occur. The type system in question determines what constitutes a type error, but a type system generally seeks to guarantee that operations expecting a certain kind of value are not used with values for which that operation makes no sense. Type safety contributes to program

correctness, but cannot guarantee it. Depending on the specific type system, a program may give the wrong result and be safely typed, producing no compiler errors. For instance, division by zero is not caught by the type checker in most programming languages; instead it is a runtime error. To prove the absence of more general defects, other kinds of formal methods are in common use.

2.8 Abstract interpretation

Abstract interpretation is a mathematical theory of approximation of semantics developed by Patrick Cousot and Rashia Cousot in 1977 (about 30 years ago).

Abstract interpretation was invented in order to deal systematically with abstractions and approximations. Abstract interpretation is seen as a general technique to reason about program semantic, at various levels of abstraction. It is successfully applied to distributed system for the verification of security and correctness of programs.

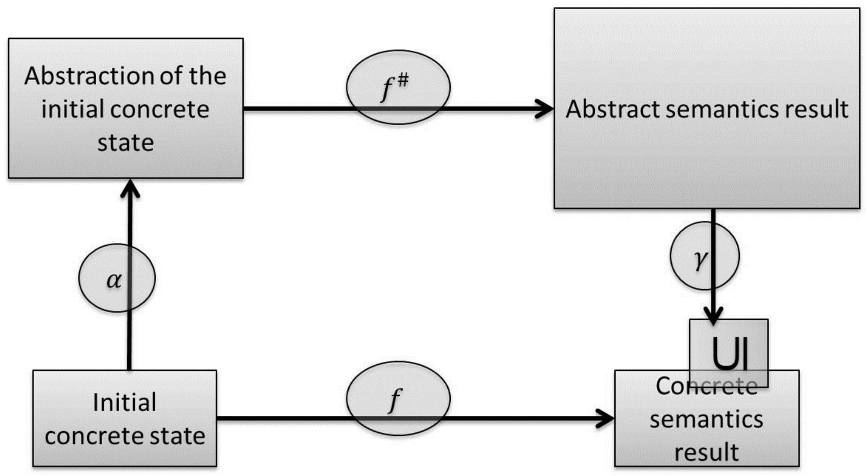
Applied to static analysis of programs, abstract interpretation allows to approximate an uncomputable concrete semantics with a computable abstract one. Approximation is required, thus making the result correct but incomplete. In fact, the inferred properties are satisfied by all the possible results of the concrete semantics, but if a property is not inferred in the abstract semantics it may still be satisfied by the concrete one. We can generalize the basic idea of abstract interpretation as follows:

- The starting point is the *concrete domain* (the formal description of a computation state) plus the *concrete semantics*, a function defined on the concrete domain, function which associates a meaning to program statements;

- Then an *abstract domain* has to be determined; an abstract domain models some properties of the concrete computations, leaving out the remaining information;
- From the abstract domain we can derive an *abstract semantics* and prove its correctness; the abstract semantics allows us to execute the program on the abstract domain in order to compute the properties modeled by such domain;
- Applying a fixpoint algorithm we can statically compute a concrete approximation of the concrete semantics.
    - If the abstract domain respects the ACC (ascending chain condition), the abstract semantics can be computed in a finite time. Otherwise we need a widening operator in order to make the analysis convergent. Thus, the use of widening on abstract domains not satisfying the ACC makes the analysis convergent still obtaining sound (even if more approximated) results.

Schematically:

$$\boxed{\boldsymbol{f \subseteq \gamma \circ f^{\#} \circ \alpha}}$$


Abstraction of the initial concrete state
$f^{\#}$
Abstract semantics result
α
γ
$\cup\!|$
Initial concrete state
f
Concrete semantics result

2.8.1 Case study – Integers sign approximation

To better understand how abstract interpretation works, we can consider a simple case study. We will simulate in a simplified environment the process that we will follow in Chapter 4, when we will define the concrete domain of strings and two possible abstract domains (together with their abstract semantics). During this case study we will also formalize the building blocks of abstract interpretation (concretization and abstraction function, Galois connection, etc.).

Let us consider a very limited language, which allow us only to multiply integers:

$$e = i \mid e * e$$

The semantics for this language can be described by a function μ defined as follows:

$$\mu : Exp \rightarrow Int$$

$$\begin{array}{rcl} \mu(i) & = & i \\ \mu(e_1 * e_2) & = & \mu(e_1) \times \mu(e_2) \end{array}$$

We can consider an abstraction of the concrete semantics based only on signs:

$$\alpha : Exp \rightarrow \{+, 0, -\}$$

$$\alpha(i) = \begin{cases} + \; if \; i > 0 \\ 0 \; if \; i = 0 \\ - \; if \; i < 0 \end{cases}$$

In this abstract semantics we approximate the actual values of the variables with their signs (plus $+$, minus $-$ or zero 0). We define the abstract version of multiplication as follows:

$$\alpha(e_1 * e_2) = \alpha(e_1) \mathbin{\overline{\times}} \alpha(e_2)$$

where the abstract operation $\overline{\times}$ is described in the following table:

$\overline{\times}$	$+$	$\mathbf{0}$	$-$
$+$	$+$	0	$-$
$\mathbf{0}$	0	0	0
$-$	$-$	0	$+$

This abstraction is correct, that is, it computes correctly the sign of expressions. Note that this abstraction does not lose any precision: to get the sign of a product, it is sufficient to know the sign of the operands. For some other operations, the abstraction may lose precision: for instance, it is impossible to know the sign of a sum whose operands are respectively positive and negative.

The function α is also called **abstraction function**. In fact, it takes as input a concrete value (an element of Exp, in our case) and it returns as output the corresponding abstract value (an element of $\{+,0-\}$). More formally, a function α is called an abstraction function if it maps an element s in the concrete set $\mathcal{D}$ to an element in the abstract set $\mathcal{A}$. The element $\alpha(s) \in \mathcal{A}$ is called the abstraction of s in $\mathcal{D}$. In our example, we have $\mathcal{D} = Exp$, $\mathcal{A} = \{+,0,-\}$.

We can consider also the opposite of the abstraction function: the **concretization function**. Formally, a function γ is called a concretization function if it maps an element a in the abstract set $\mathcal{A}$ to a set of elements in the concrete set $\mathcal{D}$. That is, the set of elements $\gamma(a) \in \mathcal{D}$ is the concretization of a in $\mathcal{A}$. In our case, we have:

$$\gamma : \{+,0,-\} \rightarrow 2^{Int}$$

$$\begin{array}{rcl} \gamma(+) & = & \{\, i : i > 0 \,\} \\ \gamma(0) & = & \{0\} \\ \gamma(-) & = & \{\, i : i < 0 \,\} \end{array}$$

The concretization function thus maps an abstract element into a *set* of concrete values. This happens because an abstract element is bound to lose some information, so it cannot map to a single concrete element.

Supposing that $\mathcal{D}$ is the concrete domain and $\mathcal{A}$ is the abstract domain, we can schematize the relationships between α, γ and μ like this:

$$\begin{tikzcd}
 & \text{A} \arrow[dd, "\gamma"] \\
\text{Exp} \arrow[ur, "\alpha"] \arrow[dr, "\mu"'] & \\
 & 2^{\text{D}}
\end{tikzcd}$$

More formally, we can also write:

$$\mu(e) \in \gamma\big(\alpha(e)\big)$$

This formula says that the exact value of an expression is contained in the concretization of the abstraction of such expression. The abstract semantics is correct, since it is an approximation of the concrete semantics.

To complicate a little our case study, let us add some operators: sign change and sum. We have to define the concrete and abstract semantics of each new operator.

$$\mu(-e) = -\mu(e)$$

$$\alpha(-e) = {=}\alpha(e)$$

$=$	$+$	$\mathbf{0}$	$-$
	$-$	0	$+$

$$\mu(e_1 + e_2) = \mu(e_1) + \mu(e_2)$$

$$\alpha(e_1 + e_2) = \alpha(e_1) \mathbin{\overline{+}} \alpha(e_2)$$

$\overline{+}$	$+$	**0**	$-$
$+$	$+$	$+$	$\top$
0	$+$	0	$-$
$-$	$\top$	$-$	$-$

Notice that, in order to define the abstract semantics of $\overline{+}$, we had to add a new abstract value (top ⊤). This happens because when we add a positive number and a negative number we cannot know the sign of the resulting value. The abstract domain must be updated:

$$\mathcal{A} = \{ +,0,-,\top\}$$

$$\gamma(\top) = Int$$

Thus, we must update the abstract semantics of all the operators:

$\overline{+}$	$+$	**0**	$-$	$\top$
$+$	$+$	$+$	$\top$	$\top$
0	$+$	0	$-$	$\top$
$-$	$\top$	$-$	$-$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\overline{-}$	$+$	**0**	$-$	$\top$
	$-$	0	$+$	$\top$

$\overline{\times}$	$+$	**0**	$-$	$\top$
$+$	$+$	0	$-$	$\top$
0	0	0	0	0
$-$	$-$	0	$+$	$\top$
$\top$	$\top$	0	$\top$	$\top$

In some cases, our abstract semantics loses some information:

$$\mu\big((1 + 2) + (-3)\big) = 0$$

$$\alpha\big((1+2)+(-3)\big) = \big(+\overline{+}\,+\big)\overline{+}(\overline{-}\,+) = +\overline{+}- = \top$$

In other cases, our abstract semantics does not lose information:

$$\mu\big((5*5)+6\big) = 31$$

$$\alpha\big((5*5)+6\big) = \big(+\,\overline{\times}\,+\big)\overline{+}(+) = +\overline{+}+ = +$$

The last addition to our domain consists in the integer division. This operator has a problem, which is the division by zero. What do we obtain when we divide a number by zero? A computation error. We must therefore introduce a new element to our abstract domain, ⊥ (bottom).

$$\mathcal{A} = \{+,0,-,\top,\bot\}$$

$$\gamma(\bot) = \emptyset$$

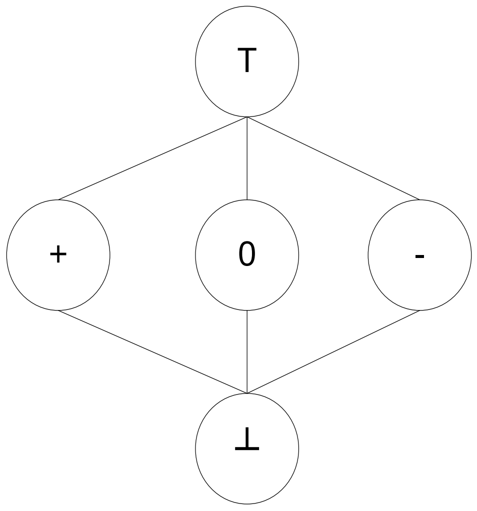
$$\bot\,\overline{+}\,x = \bot$$

$$x\,\overline{\times}\,\bot = \bot$$

$$\overline{-}\bot = \bot$$

$\overline{/}$	$+$	$\mathbf{0}$	$-$	$\top$	$\bot$
$+$	$+$	0	$-$	$\top$	$\bot$
$\mathbf{0}$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$-$	0	$+$	$\top$	$\bot$
$\top$	$\top$	0	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Our final abstract domain is thus a complete lattice:



The partial order defined on the lattice is coherent with the concretization function:

$$x \leq_{\mathcal{A}} y \Leftrightarrow \gamma(x) \subseteq \gamma(y)$$

The lattice is complete since every subset has a least upper bound and a greatest lower bound.

We can now recap what we learned about abstract interpretation during this case study. An abstract interpretation is made of:

- a concrete domain $\mathcal{D}$ and an abstract domain $\mathcal{A}$
- $\mathcal{D}$ and $\mathcal{A}$ are complete lattice; the order of the abstract lattice $\mathcal{A}$ reflects its precision (smaller equals more precise);
- a concretization and abstraction function which are monotone and form a Galois insertion
    - Given a concrete domain $(\mathcal{D}, \subseteq)$, an abstract domain $(\mathcal{A}, \leq)$, a monotone concretization function $\gamma : \mathcal{A} \rightarrow \mathcal{D}$, a monotone abstraction function $\alpha : \mathcal{D} \rightarrow \mathcal{A}$, we say that γ and α form a **Galois insertion** if they respect the following properties:

    - $\forall x \in \mathcal{D}\ \ x \subseteq \gamma(\alpha(x))$
    - $\forall y \in \mathcal{A}\ \ \alpha(\gamma(y)) = y$
- Abstract operators (functions which correctly abstract the semantics)

The critical choices when defining an abstract interpretation framework mainly regard the abstract domain to model the property of interest and the correct abstract operations.

It is important to note that we always have a tradeoff between *precision* and *efficiency*: the more precision we want, the more efficiency we are bound to lose. Usually, it is impossible to produce all the possible executions of a program. All the non-trivial properties cannot be computed. This is why we need approximation. More abstraction leads to a faster analysis, whilst less abstraction leads to a slower one. The ideal goal would therefore be an analysis that scales up and that does not produce any false alarm.

To conclude this paragraph about abstract interpretation, remember that:

- The formal **semantics** of a language provides, for all programs written in this language, a mathematical model of all possible behaviors of a computer system executing this program, in interaction with any possible environment. All interesting questions about a program semantics are undecidable. This means than no computer can always answer these questions in a finite time.
- **Abstract interpretation** is a theory of the approximation of semantics of (programming or specification) languages. It formalizes the idea that the semantics can be more or less precise according to the considered level of observation. If the approximation is coarse enough, the abstraction of a semantics yields a less precise but computable result. Because of the corresponding loss of information, not all questions can be answered, but all answers given by the effective computation of the approximate semantics are always correct.

- **Static analysis** uses abstract interpretation to derive a computable semantics from the standard semantics. Therefore computers can analyze the run-time behavior of programs and software at compile-time, that is before and without execution. This is essential, for example in critical computer systems (planes, launchers, nuclear plants) so that bugs (which sometimes escape to the perspicacity of programmers) can be detected before producing catastrophes (like in the case of Ariane-5).

3 Samples and string operators

In this chapter we present some case studies dealing with strings. From these snippets of code we will extract basic operators on strings. The syntax of the language will support all these basic operators. We will focus our attention only on the string operators. Other operators and statements (like `while`, `if`, etc) will be supported as usual in abstract interpretation. In fact, our purpose is to define an abstract domain that in the future will be plugged in an existing generic abstract analyzer. Such analyzer will take care of dealing with the programming language in general, heap structure and so on, whilst our domain should cover all (and only) the operations on strings.

3.1 Samples

Let us consider some code samples. We will refer to these samples constantly throughout the thesis. We will number them to make references easier.

3.1.1 Substrings (safe)

A first example of code dealing with substring is the following:

```
string foo1(string s)
{
      int l = s.indexOf("a");
      int r = s.indexOf("b");
      if( l >= 0 && r > l )
            return s.substring(l,r);
      else
      return s;
}
```

This example is *safe*. In fact, we return the substring between `'a'` and `'b'` only if the string contains those two characters and if the `'b'` comes after the `'a'`. We are certain that this code cannot produce exceptions or unwanted behaviors. On the other hand, we need a precise numerical domain in order to statically prove that this piece of code respects the constraints imposed by method `substring`.

3.1.2 Substrings (not safe)

Another example, similar to the previous one:

```
string foo2(string s)
{
      return s.substring( s.indexOf(“a”), s.lastIndexOf(“b”) );
}
```

This sample is not safe. In fact, we did not check anything about the presence of ‘a’ or ‘b’ in the string or about the order of such two characters in the string. The `substring` function could throw an `IndexOutOfBoundsException`.

The best representation for string would probably be something that combines inclusion of characters and ordering of them. For example, a string could be represented as “df*g*a”, meaning that the string begins with “df”, then it contains some characters, then there is a “g”, then some other characters and finally an “a”. If our abstract analysis should tell us that the string s, when it arrives as input in foo2, is always (in every execution of the program) something like “a*b”, then this example could be considered safe.

3.1.3 Constants evaluation

Let us now see samples of how we can initialize a string in the first place.

```
string s = “hello”;
string t = “ ”;
string u = “world”;
string w = s + t + u;
```

In the example above, we can see the initialization of four strings. The first three are built from constant strings. The fourth is built through strings concatenation: s concatenated to t, concatenated to u. The human can immediately see that the value of w is “hello world”. The abstract domain, unfortunately, could not be as precise. In fact, considering a representation based on character inclusion, we would have that w contains the characters that form the string “hello world” and no more.

Another example of strings built from constants:

```
int i = ...; //casual natural number
string s = i + “”;
```

In this sample the string is initialized through an integer number concatenated to the empty string. After this code we know that s contains only digits, even if we do not know which. A good (but not easy to obtain) representation for a string would be something like “{0-9}+” (resembling regular expressions).

3.1.4 SQL code

The last example of code concerns SQL queries. Imagine we would like to dynamically build a query:

```
string foo3(string t)
{
      “SELECT Name, Surname FROM People WHERE Name == ” + t;
}
```

We certainly want t to not contain spaces. If t contained spaces then it could be something like “pippo or true”, which would make us select all the rows of the database.

A safer example is the following:

```
string foo4(string s)
{
      if( !s.contains(“ ”) )
      return “SELECT Name, Surname FROM People WHERE Name == ” +
t;
```

```
      else
      {
             string t = s.substring( 0, s.indexOf(" ") );
            return "SELECT Name, Surname FROM People WHERE Name ==
" + t;
      }
}
```

We want our domain to be able to guarantee that, when we reach the building of the query, the string does not contain spaces.

3.2 Strings operators

From the examples seen in the previous paragraph, we can extract the string operators which will form our syntax. In the first example we have:

```
string foo1(string s)
{
      int l = s.indexOf("a");
      int r = s.indexOf("b");
      if( l >= 0 && r > l )
            return s.substring(l,r);
      else
      return s;
}
```

In the second one:

```
string foo2(string s)
{
      return s.substring( s.indexOf("a"), s.lastIndexOf("b") );
}
```

In the third (part one):

```
string s = “hello”;
string t = “ ”;
string u = “world”;
string w = s + t + u;
```

and in the third (part two):

```
int i = ...; //casual number
string s = i + “”;
```

And finally in the fourth:

```
string foo4(string s)
{
      if( !s.contains(“ ”) )
      return “SELECT Name, Surname FROM People WHERE Name == ” +
t;
      else
      {
             string t = s.substring( 0, s.indexOf(“ ”) );
            return “SELECT Name, Surname FROM People WHERE Name ==
” + t;
      }
}
```

Since our purpose is to tackle some real problems (and hopefully solve them), we only consider constructs seen in the samples above. Obviously there are a lot of other operators on strings out there, but for now we focus on these ones. Now we are going to summarize them and see them in greater detail:

Return value	Name	Parameters	Exceptions	Remarks
	`String`	`Char[] value`	None	`String s = “abc”` is equivalent to `char data[] = {'a', 'b', 'c'}; String str = new String(data);`
`String`	`concat`	`String str`	None	`String s = t + u;` is equivalent to `String s = t.concat(u);`
`int`	`indexOf`	`int ch`	None	Returns -1 if the character does not occur
`int`	`lastIndexOf`	`int ch`	None	Returns -1 if the character does not occur
`String`	`substring`	`int beginIndex, int endIndex`	`IndexOutOfBoundsException` if `beginIndex` is negative, or `endIndex` is larger than the length of the string, or `beginIndex` is larger than `endIndex`	
`boolean`	`contains`	`CharSequence s`	`NullPointerException` if `s` is `null`	Returns `true` if and only if the string contains the specified sequence of char values.

Table 1 - Basic string operators

Ideally, we would also like to support some other additional operators:

Return value	Name	Parameters	Exceptions	Remarks
`int`	`length`		None	Returns the length of the sequence of characters represented by the string
`boolean`	`isEmpty`		None	Returns `true` if `length = 0`, `false` otherwise
`char`	`charAt`	`int index`	`IndexOutOfBoundsException` if `index` is negative or not less than the length of the string	The first character is at index 0
`int`	`compareTo`	`String str`	None	Compares two strings lexicographically; returns 0 if the two strings are equal, a value less than 0 if the argument string is greater, a value greater than 0 if the argument string is less
`boolean`	`startsWith`	`String prefix, int toffset`	None	Tests if the substring of this string beginning at the specified index starts with the specified prefix.
`boolean`	`startsWith`	`String prefix`	None	Tests if this string starts with the specified prefix.
`boolean`	`endsWith`	`String suffix`	None	Tests if this string ends with the specified suffix.

int	indexOf	int ch, int fromIndex	None	Returns the index within the string of the first occurrence of the specified character, starting the search at the specified index (-1 if the character does not occur).
int	indexOf	String str	None	Returns the index within the string of the first occurrence of the specified substring (-1 if it does not occur).
int	lastIndexOf	String str	None	Returns the index within the string of the rightmost occurrence of the specified substring (-1 if it does not occur).
String	substring	int beginIndex	IndexOutOfBoundsException if beginIndex is negative or larger than the length of the string	
String	replace	char oldChar, char newChar	None	Returns a new string resulting from replacing all occurrences of oldChar in the string with newChar.
String	replaceFirst	String regex, String	PatternSyntaxException if the regular expression's	Replaces the first substring of the string that matches

		`replacement`	syntax is not valid	the given regular expression with the given replacement.

Table 2 - Advanced string operators

Obviously, our abstract domains will not be able to deal with all these advanced operators. We will concentrate our attention mostly on the operators seen in Table 1.

4 Concrete domain

We start by defining and formalizing our concrete domain.

Since what we want to analyze are string values, our concrete domain is simply made of strings. We already defined strings in Chapter 2: $S = K^*$, where K is our alphabet. From that definition of string we can extract the main features of a string (features we would really like to preserve in our abstract domain):

- Characters (the characters which form the string)
- Order (the characters of a string are in a precise order)
- Repetition (a character can appear zero, one or many times in the same string)
- Alphabet (the alphabet from which characters are taken: we will keep this fixed for the sake of simplicity; we consider an alphabet made by letters (“a”, “b”, “c”, …), digits (“0”, “1”, …, “9”) but also generic characters like space (“ ”), comma (“,”), dot (“.”), etc; for example, $\text{"aaa"} \in S$, $\text{"abcde"} \in S$ but also $\text{"a1,ir 13/"} \in S$)

Now we can define our lattice. We know that a string variable in our program could have different values in different executions of such program. Our goal is to *approximate* all these values (potentially infinite: think about user input) in a finite and computable manner. Our lattice will thus be made of set of strings (the powerset of S, that is the set containing all the subsets of S):

$$\boxed{\mathcal{D} = \mathcal{P}(S)}$$

Possible elements of $\mathcal{P}(S)$ are the following:

- $\emptyset$
- $\{\text{"aaa"}\}$
- $\{\text{"aaa"}, \text{"abcde"}\}$
- $\{\text{"a1,ir 13/"}\}$

- $\{$ "aaa", "abcde", "a1,ir l3/" $\}$

and so on. Note that the empty set is an element of $\mathcal{P}(S)$ (that is $\emptyset \in \mathcal{P}(S)$) but also S itself is an element of $\mathcal{P}(S)$: $S \in \mathcal{P}(S)$.

The **partial order** between elements of $\mathcal{D}$ is the set inclusion:

$$V_1 \leq_{\mathcal{D}} V_2 \iff V_1 \subseteq V_2$$

We can see that $\{$ "aaa" $\} \leq_{\mathcal{D}} \{$ "aaa", "abcde", "a1,ir l3/" $\}$, but $\{$ "aaa" $\} \nleq_{\mathcal{D}} \{$ "a1,ir l3/" $\}$ because the two sets do not have any strings in common. To understand why this order is correct, consider the two following programs:

```
//Program1
s = readInt(); // returns a strings

//Program2
s = readString();
```

The first program semantics is a subset of the second program one, since numbers can be considered as strings too ("12" $\in S$), but all the strings that could be inputted are far more than all the numbers that could be inputted. Our concrete domain respects this relationship. In fact, the approximation of the first program (let us call it P_1) is a set containing all possible strings made by digits (digits are a subset of K). The approximation of the second program (let us call it P_2) is a set containing all possible strings in general (we use the entire K). It is easy to see that P_1 is then a subset of P_2. Thus, we have $P_1 \leq_{\mathcal{D}} P_2$ which is what we said at the beginning about the semantics of the two programs.

Our **concrete lattice** (partially ordered set) is thus defined as:

$$\mathcal{L} = (\mathcal{D}, \leq_{\mathcal{D}})$$

In order for $\mathcal{L}$ to be a correct lattice, we have to satisfy two conditions:

1. Existence of binary joins, that is for any two elements $a, b \in \mathcal{L}$, they have a join (least upper bound). This is easy to prove, because the least upper bound is simply the operator of union between sets. Given $a, b \in \mathcal{L}$ the LUB of a and b is thus $a \cup b$.
2. Existence of binary meets, that is for any two elements $a, b \in \mathcal{L}$, they have a meet (greatest lower bound). This is also easy to prove, because the greatest lower bound is the operator of intersection between sets. Given $a, b \in \mathcal{L}$ the GLB of a and b is $a \cap b$.

For example, consider the two sets $A = \{\text{"aaa"}\}$ and $B = \{\text{"a1,ir l3/"}\}$. A possible upper bound of A and B is $\{\text{"aaa", "abcde", "a1,ir l3/"}\}$, although it is not the least, since it contains an element ("abcde") which was not present in any of the two sets. The least upper bound is: $\sqcup(\{A, B\}) = \{\text{"aaa", "a1,ir l3/"}\}$.

Let us make a simple and practical example about this. Imagine we have the following code:

```
if( random() > 0.5 )
      s = "a";
else
      s = "b";
```

The execution of the *if* branch can be approximated by the set $IF = \{\text{"a"}\}$, while the *else* branch can be approximated by the set $ELSE = \{\text{"b"}\}$. Since the branching condition is unpredictable at compile time, we don't know which branch will be executed. In this case, we use the least upper bound of the executions of each branch, obtaining $\sqcup(\{IF, ELSE\}) = \{\text{"a", "b"}\}$.

The least upper bound operator allows us to consider different possible executions and "merge" them together in a single approximation.

Going back to the previous example (A, B sets) we can see that the greatest lower bound is $\sqcap$ $(\{A, B\}) = \{\text{"aaa"}\} \cap \{\text{"a1,ir l3/"}\} = \emptyset$.

because the two sets have nothing in common.

Moreover, our lattice is a *bounded lattice*, that is a lattice which contains a greatest and a least element (denoted, respectively, as top and bottom). The **bottom** will be an element $\bot \in \mathcal{D}$ such that $\forall C \in \mathcal{D}, \bot \leq_{\mathcal{D}} C$. In our case, bottom is the empty set $\emptyset$. In fact, we have:

$$\emptyset \subseteq C \ \forall C \text{ (true by definition of } \emptyset\text{)}$$

$$\emptyset \subseteq C \ \forall C \in \mathcal{D} \text{ (weakening of the above condition)}$$

$$\emptyset \leq_{\mathcal{D}} C \ \forall C \in \mathcal{D} \text{ (by definition of } \leq_{\mathcal{D}}\text{)}$$

On the other hand, **top** will be an element $\top \in \mathcal{D}$ such that $\forall C \in \mathcal{D}, C \leq_{\mathcal{D}} \top$. In our case, top is the set containing all the strings S. In fact, we have:

$$C \subseteq S \ \forall C \in \mathcal{P}(S) \text{ (by definition of power-set)}$$

$$C \subseteq S \ \forall C \in \mathcal{D} \text{ (by definition of } \mathcal{D}\text{)}$$

$$C \leq_{\mathcal{D}} S \ \forall C \in \mathcal{D} \text{ (by definition of } \leq_{\mathcal{D}}\text{)}$$

This concludes the definition of our concrete domain and allows us to move on to the definition of abstract domains, in order to approximate the concrete one. The first simple domain we will build is based on the inclusion of characters.

5 Abstract domains

The questions we have to ask ourselves before starting the construction of abstract domains are the following ones: what is a string made of? What is the relevant information contained in a string? How can we approximate it in an efficient way? We already answered the first question at the beginning of the previous paragraph ("Concrete domain"). The other two questions arise from the fact that it is impossible to track precise information about *all* possible executions at compilation time. It is thus necessary to introduce some kind of approximation. Various solutions are possible, but which approximation is the best one? We want to trace information precise enough to efficiently analyze the behaviors of interest (considering the string operators we defined in section "Samples and string operators"). So, we will have to keep in mind our purpose: approximating as much as we can strings, preserving information we deem relevant (such as "the string does not contain spaces"). Obviously it is not possible to maintain all the information, since this naive approach would lead to an incomputable analysis. Imagine, for instance, a program that read a string provided by the user. If we want to take into account all the possible cases, we should consider a potentially unbounded number of inputs from the user, and, even if we bound the number of characters the user can provide, it would be impossible to compute the results in a reasonable time. So, we will have to make compromises: for example, we could consider a representation in which we maintain all the information we have about characters inclusion (which we will do in a bit) but nothing about order. This representation would behave well in programs which use string operators like `contains` (see Table 1), for example the second part of example 1.4 (SQL Code). Another possible representation keeps information about the order but not about inclusion in itself (i.e.: a string which begins with an "a" and ends with a "b": we know nothing about other characters which the string could contain, but we are sure about the first and last characters). This representation could come really useful for programs similar to examples 1.1 and 1.2 (use of `substring` operator).

Obviously, also mixed representations could be considered (i.e.: a string which begins with an “a” and then contains a “b”, an “f” and an “x”).

We will start with the first option mentioned: character inclusion and nothing else.

6 Abstract domains based on character inclusion

This abstract domain is based on the idea of identifying a string through the characters we know it surely contains/does not contain. For instance, in example 1.4 (SQL Code) it would be optimal to know that the input string for the query does not contain spaces. In other cases it could be interesting to know if the string contains a specific character (in example 1.2 the `substring` operator would throw an `IndexOutOfBoundsException` if `‘a’` wasn’t in the string, because `s.indexOf(‘a’)` would return -1, so we would like the certainty of `‘a’` being in `s`).

Of course, we could be more precise than this and include information about alternative possibilities. For example, after the following code (in the hypothesis that beforehand we don’t know anything about `‘a’`, `‘b’`, `’c’` in the string `s`):

```
...
if(...)
       s += “ab”;
else
       s += “bc”;
...
```

the string `s` contains for sure the character `‘b’` (and our abstract domain certainly should include this information). But we could also store the fact that in `s` there is one character between `‘a’` and `‘c’`: so, the characters that `s` contains for sure are `‘b’` and one between `‘a’` and `‘c’`. We can consider this as an improvement to the first abstract domain we are going to define. In fact, this domain would create some problems, mainly how to trace information about alternative possibilities without the explosion of considered cases. For the moment, therefore, we won’t explore this idea further and we will stick to the simplest one, since for the examples we are interested in it is enough.

6.1 Certainly contained characters

A string could be represented as a set, the set of certainly contained characters. Our abstract domain will be:

$$\boxed{\mathcal{A}_1 = \mathcal{P}(K)}$$

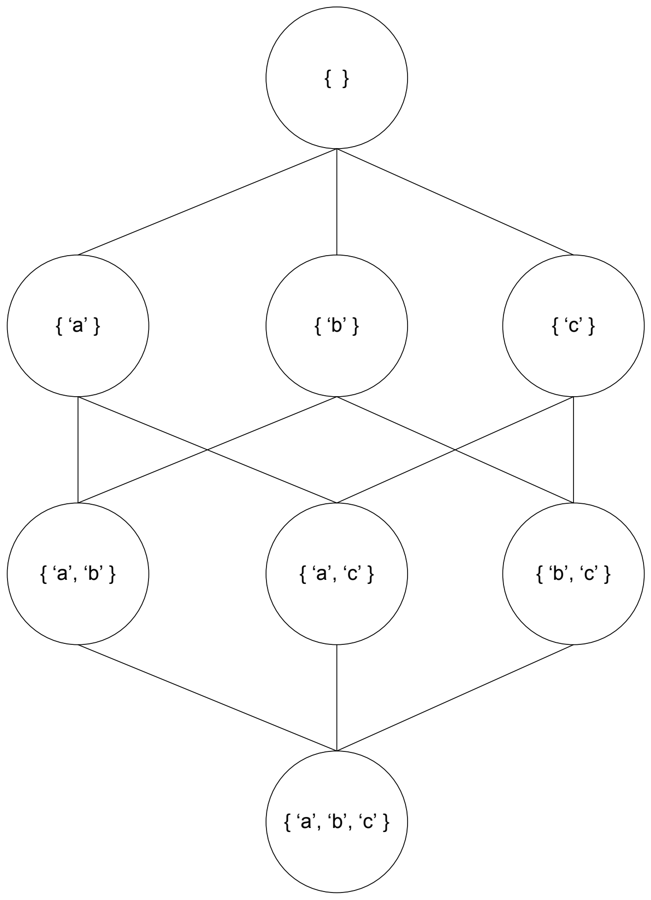
(remember that K is our alphabet, so $\mathcal{P}(K)$ is the set containing all the subsets of characters in K).

The **partial order** on this domain is:

$$\boxed{S \leq_{\mathcal{A}_1} T \Leftrightarrow T \subseteq S}$$

We can see that this definition is the opposite of the usual one (which would be: $S \leq T \Leftrightarrow S \subseteq T$). This is because, the more information we have on the string (that is, the more characters are in the set), the less number of strings we are representing. For example, $\{‘a’\}$ represents all the strings which contain at least one occurrence of $‘a’$. Instead, $\{‘a’, ‘b’\}$ represents all the strings which contain at least one occurrence of $‘a’$ and, at the same time, one of $‘b’$. The first set $\{‘a’\}$ contains more strings than the second one, because every string that contains an $‘a’$ and a $‘b’$ contains an $‘a’$, but not vice versa. So, $\{‘a’\}$ should be bigger (in our partial order) than $\{‘a’, ‘b’\}$. In fact: $\{‘a’\} \subseteq \{‘a’, ‘b’\} \Rightarrow \{‘a’, ‘b’\} \leq \{‘a’\}$.

Let $K = \{‘a’, ‘b’, ‘c’\}$, then the **lattice** of $\mathcal{A}_1$ is the following:



The **least upper bound** operator is set intersection, while the **greater lower bound** is set union. For example:

- $\sqcup\,(\{‘a’,‘b’\},\{‘a’,‘c’\}) = \{‘a’,‘b’\} \cap \{‘a’,‘c’\} = \{\,‘a’\,\}$
- $\sqcap\,(\{‘a’,‘b’\},\{‘a’,‘c’\}) = \{‘a’,‘b’\} \cup \{‘a’,‘c’\} = \{‘a’,‘b’,‘c’\}$

We can see from the lattice above that **top** and **bottom** are, respectively:

- $\emptyset$, since $\forall A : \emptyset \subseteq A$, so $A \leq_{\mathcal{A}_1} \emptyset\ \forall A$ by definition of $\leq_{\mathcal{A}_1}$
- K, since $\forall A \in \mathcal{A}_1 : A \subseteq K$, so $K \leq_{\mathcal{A}_1} A\ \forall A \in \mathcal{A}_1$ by definition of $\leq_{\mathcal{A}_1}$

In order to define the abstraction function, remember that our concrete domain is made of set of strings. We will build such function in two steps: first we will see how to go from a single concrete string to its abstract equivalent. Then we will consider more than one string and complete the definition of the abstraction function.

A string can have multiple abstractions: for example, the string "abc" can be represented (in the abstract domain $\mathcal{A}_1$) as $\{‘a’\}$ or as $\{‘a’, ‘b’\}$ or even as $\emptyset$, etc. Since collecting more information is better than less information, we choose to associate to a string the abstract value which contains the most information about it (that is, its best abstraction). In the case of "abc", the most informative representation is: $\{‘a’, ‘b’, ‘c’\}$. In this representation we include in the set of certainly contained characters all the characters which compose the string. Note that this is the smallest value (that is, the more precise) of all those associated to "abc". In fact, with respect to our order $\leq_{\mathcal{A}_1}$, we have:

$$\{‘a’, ‘b’, ‘c’\} \leq_{\mathcal{A}_1} \{‘a’, ‘b’\}$$

but also

$$\{‘a’, ‘b’, ‘c’\} \leq_{\mathcal{A}_1} \{‘a’\}$$

$$\{‘a’, ‘b’, ‘c’\} \leq_{\mathcal{A}_1} \emptyset$$

The function which abstracts a single string (α') is thus the following:

$$\alpha'_1(s) = \{\, c : c \in s\}$$

The **abstraction function** has to take us from a set of strings S (that is, an element of $\mathcal{D}$) to an element of $\mathcal{A}_1$. The solution to this problem is simple: we can abstract all the strings in the set and then compute the least upper bound of them in $\mathcal{A}_1$:

$$\boxed{\alpha_1(S) = \bigsqcup_{s \in S} \alpha'_1(s) = \bigcap_{s \in S} \alpha'_1(s)}$$

The **concretization function** is:

$$\boxed{\gamma(A) = \{\, s : s \text{ is a string} \wedge \forall \text{char } c \in A, s \text{ contains } c \,\}}$$

This means that the abstract value A maps to the set of the strings which contain all the characters present in A. Note that this set contains an infinite number of elements, for example:

$$\gamma(\{‘a’\}) \\ = \{\, "a", "ab", "bbbaaaa", "abcdef", "a12sawqa", "acds", "slakdlcacc", \dots \}$$

The widening operator is defined as follows:

$$\nabla : (\mathcal{A}_1, \mathcal{A}_1) \to \mathcal{A}_1$$

$$\nabla\big(\alpha(\mathrm{s1}), \alpha(\mathrm{s2})\big) = \alpha(s1) \cap \alpha(s2)$$

because in domains with finite height the least upper bound converges in finite time, so it is also a widening operator. Our domain has finite height, and precisely its height is $|K| + 1$. Supposing that $K = \{\, ‘a’, ‘b’, ‘c’ \,\}$, consider a chain which goes from bottom $\perp$ to top $\top$ of this domain (for example: $(\{‘a’, ’b’, ’c’\}, \{‘a’, ’b’\}, \{‘a’\}, \emptyset)$). The length of this chain will be the height of the domain. In $\perp$ we have all the characters in K (the alphabet), so $\perp$ is a set with $k = |K|$ elements. At each step of the chain we have one character less than in the previous set, until we reach top ($\top = \emptyset$). Going from k elements to 0 elements takes $k + 1$ steps. Our domain height is thus $k + 1 = |K| + 1$. Since we consider only finite alphabets, we can say that our domain height is finite.

As for the **semantics** of this abstract domain, let’s see some valid “string/abstract representation of such string” associations:

- $"a1, ir\ \ l3/" \in \gamma(\{‘a’, ‘1’\})$
- $"a1, ir\ \ l3/" \in \gamma(\emptyset)$
- $"a1, ir\ \ l3/" \in \gamma(\{‘a’, ‘1’, ‘3’\})$

- $"a1, ir\ \ l3/" \in \gamma(\{‘a’, ‘1’, ‘3’, ‘,’, ‘i’, ‘\ ’, ‘l’, ‘/’, ‘r’\})$
- $"a1, ir\ \ l3/" \in \gamma(\{‘a’, ‘3’\})$

For example, $"a1, ir\ \ l3/" \notin \gamma(\{‘a’, ‘1’, ‘5’\})$ (it is semantically wrong), since $"a1, ir\ \ l3/"$ does not contain the character ‘5’.

As final step of this abstract domain definition, we are going to see the semantics of the string operators we want to support (see “String Operators” paragraph). The notation will be the following one:

$$\boxed{S[\![< op >, input\ abstract\ values]\!] = output\ abstract\ value}$$

6.1.1 String(char[] data)

This is the string constructor. A string is made of an array of character. The semantics of this construct is easy: all the characters present in `data` will be certainly contained in the string. Consider for example the following code:

```
char[] data = {'a', 'b', 'c'};
String s = new String(data);
```

The semantics is:

$$S[\![new\ String, \text{arrayChars}]\!] = \{c : c \in \text{arrayChars}\}$$

In our example above, the resulting abstract value is { ‘a’, ‘b’, ‘c’ }.

Obviously if we don’t know what `data` contains, for example in the following case:

```
String foo(char[] data)
{
      String s = new String(data);
      return s;
}
```

or if the characters are read from user input, then we don't know anything about which characters are in the string. The semantics will thus be:

$$S[\![\text{new String, arrayChars}]\!] = \emptyset$$

6.1.2 concat(String str)

This operator takes two strings (one is the object it is invocated on) and concatenates them. For example, if `s1 = "hello "` and `s2 = "world"` then `s1.concat(s2) = "hello world"`. Suppose that $\overline{s1}$ is the abstract representation of `s1` and that $\overline{s2}$ is the abstract representation of `s2`. The semantics of this operator is:

$$S\left[\!\left[\text{concat}, \overline{s1}, \overline{s2}\right]\!\right] = \overline{s1} \cup \overline{s2}$$

In fact, when we concatenate two strings we create a new string which contains all the characters of the two initial strings. We exploit all the information we have, since all the certainly contained characters of `s1` and `s2` will be certainly contained in `s3` too.

6.1.3 indexOf(int c)

This operator returns the index within the string of the first occurrence of the specified character. Unfortunately, even if we have the certainty that the character is in the string (that is, the character is in the set of the abstract representation) we don't know in which position it is. Remember that in these first domains we are concentrating on character inclusion but we are not concerned about order.

Supposing that we are working in collaboration with a numerical abstract domain, the semantics of this operator in our domain will be $\top$ (the top element of the numerical domain) if `c` does not belong to the abstract representation of the string. Otherwise, we know that the result will be a number greater than -1 since the character is contained in the string:

$$S[[\text{indexof}, \overline{s}, \text{c}]] = \begin{cases} \top & \text{if c} \notin \overline{s} \\ > -1 & \text{if c} \in \overline{s} \end{cases}$$

6.1.4 lastIndexOf(int c)

The same considerations of the previous paragraph apply here. Since we don't know anything about order of character, we are forced to return ⊤ (if $c \notin \overline{s}$) or > -1 (if $c \in \overline{s}$).

6.1.5 substring(int beginIndex, int endIndex)

This operator returns a new string (`s2`) that is a substring of this string (`s1`). The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Unfortunately, considerations about order apply here too. Suppose that the string on which this method is called is `s1` and that its abstract representation is $\overline{s1}$. Since we don't know if the character in $\overline{s1}$ are in the range `[beginIndex, endIndex-1]` or if they are outside, we cannot propagate this information. We are forced to lose all the information we accumulated:

$$S\left[\left[\text{substring}, \overline{s1}, \text{beginIndex,endIndex}\right]\right] = \top = \emptyset$$

6.1.6 contains(CharSequence seq)

This operator returns true if and only if the string (`s1`) contains the specified sequence of char values (`seq`). If the sequence of characters `seq` contains more than one character, though, we will have to return ⊤ (the top element of a boolean domain). In fact, even if we knew that all the characters in `seq` are contained in `s1` (this happens if they all belong to $\overline{s1}$) we cannot guarantee that they are in the order specified by `seq`. Suppose for example that `seq = "eb"` and $\overline{s1} = \{'b', 'e'\}$. `s1` could be `"beldnfedv"` or `"be"` or `"dasebc"` or `"bbbbbbbbe"`, etc. It can be seen that not all of these possibilities contain `"eb"`. Things are different if `seq` contains only one character. In this case we can check if such character is present in the abstract representation of the string. In conclusion, the results of this method will be the following:

$$S\left[\left[\text{contains}, \overline{s1}, \text{seq}\right]\right] = \begin{cases} true & \text{if } length(\text{seq}) = 1 \wedge \text{seq}[0] \in \overline{s1} \\ \top & \text{otherwise} \end{cases}$$

We return `true` if the character sequence `seq` contains only one character and this character belongs to the set of certainly contained characters of `s1`. Otherwise we return ⊤.

The usefulness of this domain is for programs which need to know if a single character is present in a given string. For example, programs like:

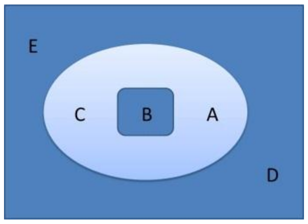
```
...
if(s.contains('a'))
       //do something
...
```

6.2 Maybe contained characters

We saw an abstract domain based on character *inclusion* in a string. Now we are going to see the opposite: an abstract domain based on character *exclusion* from a string. In fact, a string could also be represented as the *set of certainly **not** contained characters*. However, since humans are used to reason with positive knowledge rather than with negative knowledge, we are going to consider the complement of such set: the set of maybe contained characters. Let us consider a small example to understand this relationship:

```
...
if(...)
       s = "ab";
else
       s = "bc";
...
```

After this snippet of code, we know for sure that `s` contains a "b". We also know that it contains one character between "a" and "c", and it does not contain any other character in the alphabet. Supposing that our alphabet is $K = \{ a, b, c, d, e\}$ (for representation clarity), we can picture the string s like this:



The exterior quad represents the alphabet, K. In fact, it contains all the characters of K: a, b, c, d, e. The oval represents the characters that s may contain: a, b, c. The interior quad represents the characters that s contains for sure: b. Finally, the complement of the oval with respect to the exterior quad represents the characters that s certainly not contains: d, e. Thus, if we call NC the set of certainly not contained characters and MC the set of may be contained characters, we can see that the following equations hold:

$$MC = K \setminus NC$$

$$NC = K \setminus MC$$

Since:

- for humans it is more immediate to imagine “strings which may contains some characters” than “strings which certainly do not contain some characters”
- given K, MC and NC represent the same kind of information (they can be mutually determined, as you can see from the two equations above), at the condition that K is finite

we choose to use MC instead of NC.

Our abstract domain will be made of set of characters:

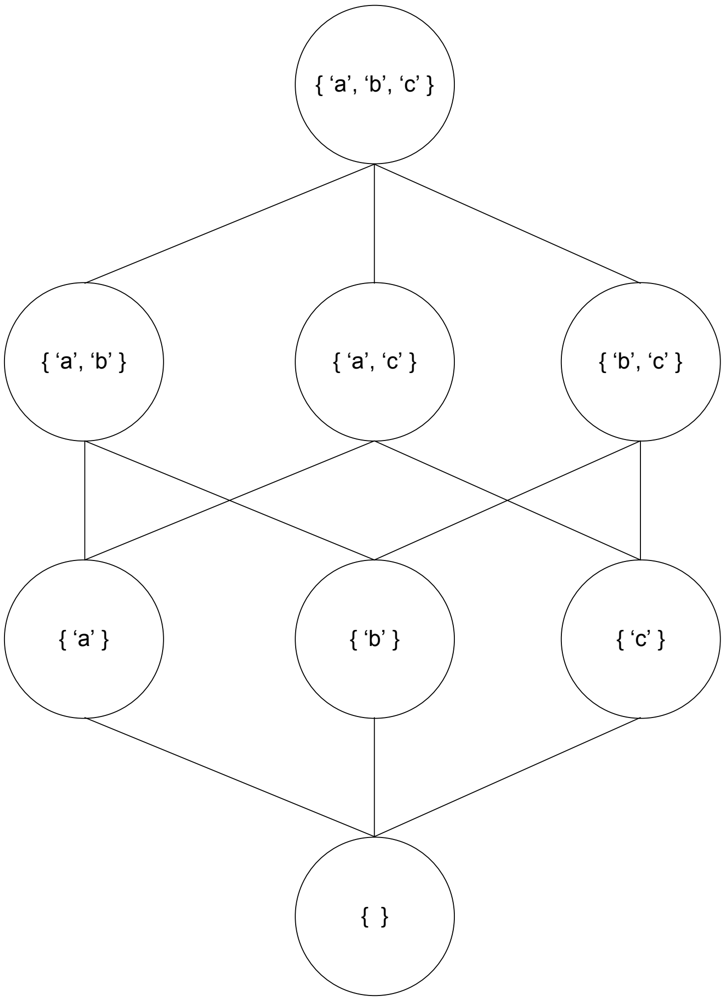
$$\boxed{\mathcal{A}_2 = \mathcal{P}(K)}$$

The **partial order** on this domain is, differently than in the previous domain ($\mathcal{A}_1$):

$$\boxed{S \leq_{\mathcal{A}_2} T \Leftrightarrow S \subseteq T}$$

Let us see an example of this definition. Consider $K = \{‘a’, ‘b’, ‘c’\}$, $S = \{ ‘a’ \}$ and $T = \{ ‘a’, ‘b’ \}$. Then, S represents all the strings which certainly do not contain ‘b’s nor ‘c’s but may contain ‘a’s: $\{ \epsilon, a, aa, aaa, aaaa, \dots \}$. T, instead, represents all the strings which certainly do not contain any ‘c’s, but may contain ‘a’s or ‘b’s: $\{ \epsilon, a, b, ab, aa, ba, bb, aaa, aab, \dots \}$. It is immediate to see that T represents more strings than S: every string which may contain ‘a’ or ‘b’, may obviously contain ‘a’, but not viceversa. So, T should be bigger (in our partial order) than S. In fact: $\{ ‘a’ \} \subseteq \{ ‘a’, ‘b’ \} \Rightarrow \{ ‘a’ \} \leq_{\mathcal{A}_2} \{ ‘a’, ‘b’ \} \Rightarrow S \leq_{\mathcal{A}_2} T$.

Let $K = \{‘a’, ‘b’, ‘c’\}$, then the **lattice** of $\mathcal{A}_2$ is the following:



Notice that the lattice is similar to the one of $\mathcal{A}_1$, but it is reversed. This happens because the partial order has the opposite definition. **Least upper bound**, **greatest lower bound**, **top** and **bottom** are the opposite too. The least upper bound operator is set union, whilst the greatest lower bound is set intersection. For example:

- $\sqcup\,(\{‘a’,‘b’\},\{‘a’,‘c’\}) = \{‘a’,‘b’\} \cup \{‘a’,‘c’\} = \{‘a’,‘b’,‘c’\}$
- $\sqcap\,(\{‘a’,‘b’\},\{‘a’,‘c’\}) = \{‘a’,‘b’\} \cap \{‘a’,‘c’\} = \{‘a’\}$

Top and bottom are, respectively:

- K, since $A \subseteq K\ \forall A \in \mathcal{A}_2$, so $A \leq_{\mathcal{A}_2} K\ \forall A \in \mathcal{A}_2$
- $\emptyset$, since $\emptyset \subseteq A\ \forall A \in \mathcal{A}_2$, so $\emptyset \leq_{\mathcal{A}_2} A\ \forall A \in \mathcal{A}_2$

To define the abstraction function, we will follow the same procedure we used for $\mathcal{A}_1$. In fact, a string can have multiple abstractions in this abstract domain too. For example, the string "abc" can be represented (in $\mathcal{A}_2$) as $\{\text{‘}a\text{’}, \text{‘}b\text{’}, \text{‘}c\text{’}\}$ or as $\{\text{‘}a\text{’}, \text{‘}b\text{’}, \text{‘}c\text{’}, \text{‘}d\text{’}, \text{‘}e\text{’}\}$ or even as K, etc. In the case of "abc", the representation which is most informative is: $\{\text{‘}a\text{’}, \text{‘}b\text{’}, \text{‘}c\text{’}\}$. In this representation we include in the set of maybe contained characters only characters which effectively compose the string. Note that this is the smallest value (that is, the more precise) of all those associated to "abc". In fact, with respect to our order $\leq_{\mathcal{A}_2}$, we have (supposing that $d, e, f \in K$):

$$\{\text{‘}a\text{’}, \text{‘}b\text{’}, \text{‘}c\text{’}\} \leq_{\mathcal{A}_2} K = \{\text{‘}a\text{’}, \text{‘}b\text{’}, \text{‘}c\text{’}, \text{‘}d\text{’}, \text{‘}e\text{’}, \text{‘}f\text{’}\}$$

but also

$$\{\text{‘}a\text{’}, \text{‘}b\text{’}, \text{‘}c\text{’}\} \leq_{\mathcal{A}_2} \{\text{‘}a\text{’}, \text{‘}b\text{’}, \text{‘}c\text{’}, \text{‘}d\text{’}, \text{‘}e\text{’}\}$$

$$\{\text{‘}a\text{’}, \text{‘}b\text{’}, \text{‘}c\text{’}\} \leq_{\mathcal{A}_2} \{\text{‘}a\text{’}, \text{‘}b\text{’}, \text{‘}c\text{’}, \text{‘}d\text{’}\}$$

The function which abstracts a single string is thus the following:

$$\alpha_2'(s) = \{\, c : c \in s \,\}$$

The **abstraction function** follows the same principle we saw for the abstraction function of $\mathcal{A}_1$: abstract all the strings in the concrete set and then compute the least upper bound of them in the domain.

$$\boxed{\alpha_2(S) = \coprod_{s \in S} \alpha_2'(s) = \bigcup_{s \in S} \alpha_2'(s)}$$

The **concretization function** will thus be:

$$\boxed{\gamma(A) = \{\, s : s \text{ is a string} \wedge \forall c \in s : c \in A \,\}}$$

This means that the abstract value A maps to the set of the strings which contain only characters taken from A. Note that this set generally contains an *infinite* number of elements, for example:

$$\gamma(\{‘a’\}) = \{\, \epsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa, \dots \}$$

The only abstract value which concretization is a *finite* set, is bottom ($\emptyset$). In fact, the only string which contains only characters present in $\emptyset$ is the empty string (ϵ), so:

$$\gamma(\emptyset) = \{\, \epsilon \,\}$$

The widening operator is the following:

$$\nabla : (\mathcal{A}_2, \mathcal{A}_2) \rightarrow \mathcal{A}_2$$

$$\nabla\big(\alpha(\mathrm{s1}), \alpha(\mathrm{s2})\big) = \alpha(s1) \cup \alpha(s2)$$

because (as we said for $\mathcal{A}_1$) in domains with finite height the least upper bound operator converges in finite time, so it is also a widening operator. The lattice of this domain corresponds (only reversed, but this does not change the height) to the one of $\mathcal{A}_1$, so we don’t need to repeat the reasoning about the lattice’s height.

Some valid “string/abstract representation of such string” associations are the following ones:

- $"a1, ir\ \ l1" \in \gamma(\{‘a’, ‘1’, ‘,’, ‘i’, ‘r’, ‘\ ’, ‘l’\})$
- $"a1, ir\ \ l1" \in \gamma(\{‘a’, ‘1’, ‘,’, ‘i’, ‘r’, ‘\ ’, ‘l’, ‘b’, ‘c’, ‘d’, ‘e’\})$
- $"a1, ir\ \ l1" \in \gamma(K)$
- $"a1, ir\ \ l1" \in \gamma(\{‘a’, ‘1’, ‘,’, ‘i’, ‘r’, ‘\ ’, ‘l’, ‘k’\})$
- $"a1, ir\ \ l1" \in \gamma(\{‘a’, ‘1’, ‘,’, ‘i’, ‘r’, ‘\ ’, ‘l’, ‘v’, ‘z’\})$

For example, $"a1, ir\ \ l1" \notin \gamma(\{‘a’, ‘3’, ‘l’\})$ (it is semantically wrong), since the characters $‘1’, ‘,’, ‘i’, ‘r’$ do not belong to $\{‘a’, ‘3’, ‘l’, \}$.

As final step of this second abstract domain definition, we are going to see the semantics of the string operators we want to support (see “String Operators” paragraph).

6.2.1 String(char[] data)

As it was for the first domain, the semantics of this construct is easy: all the characters present in `data` *maybe* contained in the string (we also know that they are *certainly* contained). Consider for example the following code:

```
char[] data = {'a', 'b', 'c'};
String s = new String(data);
```

The semantics of this operator will be:

$$S[\![\text{new String, arrayChars}]\!] = \{c : c \in \text{arrayChars}\}$$

Considering the snippet of code above, the resulting abstract value is $\{ ‘a’, ‘b’, ‘c’ \}$.

Obviously if we don’t know what `data` contains, for example in the following case:

```
String foo(char[] data)
{
      String s = new String(data);
      return s;
}
```

or if the characters are read from user input, then we don’t know anything about which characters are or are not in the string. The semantics will thus be:

$$S[\![\text{new String, arrayChars}]\!] = K$$

since the string may contain any character from the alphabet K. Note that we can build this abstract representation only in the hypothesis that K is a finite set. Even with this assumption, this operation is computationally expensive.

6.2.2 concat(String s2)

This operator takes two strings (one is the object it is invocated on, `s1`) and concatenates them. Suppose that $\overline{s1}$ is the abstract representation of `s1` and that $\overline{s2}$ is the abstract representation of `s2`. The semantics of this operator will be:

$$S\left[\left[\text{concat}, \overline{s1}, \overline{s2}\right]\right] = \overline{s1} \cup \overline{s2}$$

In fact, when we concatenate two strings we create a new string which contains all the characters of the two initial strings. If we know that a particular character could have been present in any of the two initial strings, then we know that it could be contained in the result string too. For example, suppose that $\overline{s1} = \{\, 'a', 'b' \,\}$ and $\overline{s2} = \{\, 'b', 'c' \,\}$. Then, according to our definition $\overline{s3} = \{'a', 'b', 'c' \,\}$.

6.2.3 indexOf(int c)

This operator returns the index within the string of the first occurrence of the specified character. Even if we lack information about character order, we can return `-1` if the character `c` is not present in the abstract representation of the string.

Supposing that we are working in collaboration with a numerical abstract domain, the semantics of this operator in our domain will thus be:

$$S[[\text{indexOf}, \overline{s}, \text{c}]] = \begin{cases} -1 & if\ \text{c} \notin \overline{s} \\ \top & \text{otherwise} \end{cases}$$

where ⊤ is the top element of the numerical domain.

6.2.4 lastIndexOf(int c)

The same considerations of the previous paragraph apply here. We will return -1 if the character is certainly not contained in the string, $\top$ otherwise.

6.2.5 substring(int beginIndex, int endIndex)

This operator returns a new string (s2) that is a substring of this string (s1). Suppose that the string on which this method is called is s1 and that its abstract representation is $\overline{s1}$. The characters which may be contained in s1 may be contained in any substring of s1 too. So we can propagate all the information we have gained:

$$S\left[\left[\text{substring}, \overline{s1}, \text{beginIndex}, \text{endIndex}\right]\right] = \overline{s1}$$

6.2.6 contains(CharSequence seq)

This operator returns true if and only if the string (s1) contains the specified sequence of char values (seq). In this case, it doesn't matter if seq contains more than one character or not. In fact, it is sufficient that at least one character of seq is not in the abstract representation of the string and we can return false. Otherwise, if all the characters in seq are in $\overline{s1}$ we will have to return $\top$ (the top element of a boolean domain):

$$S\left[\left[\text{contains}, \overline{s1}, \text{seq}\right]\right] = \begin{cases} false & \text{if } \exists c \in \text{seq} : c \notin \overline{s1} \\ \top & \text{otherwise} \end{cases}$$

The usefulness of this domain is for programs which need to know if a character is not present in a given string, like example 1.4 (SQL Code).

6.3 Certainly contained and maybe contained characters

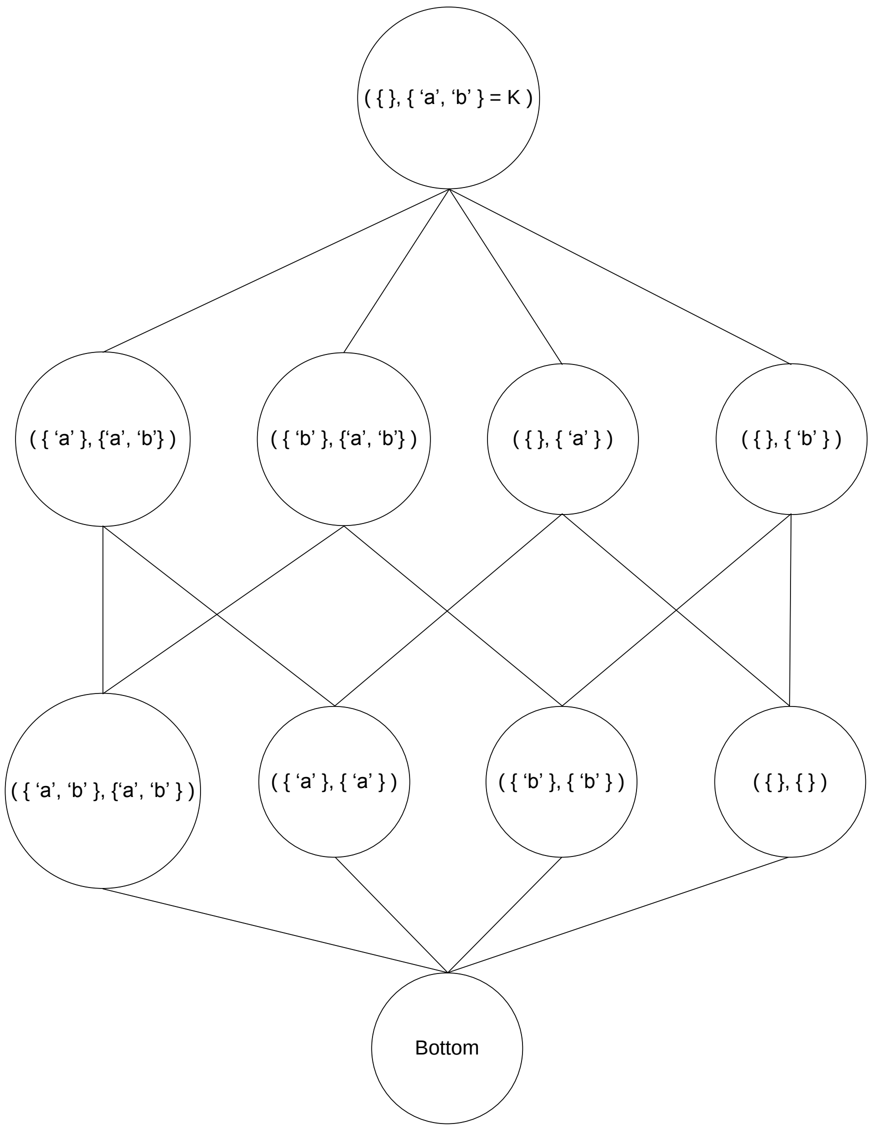
We can combine the abstract domains we created ($\mathcal{A}_1$ and $\mathcal{A}_2$) into one. We can thus represent a string as the combination of characters it certainly contains and the characters it may contain (i.e., the complement of the characters it certainly not contains). Our abstract domain is the Cartesian product of $\mathcal{A}_1$ and $\mathcal{A}_2$:

$$\boxed{\mathcal{A}_3 = \mathcal{A}_1 \times \mathcal{A}_2 = \mathcal{P}(K) \times \mathcal{P}(K)}$$

The **partial order** of $\mathcal{A}_3$ ($\leq_{\mathcal{A}_3}$) is the Cartesian product of the two orders $\leq_{\mathcal{A}_1}$ and $\leq_{\mathcal{A}_2}$:

$$\begin{gathered}(C_1, MC_1) \leq_{\mathcal{A}_3} (C_2, MC_2) \Leftrightarrow (C_1, MC_1) = \bot \vee \left(C_1 \leq_{\mathcal{A}_1} C_2 \wedge MC_1 \leq_{\mathcal{A}_2} MC_2\right) \\ \Leftrightarrow (C_1, MC_1) = \bot \vee (C_2 \subseteq C_1 \wedge MC_1 \subseteq MC_2)\end{gathered}$$

The **lattice** of this domain is (with $K = \{\text{'}a\text{'}, \text{'}b\text{'}\}$, for representation clarity):



The **least upper bound** is based on both set intersection and set union: $\sqcup\big((A, B), (C, D)\big) = (A \cap C, B \cup D)$.

The **greatest lower bound** is based both on set union and set intersection: $\sqcap\big((A,B),(C,D)\big) = (A \cup C, B \cap D)$.

The **top** element of the lattice is:

$$\top = (\emptyset, K)$$

that is, an element which could represent any string (since we know nothing about it, neither about inclusion or exclusion). We can easily prove that $\top$ respects the definition of top:

$$\emptyset \subseteq C \ \forall C \text{ (definition of } \emptyset\text{)}$$

$$\emptyset \subseteq C \ \forall C \in \mathcal{A}_1 \text{ (weakening of the above condition)}$$

$$MC \subseteq K \ \forall MC \in \mathcal{P}(K) \text{ (definition of } \mathcal{P}(K)\text{)}$$

$$MC \subseteq K \ \forall MC \in \mathcal{A}_2 \text{ (equivalent to the above condition, since } \mathcal{A}_2 = \mathcal{P}(K)\text{)}$$

$$\emptyset \subseteq C \ \forall C \in \mathcal{A}_1 \wedge MC \subseteq K \ \forall MC \in \mathcal{A}_2$$

$$C \leq_{\mathcal{A}_1} \emptyset \ \forall C \in \mathcal{A}_1 \wedge MC \leq_{\mathcal{A}_2} K \ \forall MC \in \mathcal{A}_2 \text{(definition of } \leq_{\mathcal{A}_1} \text{ and } \leq_{\mathcal{A}_2}\text{)}$$

$$(C, MC) \leq_{\mathcal{A}_3} (\emptyset, K) \ \forall (C, MC) \in \mathcal{A}_3 \text{ (definition of } \leq_{\mathcal{A}_3}\text{)}$$

The **bottom** element of the lattice is defined as:

$$\bot = \{\, (C, NC) : C \nsubseteq MC \,\}$$

For example, $(\{\,'a','b'\,\}, \{'b'\,\})$ is bottom, but also $(\{\,'a'\,\}, \emptyset)$ is bottom, and so on. In fact, every character that is certainly contained in the string must belong to the set of maybe contained characters. $\bot$ does not represent any concrete string. Its concretization will thus be an empty set. The partial order $\leq_{\mathcal{A}_3}$ is specifically built to guarantee that $\bot \leq_{\mathcal{A}_3} A \ \forall A \in \mathcal{A}_3$, in order to respect the definition of bottom element.

Now we want to define the abstraction function and concretization function of this abstract domain. Let's start with the abstraction function.

As we said for $\mathcal{A}_1$ and $\mathcal{A}_2$, a string can have multiple abstractions. For example, the string "abc" can be represented (in $\mathcal{A}_3$) as $(\{‘a’\}, K)$ or as $(\{‘a’,‘b’\},\{‘a’,‘b’,‘c’\})$ or even as $(\emptyset,\{‘a’,‘b’,‘c’,‘d’,‘e’\})$, etc. In the case of "$abc$", supposing that our characters set is called K, the representation which is most informative is: $(\{‘a’,‘b’,‘c’\},\{‘a’,‘b’,‘c’\})$. In this representation we include in both the sets C (certainly contained characters) and MC (maybe contained characters) all the characters which compose the string. As always, this is the smallest value (that is, the more precise) of all those associated to "abc". In fact, with respect to our order $\leq_{\mathcal{A}_3}$, we have (supposing that $d, e, f \in K$):

$$(\{‘a’,‘b’,‘c’\},\{‘a’,‘b’,‘c’\}) \leq_{\mathcal{A}_3} (\{‘a’\}, K)$$

but also

$$(\{‘a’,‘b’,‘c’\},\{‘a’,‘b’,‘c’\}) \leq_{\mathcal{A}_3} (\{‘a’,‘b’\},\{‘a’,‘b’,‘c’\})$$

$$(\{‘a’,‘b’,‘c’\},\{‘a’,‘b’,‘c’\}) \leq_{\mathcal{A}_3} (\emptyset,\{‘a’,‘b’,‘c’,‘d’,‘e’\})$$

The function which abstracts a single string is thus the following (note that it is the Cartesian product of the two function defined in $\mathcal{A}_1$ and $\mathcal{A}_2$):

$$\begin{aligned} \alpha_3'(s) &= (C_s, MC_s) \\ &= (\{c : c \in K \wedge c \in s\}, \{c : c \in K \wedge c \in s\}) \end{aligned}$$

The **abstraction function** takes us from a set of strings S ($S \in \mathcal{D}$) to $(C_S, MC_S) \in \mathcal{A}$:

$$\boxed{\alpha_3(S) = (C_S, MC_S) = \coprod_{s \in S} \alpha_3'(s) = \left(\bigcap_{s \in S} C_s, \bigcup_{s \in S} MC_s\right)}$$

At the opposite of the abstraction function we find the **concretization function**. Remember that such function maps an element $a \in \mathcal{A}$ to an element $d \in \mathcal{D}$. Since our concrete domain $\mathcal{D}$ is made of string sets, an abstract value maps to a set of possible strings, which is exactly what we want. In fact, since an abstract value is an approximation of the real value it represents, it is bound to lose some information. The lost information leads to the abstract value representing more than one real value. Our concretization function is the following:

$$\boxed{\gamma(a) = \gamma((C, MC)) = \{\, s : s \text{ is a string} \wedge \forall \text{char } c \in C, c \in s \wedge \forall \text{char } c \in s, c \in MC \,\}}$$

Note that the concretization function is the Cartesian product of the two concretization functions defined for $\mathcal{A}_1$ and $\mathcal{A}_2$. An example:

$$\gamma\big((\{‘a’, ‘b’\}, \{‘a’, ‘b’, ‘c’, ‘d’, ‘e’\})\big) = \{\, "ab", "abc", "abce", "aaaabbbb", "bababc", "aedcbae", etc \ldots \}$$

Note that this set is an infinite set. Even when we have the most detailed information (that is, the certainly contained and maybe contained characters are the same set), the resulting set is infinite (due to possible characters repetitions):

$$\gamma\big((\{‘a’, ‘b’, ‘c’\}, \{‘a’, ‘b’, ‘c’\})\big) = \{"abc", "aabc", "abbc", "abcc", "aaacacb", "bcabcac", "cccccccab", etc \ldots \}$$

The widening operator is the following:

$$\nabla : (\mathcal{A}_3, \mathcal{A}_3) \rightarrow \mathcal{A}_3$$

$$\nabla\big(\alpha(\mathrm{s1}), \alpha(\mathrm{s2})\big) = \Omega\big((C_1, NC_1), (C_2, NC_2)\big) = (C_1 \cap C_2, NC_1 \cup NC_2)$$

because in domains with finite height the least upper bound operator is also a widening operator (it converges in finite time). Our domain has finite height, and precisely its height is $|K| + 1$. In fact, consider a chain which goes from bottom $\perp$

to top ⊤ of this domain (for example: $\bot \rightarrow (\{‘a’,’b’\},\{‘a’,’b’\}) \rightarrow (\{‘a’\},\{‘a’,’b’\}) \rightarrow$ $\rightarrow (\emptyset,\{‘a’,’b’\})$). The length of this chain will be the height of the domain. Let us exclude the first step of the chain, which takes from $\bot$ to $(\{‘a’,’b’\},\{‘a’,’b’\})$, since it is constant (1 step) and it is the same for every chain of the domain. The other part of the chain starts with two equal sets (we can call them S: (S,S)) and ends with $(\emptyset,K)$. Let $n_1 = |S|$ and $n_2 = |K|$. At each step of the chain we have two possibility: the first set loses an element (until it becomes $\emptyset$) or the second set gains an element (until it becomes K). Thus, the first set entails n_1 steps; the second set entails $n_2 - n_1$ steps. The total number of steps is $n_1 + (n_2 - n_1) = n_2 = |K|$. Adding the constant step from bottom to (S,S), we conclude that our domain height is $|K| + 1$. Since we consider only finite alphabets, we can say that our domain height is finite.

As for the **semantics** of this domain ($\mathcal{A}_3$), let’s see some valid “string/abstract representation of such string” associations:

- $"a1,ir\ \ l1" \in \gamma((\{'a','1',' '\},\ \{'a','1',' ',',','i','r','l'\}))$
- $"a1,ir\ \ l1" \in \gamma((\emptyset,\ \{'a','1',' ',',','i','r','l','b','c'\}))$
- $"a1,ir\ \ l1" \in \gamma((\{'a','1',' '\},\ K))$
- $"a1,ir\ \ l1" \in \gamma((\{'a','1',',','i','r',' ','l'\},$ $\{'a','1',',','i','r',' ','l','z','k','t'\}))$
- $"a1,ir\ \ l1" \in \gamma\big((\emptyset,\ K)\big)$

Some invalid associations are instead the following:

- $"a1,ir\ \ l1" \notin \gamma((\{'a','1','5'\},\ \{'a','1',' ',',','i','r','l','b','5'\}))$, because the string does not contain the character ‘5’;
- $"a1,ir\ \ l1" \notin \gamma((\{'a','1',' '\},\ \{'a','1',' ',',','i','r'\}))$, because the string does contain the character ‘l’ which is not present in MC;
- $"a1,ir\ \ l1" \notin \gamma((\{'a','1','b'\},\ \{'a','1',' ',',','i','r','l'\}))$, because the string does not contain the character ‘b’; in this case, though, we know

there is something wrong even without looking at the original string, because the same character (‘b’) appears in the first set but not in the second: as we know, a character cannot be contained and not contained at the same time. This element would be classified as ⊥, since $C \nsubseteq MC$.

As we did for $\mathcal{A}_1$ and $\mathcal{A}_2$, we are now going to define the semantics of the supported string operators (see “String Operators” paragraph). Since $\mathcal{A}_3$ is the Cartesian product of $\mathcal{A}_1$ and $\mathcal{A}_2$ and the semantics of those two domains have already been identified, defining the semantics of this domain will be easy.

6.3.1 String(char[] data)

All the characters present in `data` will certainly be contained in the string. All the other characters will be certainly *not* contained in the string. Consider for example the following code:

```
char[] data = {'a', 'b', 'c'};
String s = new String(data);
```

The semantics of this operator will be:

$$S[\![\text{new String, arrayChars}]\!] \\ = (\{c : c \in K \wedge c \in \text{arrayChars}\}, \{c : c \in K \wedge c \in \text{arrayChars}\})$$

The abstract value resulting from our simple example above is $(\{‘a’,‘b’,‘c’\},\{‘a’,‘b’,‘c’\})$.

Obviously if we don’t know what `data` contains (for example when the characters are read from user input) our semantics will be:

$$S[\![\text{new String, arrayChars}]\!] = (\emptyset, K)$$

6.3.2 concat(String s2)

This operator takes two strings (one is the object it is invocated on, `s1`) and concatenates them. Suppose that $\overline{s1} = (C_1, MC_1)$ is the abstract representation of `s1` and that $\overline{s2} = (C_2, MC_2)$ is the abstract representation of `s2`. The semantics of this operator will be:

$$S\left[\left[\text{concat}, \overline{s1}, \overline{s2}\right]\right] = \overline{s3} = (C_3, NC_3) = (C_1 \cup C_2, NC_1 \cup NC_2)$$

In fact, when we concatenate two strings we create a new string which contains all the characters of the two initial strings. If we know that a character is present in at least one of the two input strings then it will certainly be contained in the resulting string too. Similarly, if we know that a particular character could be contained in any of the two strings, then it could be contained in the result string too.

6.3.3 indexOf(int c)

This operator returns the index within the string of the first occurrence of the specified character. If we know for sure that the character is in the string we can assume that the returned value will be different (greater) from -1. We can return `-1` if we are sure that the character `c` is not contained in the string. Otherwise, we are not able to return a valid integer, and we will be forced to return $\top$. Supposing that we are working in collaboration with a numerical abstract domain and that the abstract representation of `s1` is $\overline{s1} = (C_1, MC_1)$, the semantics of this operator in our domain will thus be:

$$S\left[\left[\text{indexOf}, \overline{s1}, \text{c}\right]\right] = \begin{cases} -1 & \text{if c} \notin MC_1 \\ > -1 & \text{if c} \in C_1 \\ \top & \text{otherwise} \end{cases}$$

where $\top$ is the top element of the numerical domain.

6.3.4 lastIndexOf(int c)

The same considerations of the previous paragraph apply here. We will return -1 if the character is certainly not contained in the string, something greater than -1 if the character is certainly contained in the string, $\top$ otherwise.

6.3.5 substring(int beginIndex, int endIndex)

This operator returns a new string (s2) that is a substring of this string (s1). Suppose that the string on which this method is called is s1 and that its abstract representation is $\overline{s1} = (C_1, MC_1)$. The characters which are may be contained in s1 could be contained in any substring of s1 too. We cannot say anything about contained characters.

$$S\left[\left[\text{substring}, \overline{s1}, \text{beginIndex,endIndex}\right]\right] = \overline{s2} = (C_2, MC_2) = (\emptyset, MC_1)$$

6.3.6 contains(CharSequence seq)

This operator returns true if and only if the string (s1) contains the specified sequence of char values (seq). We will have to discern two cases: when seq contains at least one character and when it contains exactly one character. In the first case, we can return `false` if we know that any of the characters in seq is certainly not contained in the string. In the second case, we can return `true` if the single character contained in seq is certainly contained in our string. Supposing that $\overline{s1} = (C_1, MC_1)$ we have:

$$S\left[\left[\text{contains}, \overline{s1}, \text{seq}\right]\right] = \begin{cases} true & \text{if seq.}Length = 1 \wedge \text{seq}[0] \in C_1 \\ false & \text{if } \exists c \in \text{seq}: \ c \notin MC_1 \\ \top & \text{otherwise} \end{cases}$$

where ⊤ is the top element of the boolean domain.

The usefulness of this domain is evident, since it combines the strengths of $\mathcal{A}_1$ and those of $\mathcal{A}_2$ without adding much computational cost.

7 Abstract domains based on prefix and suffix

In the previous chapter we focused on characters inclusion. We approximated a string through the characters we knew it contained and the ones we knew it may have contained. Each character contained in that representation was completely unrelated with regards to the others. We did not maintain any information about order between characters.

The two abstract domains we will define in this chapter will consider both inclusion and order among characters, but with a limitation. We will concentrate only on how the strings begin and how the strings end. In fact, one of these domains will be able to track information like "*this string begins with "abc"* "; the other, information like "*this string ends with "def"* ". Thus, a string will be identified through its prefix and suffix, respectively.

7.1 Prefix

As we mentioned before, in this abstract domain ($\mathcal{A}_4$) we will identify strings through their prefix. A string will thus be something which begins with a certain sequence of character. We do not know anything about how the string continues after such prefix. Let us see some examples, in which we will introduce the notation of such representation (supposing $K = \{a, b, c, d, e\}$):

- $abc* = \{ abc, abca, abcb, abcc, abcd, abce, abcaa, abcab, abcac, \dots \}$
- $ab* = \{ ab, aba, abb, abc, abd, abe, abaa, abab, abac, \dots \}$
- $d* = \{ d, da, db, dc, dd, de, daa, dab, dac, dad, dae, dba, \dots \}$
- $* = \{\epsilon, a, b, c, d, e, aa, ab, ac, ad, ae, ba, \dots \}$

We represent a string as a sequence of characters followed by an asterisk $*$. The asterisk $*$ represents any string (ϵ included). For example, $abc*$ represents all the strings which begin with "abc", including "abc" itself. Instead, $*$ represents all possible strings, since the fixed prefix is empty: there is no constraint on how the string should begin.

Since the asterisk $*$ at the end of the representation is always present, a particular string representation is uniquely identified only through its prefix. Thus, our domain is the following:

$$\mathcal{A}_4 = K^{\vec{+}} \cup \bot$$

(the notation $\cdot^{\vec{+}}$ has been introduced in Chapter 2). An element of our domain is a sequence of characters (taken from our alphabet K) representing a string prefix. The asterisk $*$, which should always follow the sequence of characters, has been omitted to lighten the notation. So, the element "abc" $\in \mathcal{A}_4$ corresponds to $abc\ *$.

The **partial order** on this domain is:

$$\boxed{S \leq_{\mathcal{A}_4} T \Leftrightarrow (len(T) \leq len(S) \wedge T[i] = S[i]\ \forall i \in [0, len(T) - 1])\ \vee S = \bot}$$

Let us explain this definition. An abstract string S is smaller (in our order) than another abstract string T if T is a prefix of S. For T to be a prefix of S, we need to validate two conditions:

1. $len(T) \leq len(S)$
   a. that is, T must contain less or equal characters than S;
2. $T[i] = S[i]\ \forall i \in [0, len(T) - 1]$
   a. that is, the characters of T must be equal to the corresponding first $len(T)$ characters of S;

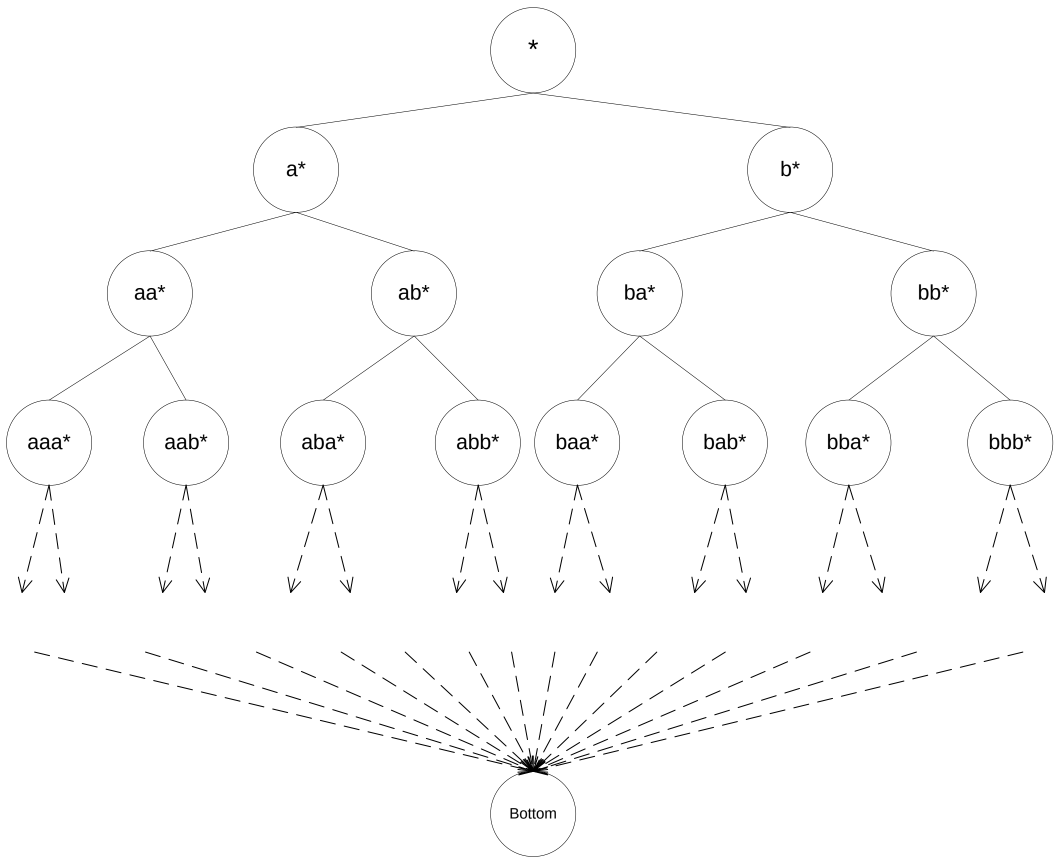
Otherwise, $S \leq_{\mathcal{A}_4} T$ if S is the bottom $\bot$ of the domain.

Some examples of this order:

- Consider $S = abc\ *$ and $T = ab\ *$; $len(S) = 3$ and $len(T) = 2$, so $len(T) < len(S)$. Moreover, $T[0] = a = S[0]$ and $T[1] = b = S[1]$. Then, $S \leq_{\mathcal{A}_4} T$. This is reasonable: the strings represented by S are a subset of those represented by T. In fact, the strings beginning with ab include those beginning with abc.

- Consider $S = d*$ and $T = *$; $len(S) = 1$ and $len(T) = 0$. We have $len(T) < len(S)$ but we do not have to compare any characters, since $len(T) = 0$. Then, $S \leq_{\mathcal{A}_4} T$. In fact, the strings beginning with d are a subset of all possible strings.
- Consider $S = abc*$ and $T = abd*$; $len(S) = lent(T) = 3$. We have that $T[0] = a = S[0], T[1] = b = S[1]$ but $T[2] = d \neq S[2] = c$. Thus, $S \nleq_{\mathcal{A}_4} T$ and $T \nleq_{\mathcal{A}_4} S$. In fact, each string beginning with abc is different from any string beginning with abd. The two abstract elements S and T have nothing in common.

Let $K = \{‘a’, ‘b’\}$ (for representation clarity), then the **lattice** of $\mathcal{A}_4$ is the following:



Note that this is an *infinite* domain. In fact, given any prefix, we can always add a character at the end of it, thus obtaining a new prefix, longer (and smaller according to our order $\leq_{\mathcal{A}_4}$) than the first one. However, the domain respects the ACC condition (see Chapter 2). In fact, given a certain prefix S, where $len(S) = n$, the ascending chain starting at S is the following:

$$S \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n$$

where $S_1 = trunc(S, n-1)$ (that is, S_1 corresponds to S without its last character; we defined $trunc$ operator in paragraph 2.2), $S_2 = trunc(S_1, n-2)$, $S_3 = trunc(S_2, n-3)$, and so on, until we reach $S_n = trunc(S_{n-1}, n-n) = trunc(S_{n-1}, 0) = \epsilon^l$. S_n corresponds to an empty prefix: it is $*$, which represents any string, the top of our domain. Thus, given any prefix S of length n (which is finite), the ascending chain starting at S has finite length $n+1$. On the opposite, our domain does not satisfy the DCC condition.

The **least upper bound** operator is rather easy. Given two prefixes, their least upper bound is their longest common prefix (defined in paragraph 2.3). If the two prefixes do not have anything in common, the least upper bound is $*$ (the prefix is empty). Examples:

- $\sqcup\,(ab\,*, abc\,*) = ab\,*$
- $\sqcup\,(abc\,*, ade\,*) = a\,*$
- $\sqcup\,(dbc\,*, abc\,*) = \,*$

The **greater lower bound** is defined as follows:

$$\sqcap\,(S_1, S_2) = \begin{cases} S_1 \text{ if } S_1 \leq_{\mathcal{A}_4} S_2 \\ S_2 \text{ if } S_2 \leq_{\mathcal{A}_4} S_1 \\ \bot \;\text{ otherwise} \end{cases}$$

In fact, if the two prefixes are not comparable with respect to our order (think about $a\,*$ and $b\,*$, for example), then the strings sets they represent have nothing in common. Their greatest lower bound is thus $\bot$. If they are comparable, it is immediate to see why the smaller element is the greatest lower bound. Examples:

- $\sqcap\,(ab\,*, abc\,*) = abc\,*$
- $\sqcap\,(abc\,*, ade\,*) = \bot$
- $\sqcap\,(abcde\,*, a\,*) = abcde\,*$

We can see from the lattice above that **top** and **bottom** are, respectively:

- $\top = *$, since $*$ is the empty prefix, which is prefix of any other prefix; $\top$ represents any possible string;
- $\bot$, by definition of $\mathcal{A}_4$;

In order to define the abstraction function, remember that our concrete domain is made of set of strings. We will build such function in two steps, as usual. First we will see how to go from a single concrete string to its abstract equivalent. Then we will consider more than one string and complete the definition of the abstraction function.

A string can have multiple abstractions: for example, the string "abc" can be represented (in our abstract domain $\mathcal{A}_4$) as $a\,*$ or as $ab\,*$ or as $abc\,*$ or even as $*$. Since more information is better than less information, we choose to associate to a string the abstract value which contains the most information about it (that is, its best abstraction). In the case of "abc", the representation which is most informative is: $abc\,*$. In this representation we use the longest prefix possible: the entire string. Note that this is the smallest value (that is, the more precise) of all those associated to "abc". In fact, with respect to our order $\leq_{\mathcal{A}_4}$, we have:

$$abc\,* \leq_{\mathcal{A}_4} a\,*$$

$$abc\,* \leq_{\mathcal{A}_4} ab\,*$$

$$abc\,* \leq_{\mathcal{A}_4} *$$

The function which abstracts a single string is thus the following:

$$\alpha'_4(s) = s\,*$$

The **abstraction function** has to take us from a set of strings S (that is, an element of $\mathcal{D}$) to an element of $\mathcal{A}_4$. We can abstract all the strings in the set and then compute the least upper bound of them in $\mathcal{A}_4$:

$$\boxed{\alpha_4(S) = \coprod_{s \in S} \alpha'_4(s)}$$

This means that we take the longest common prefix amongst all the strings in the set S. Some examples:

- $\alpha(\{a, abc, adef, aaa\}) = \sqcup\,(a*, abc*, adef*, aaa*) = a*$
- $\alpha(\{a, abc, adef, baa\}) = \sqcup\,(a*, abc*, adef*, baa*) = * = \top$

The **concretization function** is:

$$\boxed{\gamma(S) = \{\, s : s \text{ is a string} \wedge\ len(s) \geq len(S) \wedge \forall i \in [0, len(S)-1] :\ s[i] = S[i]\,\}}$$

This means that the abstract value S maps to the set of the strings which begin with the sequence of characters represented by S. Note that this set contains an infinite number of elements, for example (supposing $K = \{a, b\}$):

$$\gamma(a*) = \{\, a, aa, ab, aaa, aab, aba, abb, aaaa, aaab, aaba, aabb, \ldots\}$$

The **widening operator** is the following:

$$\nabla : (\mathcal{A}_4, \mathcal{A}_4) \rightarrow \mathcal{A}_4$$

$$\nabla\big(\alpha(\mathrm{s1}), \alpha(\mathrm{s2})\big) = \sqcup\,\big(\alpha(s1), \alpha(s2)\big)$$

because, in domains which satisfy ACC, the least upper bound converges in finite time, so it is also a widening operator. We proved that our domain $\mathcal{A}_4$, though having infinite height, satisfies ACC; so we can use the least upper bound operator as widening operator.

As for the **semantics** of this abstract domain, let us see some valid "string/abstract representation of such string" associations:

- $"abcdef" = abcdef*$
- $"abcdef" = abcde*$
- $"abcdef" = abcd*$
- $"abcdef" = abc*$
- $"abcdef" = ab*$
- $"abcdef" = a*$
- $"abcdef" = *$

For example, "$abcdef$" $= babcd*$ is semantically wrong, since "$abcdef$" does not begin with a 'b', as the prefix $babcd*$ imposes.

Now we are going to see the semantics of the string operators. The notation will be, as always, the following:

$$\boxed{S[\![< op >, input\ abstract\ values]\!] = output\ abstract\ value}$$

7.1.1 String(char[] data)

This is the string constructor. The semantics of this construct is easy: the prefix is made by the entire string, followed by the asterisk $*$. Consider for example the following code:

```
char[] data = {'a', 'b', 'c'};
String s = new String(data);
```

The semantics is:

$$S[\![new\ String, \text{arrayChars}]\!] = \text{arrayChars}*$$

In our example above, the resulting abstract value is abc $*$.

Obviously if we don't know what `data` contains (for example if the characters are read from user input), then we don't know anything about which characters are in the string. The prefix will be empty and the semantics will thus be:

$$S[\![\text{new String, arrayChars}]\!] = *$$

7.1.2 concat(String str)

This operator takes two strings and concatenates them. For example, if `s1 = "hello "` and `s2 = "world"` then `s1.concat(s2) = "hello world"`. Suppose that $\overline{s1}$ is the abstract representation of `s1` and that $\overline{s2}$ is the abstract representation of `s2`. The semantics of this operator is:

$$S\left[\left[\text{concat}, \overline{s1}, \overline{s2}\right]\right] = \overline{s1}$$

In fact, when we concatenate two strings we create a new string which begins with the first one. The prefix of the resulting string will then be the same one as the prefix of the first string. Sometimes, this operation makes us lose information. In the example cited above, we have:

$$\overline{s1} = "hello\ " *$$

$$\overline{s2} = "\ world" *$$

The abstract value resulting from concatenation of `s1` and `s2` is:

$$\overline{s_{res}} = "hello\ " *$$

when, actually, we know something more ($\overline{s_{res}}$ should be "$hello\ world$" $*$). This happens because, in this particular case, the asterisk of `s1` did not represent any character. The abstract approximation caused the loss of information.

7.1.3 indexOf(int c)

This operator returns the index within the string of the first occurrence of the specified character. If such character occurs in our prefix, then we know exactly its index and we can return it. Otherwise, we are forced to return ⊤ (supposing that we are working in collaboration with a numerical abstract domain), because we do not know if the character is masked by the asterisk $*$ or not.

The semantics of this operator in our domain will thus be:

$$S[[\text{indexof}, \overline{s}, \text{c}]] = \begin{cases} i & \text{if } \exists i : \overline{s}[i] = c \wedge \nexists j < i : \overline{s}[j] = c \\ \top & \text{otherwise} \end{cases}$$

7.1.4 lastIndexOf(int c)

This operator returns the index within the string of the last occurrence of the specified character.

If the character occurs in the prefix of our abstract value at index i, then we know that the index of its last occurrence will be $\geq i$, since it surely appears in the string at index i. We cannot know the precise index of the last occurrence, though, because we do not know if the character appears *again* in the "$*$ part" of the string.

Otherwise (if the character does not occur in the prefix of the abstract value), we are forced to return $\top$. In fact, we do not know if the character will appear in "$*$ part" of the string (and where) or not.

The semantics is thus the following:

$$S[\![\text{lastIndexOf}, \overline{s}, \text{c}]\!] = \begin{cases} \geq i & \text{if } \exists i : \overline{s}[i] = c \wedge \nexists j > i : \overline{s}[j] = c \\ \top \text{ otherwise} \end{cases}$$

7.1.5 substring(int beginIndex, int endIndex)

This operator returns a new string (`s2`) that is a substring of this string (`s1`). The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. We have to distinguish three different cases:

1. $endIndex \leq len(\overline{s1})$; in this case, the substring is completely included in the known prefix;
2. $endIndex > len(\overline{s1})$ but $beginIndex < len(\overline{s1})$; in this case, only the first part of the substring is in the prefix;
3. $beginIndex \geq len(\overline{s1})$; in this case, the substring is completely further the prefix; we are forced to return $\top$.

Given a sequence of characters $\overline{s1}$, we use the notation $\overline{s1}[i \cdots j]$ to indicate the subsequence starting at $\overline{s1}[i]$ and ending at $\overline{s1}[j]$. Then, the semantics of the `substring` operator will be:

$$S\left[\left[\text{substring}, \overline{s1}, \text{bI,eI}\right]\right] = \begin{cases} \overline{s1}[\text{bI} \cdots \text{eI}-1] \text{ if eI} \leq len(\overline{s1}) \\ \overline{s1}[\text{bI} \cdots len(\overline{s1})-1] \text{ if eI} > len(\overline{s1}) \wedge \text{bI} < len(\overline{s1}) \\ * \text{ if bI} \geq len(\overline{s1}) \end{cases}$$

Let us see some examples to clarify this definition. Consider the abstract value $\overline{s1} = abcdefg*$. Then, $len(\overline{s1}) = 7$.

- $S\left[\left[\text{substring}, \overline{s1}, 3{,}7\right]\right] = defg*$, since $7 = \text{eI} \leq len(\overline{s1}) = 7$;
- $S\left[\left[\text{substring}, \overline{s1}, 5{,}9\right]\right] = fg*$, since $9 = \text{eI} > len(\overline{s1}) = 7$ and $5 = \text{bI} < len(\overline{s1}) = 7$;
- $S\left[\left[\text{substring}, \overline{s1}, 8{,}12\right]\right] = *$, since $8 = \text{bI} \geq len(\overline{s1}) = 7$.

7.1.6 contains(CharSequence seq)

This operator returns true if and only if the string (`s1`) contains the specified sequence of char values (`seq`). If the sequence of characters `seq` contains more characters than the length of our prefix, we will certainly have to return ⊤ (the top element of a boolean domain). Otherwise, we can check the prefix to see if the sequence `seq` appears in it. In conclusion, the results of this method will be the following:

$$S\left[\left[\text{contains}, \overline{s1}, \text{seq}\right]\right] = \begin{cases} true & \text{if } len(\text{seq}) \leq len(\overline{s1}) \wedge \exists i : \left(i + len(\text{seq}) \leq len(\overline{s1})\right) \wedge \text{seq} = \overline{s1}[i \cdots i + len(\text{seq}) - 1] \\ \top & \text{otherwise} \end{cases}$$

We return `true` if the character sequence `seq` is present in the prefix, otherwise we return ⊤.

7.2 Suffix

In the abstract domain $\mathcal{A}_4$ we identified strings through their prefix. In this new domain, $\mathcal{A}_5$, we will identify strings through their suffix. A string will now be something which **ends** with a certain sequence of character. We do not know anything about how the string is made *before* such suffix.

The notation is very similar to the one used for $\mathcal{A}_4$. We represent a string as an asterisk $*$ followed by a sequence of characters. The asterisk $*$ represents any string (ϵ included). For example, $*\,abc$ represents all the strings which end with "abc", including "abc" itself. Instead, $*$ represents all possible strings, since the fixed suffix is empty: there is no constraint on how the string should end. Some examples of this notation (supposing $K = \{a, b, c, d, e\}$):

- $*\,abc \;= \{\, abc, aabc, babc, cabc, dabc, eabc, aaabc, ababc, acabc, \dots \}$
- $*\,ab \;= \{\, ab, aab, bab, cab, dab, eab, aaab, abab, acab, \dots \}$
- $*\,d \;= \{\, d, ad, bd, cd, dd, ed, aad, abd, acd, add, aed, bad, \dots \}$
- $* = \{\epsilon, a, b, c, d, e, aa, ab, ac, ad, ae, ba, \dots \}$

Since the asterisk $*$ at the beginning of the representation is always present, a particular string representation is uniquely identified only through its suffix. Thus, our domain is the following:

$$\mathcal{A}_5 = K^{\overrightarrow{+}} \cup \bot$$

An element of our domain is a sequence of characters (taken from our alphabet K) representing a string suffix. The asterisk $*$, which should always precede the sequence of characters, has been omitted to lighten the notation. So, the element "abc" $\in \mathcal{A}_5$ corresponds to $*\,abc$.

The **partial order** on this domain is:

$$\boxed{\begin{aligned} S \leq_{\mathcal{A}_5} T \Leftrightarrow (len(T) \leq len(S) \wedge \forall i \in [0, len(T) - 1] : T[i] \\ = S[i + len(S) - len(T)]\,) \;\vee S = \bot \end{aligned}}$$

Let us explain this definition. An abstract string S is smaller (in our order) than another abstract string T if T is a suffix of S. For T to be a suffix of S, we need to validate two conditions:

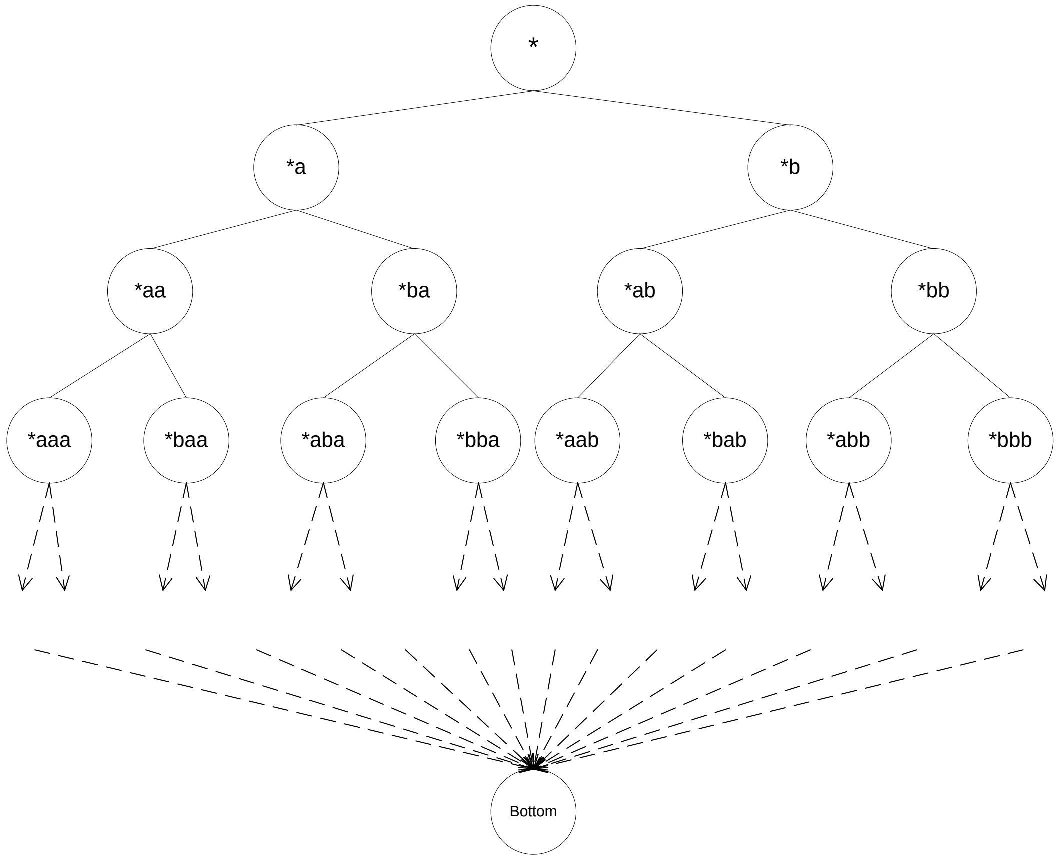
3. $len(T) \leq len(S)$
    a. that is, T must contain less or equal characters than S;
4. $T[i] = S[i + len(S) - len(T)]\ \forall i \in [0, len(T) - 1]$
    a. that is, the characters of T must be equal to the corresponding *last* $len(T)$ characters of S;

Otherwise, $S \leq_{\mathcal{A}_4} T$ if S is the bottom $\perp$ of the domain.

Some examples of this order:

- Consider $S = *\,abc$ and $T = *\,bc$; $len(S) = 3$ and $len(T) = 2$, so $len(T) < len(S)$. Moreover, $T[0] = b = S[0 + 3 - 2] = S[1]$ and $T[1] = c = S[1 + 3 - 2] = S[2]$. Then, $S \leq_{\mathcal{A}_4} T$. This is reasonable: the strings represented by S are a subset of those represented by T. In fact, the strings ending with bc include those ending with abc.
- Consider $S = *\,d$ and $T = *$; $len(S) = 1$ and $len(T) = 0$. We have $len(T) < len(S)$ but we do not have to compare any characters, since $len(T) = 0$. Then, $S \leq_{\mathcal{A}_4} T$. In fact, the strings ending with d are a subset of all possible strings.
- Consider $S = *\,abc$ and $T = *\,abd$; $len(S) = lent(T) = 3$. We have that $T[0] = a = S[0 + 3 - 3], T[1] = b = S[1 + 3 - 3]$ but $T[2] = d \neq S[2 + 3 - 3] = c$. Thus, $S \nleq_{\mathcal{A}_4} T$ and $T \nleq_{\mathcal{A}_4} S$. In fact, each string ending with abc is different from any string ending with abd. The two abstract elements S and T have nothing in common.

Let $K = \{‘a’, ‘b’\}$ (for representation clarity), then the **lattice** of $\mathcal{A}_5$ is the following:



Note that this is an *infinite* domain. In fact, given any suffix, we can always add a character at its beginning, thus obtaining a new suffix, longer (and smaller according to our order $\leq_{\mathcal{A}_5}$) than the first one. As happened with $\mathcal{A}_4$, this domain respects the ACC condition (see Chapter 2). In fact, given a certain suffix S, where $len(S) = n$, the ascending chain starting at S is the following:

$$S \rightarrow S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n$$

Where $S_1 = withoutFirst(S)$ (that is, S_1 corresponds to S without its first character; thus S_1 has length $n-1$; we defined $withoutFirst$ in paragraph 2.2), $S_2 = withoutFirst(S_1)$ (thus $len(S_2) = n-2$), $S_3 = withoutFirst(S_2)$ (thus $len(S_3) = n-3$), and so on, until we reach $S_n = withoutFirst(S_{n-1}) = \epsilon^l$ (since

$len(S_n) = n - n = 0$). S_n corresponds to an empty suffix: it is $*$, which represents any string, the top of our domain. Thus, given any suffix S of length n (which is finite), the ascending chain starting at S has finite length $n + 1$. On the opposite, our domain does not satisfy the DCC condition.

The **least upper bound** operator is rather easy. Given two suffixes, their least upper bound is their longest common suffix (formally defined in paragraph 2.3). If the two suffixes do not have anything in common, the least upper bound is $*$ (the suffix is empty). Examples:

- $\sqcup\,(*\,ba,\ *\,cba) = *\,ba$
- $\sqcup\,(*\,cba,\ *\,eda) = *\,a$
- $\sqcup\,(*\,cbd,\ *\,cba) =\ *$

The **greater lower bound** is defined as follows:

$$\sqcap\,(S_1, S_2) = \begin{cases} S_1 \text{ if } S_1 \leq_{\mathcal{A}_5} S_2 \\ S_2 \text{ if } S_2 \leq_{\mathcal{A}_5} S_1 \\ \bot \ \text{ otherwise} \end{cases}$$

The motivation is the same used in $\mathcal{A}_4$: if the two suffixes are not comparable with respect to our order (think about $*\,a$ and $*\,b$, for example), then the strings sets they represent have nothing in common. Their greatest lower bound is thus $\bot$. If they are comparable, it is immediate to see why the smaller element is the greatest lower bound. Examples:

- $\sqcap\,(*\,ba,\ *\,cba) = *\,cba$
- $\sqcap\,(*\,cba,\ *\,eda) = \bot$
- $\sqcap\,(*\,edcba,\ *\,a) = *\,edcba$

We can see from the lattice above that **top** and **bottom** are, respectively:

- $\top = *$, since $*$ is the empty suffix, which is suffix of any other suffix; $\top$ represents any possible string;

- $\perp$, for definition of $\mathcal{A}_5$;

Now we want to define the abstraction function of this domain. As always, a string can have multiple abstractions: for example, the string “abc” can be represented (in our abstract domain $\mathcal{A}_5$) as $*\,c$ or as $*\,bc$ or as $*\,abc$ or even as $*$. The representation which contains the most information about the string is: $*\,abc$. In this representation we use the longest suffix possible: the entire string. Note that this is the smallest value (that is, the more precise) of all those associated to “abc”. In fact, with respect to our order $\leq_{\mathcal{A}_5}$, we have:

$$*\,abc \;\leq_{\mathcal{A}_5} *\,c$$

$$*\,abc \;\leq_{\mathcal{A}_5} *\,bc$$

$$*\,abc \;\leq_{\mathcal{A}_5} *$$

The function which abstracts a single string is thus the following:

$$\alpha'_5(s) = \,*\,s$$

The **abstraction function** has to take us from a set of strings S (that is, an element of $\mathcal{D}$) to an element of $\mathcal{A}_5$. We can abstract all the strings in the set and then compute the least upper bound of them in $\mathcal{A}_5$:

$$\boxed{\alpha_5(S) = \coprod_{s\in S} \alpha'_5(s)}$$

This means that we take the longest common suffix amongst all the strings in the set S. Some examples:

- $\alpha(\{a, cba, feda, aaa\}) = \sqcup\,(*\,a, *\,cba, *\,feda, *\,aaa) = *\,a$
- $\alpha(\{a, cba, feda, aab\}) = \sqcup\,(*\,a, *\,cba, *\,feda, *\,aab) = * = \top$

The **concretization function** is:

$$\boxed{\begin{aligned}\gamma(S) = \{\, s : s \text{ is a string} \wedge\ len(s) \geq len(S) \wedge \forall i \in [0, len(S)-1] :\ S[i] \\ = s[i + len(s) - len(S)] \,\}\end{aligned}}$$

This means that the abstract value S maps to the set of the strings which end with the sequence of characters represented by S. Note that this set contains an infinite number of elements, for example (supposing $K = \{a, b\}$):

$$\gamma(*\, a) = \{\, a, aa, ba, aaa, baa, aba, bba, aaaa, baaa, abaa, bbaa, \dots \}$$

The **widening operator** is the following:

$$\nabla : (\mathcal{A}_5, \mathcal{A}_5) \to \mathcal{A}_5$$

$$\nabla\big(\alpha(\mathrm{s1}), \alpha(\mathrm{s2})\big) = \sqcup\ \big(\alpha(s1), \alpha(s2)\big)$$

because, in domains which satisfy ACC, the least upper bound converges in finite time, so it is also a widening operator. We proved that our domain $\mathcal{A}_5$, though having infinite height, satisfies ACC; so we can use the least upper bound operator as widening operator.

As for the **semantics** of this abstract domain, let us see some valid “string/abstract representation of such string” associations:

- $"abcdef" \in \gamma(\, *\, abcdef)$
- $"abcdef" \in \gamma(*\, bcdef)$
- $"abcdef" \in \gamma(*\, cdef)$
- $"abcdef" \in \gamma(*\, def)$
- $"abcdef" \in \gamma(*\, ef)$
- $"abcdef" \in \gamma(*\, f)$
- $"abcdef" \in \gamma(*)$

For example, $"abcdef" \notin\ \gamma(*\, babcd)$, since $"abcdef"$ does not end with a ‘d’, as the prefix $*\, babcd$ imposes.

Now we are going to see the semantics of the string operators, with the usual notation.

7.2.1 String(char[] data)

In the string constructor, the suffix will be made by the entire string, preceded by the asterisk $*$. Consider for example the following code:

```
char[] data = {'a', 'b', 'c'};
String s = new String(data);
```

The semantics is:

$$S\big[[new\ String, \text{arrayChars}]\big] = * \text{ arrayChars}$$

In our example above, the resulting abstract value is $*$ abc.

Obviously if we don't know what `data` contains (think about user input), then the suffix will be empty and the semantics will thus be:

$$S\big[[\text{new String, arrayChars}]\big] = *$$

7.2.2 concat(String str)

This operator takes two strings and concatenates them. For example, if `s1` = "`hello` " and `s2` = "`world`" then `s1.concat(s2)` = "`hello world`". Suppose that $\overline{s1}$ is the abstract representation of `s1` and that $\overline{s2}$ is the abstract representation of `s2`. The semantics of this operator is:

$$S\left[[\text{concat}, \overline{s1}, \overline{s2}]\right] = \overline{s2}$$

In fact, when we concatenate two strings we create a new string which ends with the second one. The suffix of the resulting string will then be the same one as the suffix of the second string. Sometimes, this operation makes us lose information. In the example cited above, we have:

$$\overline{s1} = *\,"hello\ "$$

$$\overline{s2} = *\,"\ world"$$

The abstract value resulting from concatenation of `s1` and `s2` is:

$$\overline{s_{res}} = *\,"\ world"$$

when, actually, we know something more ($\overline{s_{res}}$ should be $*\,"hello\ world"$). This happens because, in this particular case, the asterisk of `s2` did not represent any character. The abstract approximation caused the loss of information.

7.2.3 indexOf(int c)

This operator returns the index within the string of the first occurrence of the specified character. If such character occurs in our suffix, then we know that the answer will be greater than -1. Unfortunately, though, we do not have any other information, since we do not know the index of such character. Otherwise, we are forced to return $\top$ (supposing that we are working in collaboration with a numerical abstract domain), because we do not know if the character is masked by the asterisk $*$ or not.

The semantics of this operator in our domain will thus be:

$$S[[\text{indexof}, \overline{s}, \text{c}]] = \begin{cases} > -1 & \text{if } \exists i : \overline{s}[i] = c \\ \top & \text{otherwise} \end{cases}$$

7.2.4 lastIndexOf(int c)

This operator returns the index within the string of the last occurrence of the specified character. If the character occurs in the suffix of our abstract value and its last occurrence is at index i, then we know that the index of its last occurrence in the string will be $\geq i$. We cannot know the precise index of the last occurrence, though, because we do not know how many characters precede the suffix.

If the character does not occur in the suffix of the abstract value, instead, we are forced to return ⊤. In fact, we do not know if the character will appear in "$*$ part" of the string (and where) or not.

The semantics is thus the following:

$$S[\![\text{lastIndexOf}, \overline{s}, \text{c}]\!] = \begin{cases} \geq i & \text{if } \exists i : \overline{s}[i] = c \land \nexists j > i : \overline{s}[j] = c \\ \top \text{ otherwise} & \end{cases}$$

7.2.5 substring(int beginIndex, int endIndex)

This operator returns a new string (`s2`) that is a substring of this string (`s1`). The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`. Unfortunately, with this domain, we will always have to return ⊤. In fact, `beginIndex` and `endIndex` are relative to the beginning of the string: "index 1" means "the second character *from the beginning* of the string". Since in this domain we do not know how the string begins, we also do not know the indices of the characters which form the suffix. The semantics will thus be:

$$S\left[\!\left[\text{substring}, \overline{s1}, \text{beginIndex,endIndex}\right]\!\right] = \top$$

7.2.6 contains(CharSequence seq)

This operator returns true if and only if the string (`s1`) contains the specified sequence of char values (`seq`). If the sequence of characters `seq` contains more characters than the length of our suffix, we will certainly have to return ⊤ (the top element of a boolean domain). Otherwise, we can check the suffix to see if the sequence `seq` appears in it. In conclusion, the results of this method will be the following:

$$S\left[\!\left[\text{contains}, \overline{s1}, \text{seq}\right]\!\right] = \begin{cases} true & \text{if } len(\text{seq}) \leq len\left(\overline{s1}\right) \land \exists i : \text{seq} = \overline{s1}[i \cdots i + len(\text{seq}) - 1] \\ \top & \text{otherwise} \end{cases}$$

We return `true` if the character sequence `seq` is present in the suffix, otherwise we return `T`.

8 Abstract domain based on regular expressions

Until now, we saw five abstract domains:

1. The domain of *certainly contained characters*; the string was approximated through the characters we knew the string contained for sure;
2. The domain of *maybe contained characters* (the complement of the certainly not contained characters); the string was approximated through the characters we knew it may have contained; the characters not present in such approximation were surely not contained in the string;
3. The *combination of the first two* domains (certainly contained and maybe contained);
4. The *prefix* domain; the string was approximated through the known prefix, while the suffix was unknown;
5. The *suffix* domain; the string was approximated through the known suffix, while the prefix was unknown.

In the first three domains, the only focus was character inclusion. Each character contained in those representations was completely unrelated with regards to the others. We did not maintain any information about order between characters. In the last two domains (prefix and suffix) we also considered order, but limited at the beginning (or at the end) of the string.

The new abstract domain we will define in this paragraph will consider (just like domains $\mathcal{A}_4$ and $\mathcal{A}_5$) both inclusion and order among characters, but not limited to start/end of the string. For example, we will be able to track information like '*this string begins with "SELECT ", then it has some characters which we do not know, then it continues with " FROM TABLE T1" and then it ends with some other characters which we do not know* '. This abstract domain is based on the idea of identifying a string through a regular expression. The regular expressions we described in Chapter 2 are too much complex for our purposes: we could not build an efficient (or even convergent) analysis using their full expressivity; thus we will have to use a simplification of them.

8.1 Brick

We will approximate a string through the combination of simple elements, which are different instantiations of one simple "*construction brick*". The brick is defined as follows:

$$\boxed{\mathcal{B} = [S]^{min,max}}$$

The brick is made of a string set S (for example: $S = \{\, 'mo', 'de' \,\}$) and two integer numbers (min and max, for example 1 and 2). max must be greater or equal than min: $max \geq min$. This brick represents all the strings which can be built through the strings in S, taken between min and max times altogether. For example:

$$[\{'mo', 'de'\}]^{1,2} = \{\, mo, de, momo, dede, mode, demo \,\}$$

$$[\{'abc'\}]^{0,1} = \{\, \epsilon, abc \,\}$$

$$[\{'a'\}]^{0,+\infty} = \{\, \epsilon, a, aa, aaa, aaaa, \dots \,\}$$

$$[\{'demo'\}]^{1,1} = \{\, demo \,\}$$

Moreover, we impose that:

$$[\emptyset]^{0,0} = \{\, \epsilon \,\}$$

$$[\emptyset]^{(m,M)\neq(0,0)} = \emptyset$$

$$[S \neq \emptyset]^{0,0} = \emptyset$$

8.2 String representation

In the examples above, we considered only one brick at a time. But we can also consider combination of bricks. In particular, we choose to represent strings as ordered lists of bricks. For example:

$$[\{'straw'\}]^{1,1}[\{'berry'\}]^{1,1} = \{\, strawberry \,\}$$

$$[\{'straw'\}]^{0,1}[\{'berry'\}]^{1,1} = \{\, berry, strawberry \,\}$$

$$[\{'straw'\}]^{1,1}[\{'berry'\}]^{0,1} = \{\, straw, strawberry \,\}$$

$$[\{'straw'\}]^{0,1}[\{'berry'\}]^{0,1} = \{\, \epsilon, straw, strawberry, berry \,\}$$

To be more formal, we define the concatenation between bricks like this:

$$\mathcal{B}_1\mathcal{B}_2 = \{\, \alpha\beta : \alpha \in strings(\mathcal{B}_1) \wedge \beta \in strings(\mathcal{B}_2) \,\}$$

where $strings(\mathcal{B})$ represents all the strings which can be built from brick $\mathcal{B}$, and $\alpha\beta$ is the string concatenation we already defined in Chapter 2. Concatenation between bricks is associative:

$$\mathcal{B}_1\mathcal{B}_2\mathcal{B}_3 = (\mathcal{B}_1\mathcal{B}_2)\mathcal{B}_3 = \mathcal{B}_1(\mathcal{B}_2\mathcal{B}_3)$$

Now, let us consider different possible representations of a certain string (for example $'look'$) through this “brick mechanism”:

- $[l]^{1,1}[o]^{1,1}[o]^{1,1}[k]^{1,1}$
- $[lo]^{1,1}[o]^{1,1}[k]^{1,1}$
- $[l]^{1,1}[o]^{1,1}[ok]^{1,1}$
- $[l]^{1,1}[oo]^{1,1}[k]^{1,1}$
- $[l]^{1,1}[o]^{2,2}[k]^{1,1}$
- $[look]^{1,1}$

All those representations model the same string ($'look'$). Deciding which one is the best is a subjective procedure. We could prefer having fewer possible blocks in the list; or we could prefer blocks with the simplest sets of strings; or we could prefer blocks risen at the smallest range ($max - min$).

In order to find a *canonical form* for our string representation, we are now going to see five normalizing rules. We can reach the normal form of a certain representation applying repeatedly those rules, until no more rules can be further applied. The normal representation can be seen as the fixpoint of the application of our five rules to a given representation.

8.2.1 Rule 1

The first rule removes unnecessary blocks. An unnecessary block has the form:

$$[\emptyset]^{0,0}$$

In fact, this block represents only the empty string ($[\emptyset]^{0,0} = \{\epsilon\}$), which is the neutral element of the concatenation operation.

$\boldsymbol{\mathcal{R}_1}$: given a block list $\mathcal{L}$, remove from $\mathcal{L}$ all the blocks of the form $[\emptyset]^{0,0}$.

Let us see an example. Consider the block list $\mathcal{L} = [\mathcal{B}_1; \mathcal{B}_2; \mathcal{B}_3]$ where $\mathcal{B}_1 = [a]^{1,1}$, $\mathcal{B}_2 = [\emptyset]^{0,0}$ and $\mathcal{B}_3 = [b]^{0,2}$. Then $\mathcal{L} = \{a, ab, abb\}$. Applying our rule $\boldsymbol{\mathcal{R}_1}$, we obtain $\mathcal{L}' = [\mathcal{B}_1; \mathcal{B}_3]$, having removed the block $\mathcal{B}_2$. $\mathcal{L}' = \{a, ab, abb\} = \mathcal{L}$: the set of strings represented by the list of blocks does not change.

8.2.2 Rule 2

This rule merges successive blocks with the same indices, $min = 1$ and $max = 1$.

$\boldsymbol{\mathcal{R}_2}$: given a block list $\mathcal{L}$, where $\mathcal{B}_i = [S_i]^{1,1}$ and $\mathcal{B}_{i+1} = [S_{i+1}]^{1,1}$, merge $\mathcal{B}_i$ and $\mathcal{B}_{i+1}$ in a new single block $\mathcal{B}_{i_{new}}$ of the following form:

$$\mathcal{B}_{i_{new}} = [S_i S_{i+1}]^{1,1}$$

where $S_i S_{i+1}$ represents the concatenation between sets of strings (defined in Chapter 2, in the paragraph about regular expressions).

Let us see some examples to clarify this rule.

- Consider the block list $\mathcal{L} = [\mathcal{B}_1; \mathcal{B}_2; \mathcal{B}_3]$ where $\mathcal{B}_1 = [a]^{1,1}$, $\mathcal{B}_2 = [c]^{1,1}$ and $\mathcal{B}_3 = [b]^{0,2}$. Then $\mathcal{L} = \{ac, acb, acbb\}$. Applying our rule $\boldsymbol{\mathcal{R}_2}$, we merge $\mathcal{B}_1$ and $\mathcal{B}_2$ in $\mathcal{B}_{1_{new}} = [\{a\}\{c\}]^{1,1} = [ac]^{1,1}$. Thus $\mathcal{L}' = [\mathcal{B}_{1_{new}}; \mathcal{B}_3] = \{ac, acb, acbb\}$.
- Consider the block list $\mathcal{L} = [\mathcal{B}_1; \mathcal{B}_2]$ where $\mathcal{B}_1 = [\{a, cd\}]^{1,1}$ and $\mathcal{B}_2 = [\{b, ef\}]^{1,1}$. Then $\mathcal{L} = \{ab, aef, cdb, cdef\}$. Applying our rule $\boldsymbol{\mathcal{R}_2}$, we

merge $\mathcal{B}_1$ and $\mathcal{B}_2$ in $\mathcal{B}_{1_{new}} = [\{a, cd\}\{b, ef\}]^{1,1} = [\{ab, aef, cdb, cdef\}]^{1,1}$. Thus $\mathcal{L}' = \left[\mathcal{B}_{1_{new}}\right] = \{ab, aef, cdb, cdef\}$.

This rule has the purpose to decrease the number of blocks in the list.

8.2.3 Rule 3

This rule transforms a block in which the number of applications is constant ($min = max$) into one in which the indices are 1 ($min = max = 1$).

$\boldsymbol{\mathcal{R}_3}$: given a block list $\mathcal{L}$ in which $\mathcal{B}_i = [S_i]^{m,m}$ and $m > 1$, transform $\mathcal{B}_i$ into $\mathcal{B}_{i_{new}}$ of the following form:

$$\mathcal{B}_{i_{new}} = [S_i^m]^{1,1}$$

where $S_i^m = \underbrace{S_i S_i \ldots S_i}_{m \text{ times}}$.

Some examples:

- $\mathcal{B} = [a]^{3,3}$ becomes $\mathcal{B}_{new} = [aaa]^{1,1}$
- $\mathcal{B} = [\{a, b\}]^{2,2}$ becomes $\mathcal{B}_{new} = [\{a, b\}^2]^{1,1} = [\{a, b\}\{a, b\}]^{1,1} = [\{aa, ab, ba, bb\}]^{1,1}$
- $\mathcal{B} = [\{ab, cd\}]^{2,2}$ becomes $\mathcal{B}_{new} = [\{ab, cd\}^2]^{1,1} = [\{ab, cd\}\{ab, cd\}]^{1,1} = [\{abab, abcd, cdab, cdcd\}]^{1,1}$

This rule has the purpose of simplifying the indices of the blocks in the list (we prefer $min = max = 1$, when possible).

8.2.4 Rule 4

This rule merges two successive blocks in which the set of strings is the same ($S_i = S_{i+1}$) into a single one (modifying the indices, of course).

$\boldsymbol{\mathcal{R}_4}$: given a block list $\mathcal{L}$, where $\mathcal{B}_i = [S_i]^{m_1,M_1}$ and $\mathcal{B}_{i+1} = [S_i]^{m_2,M_2}$, merge $\mathcal{B}_i$ and $\mathcal{B}_{i+1}$ in a new single block $\mathcal{B}_{i_{new}}$ of the following form:

$$\mathcal{B}_{i_{new}} = [S_i]^{m_1+m_2,M_1+M_2}$$

Some examples:

- Consider the block list $\mathcal{L} = [\mathcal{B}_1; \mathcal{B}_2]$ where $\mathcal{B}_1 = [a]^{1,3} = \{a, aa, aaa\}$ and $\mathcal{B}_2 = [a]^{1,1} = \{a\}$. Then $\mathcal{L} = \{a, aa, aaa\}\{a\} = \{aa, aaa, aaaa\}$. Applying our rule $\boldsymbol{\mathcal{R}_4}$, we merge $\mathcal{B}_1$ and $\mathcal{B}_2$ in $\mathcal{B}_{1_{new}} = [a]^{1+1,3+1} = [a]^{2,4}$. Thus $\mathcal{L}' = \mathcal{B}_{1_{new}} = [a]^{2,4} = \{aa, aaa, aaaa\} = \mathcal{L}$.
- Consider the block list $\mathcal{L} = [\mathcal{B}_1; \mathcal{B}_2]$ where $\mathcal{B}_1 = [\{a, b\}]^{0,2} = \{\epsilon, a, b, aa, ab, ba, bb\}$ and $\mathcal{B}_2 = [\{a, b\}]^{1,1} = \{a, b\}$. Then $\mathcal{L} = \{\epsilon, a, b, aa, ab, ba, bb\}\{a, b\} = \{a, aa, ba, aaa, aba, baa, bba, b, ab, bb, aab, abb, bab, bbb\}$. Applying our rule $\boldsymbol{\mathcal{R}_4}$, we merge $\mathcal{B}_1$ and $\mathcal{B}_2$ in $\mathcal{B}_{1_{new}} = [\{a, b\}]^{0+1,2+1} = [\{a, b\}]^{1,3}$. Thus $\mathcal{L}' = \mathcal{B}_{1_{new}} = [\{a, b\}]^{1,3} = \{a, b\} \cup \{a, b\}^2 \cup \{a, b\}^3 = \{a, b\} \cup \{aa, ab, ba, bb\} \cup \{aaa, aab, aba, abb, baa, bab, bba, bbb\} = \mathcal{L}$.

This rule has the same purpose as rule $\boldsymbol{\mathcal{R}_2}$, that is to reduce the number of blocks in the list.

8.2.5 Rule 5

This rule breaks a single block with high indices ($min \geq 1, max > min$) into two simpler blocks.

$\boldsymbol{\mathcal{R}_5}$: given a block list $\mathcal{L}$ in which $\mathcal{B}_i = [S_i]^{min,max}$, $min \geq 1$ and $max \neq min$, transform $\mathcal{B}_i$ into $\mathcal{B}_{i_1}\mathcal{B}_{i_2}$ of the following forms:

$$\mathcal{B}_{i_1} = \left[S_i^{min}\right]^{1,1}$$

$$\mathcal{B}_{i_2} = [S_i]^{0,max-min}$$

We can see that $\mathcal{B}_i$ and $\mathcal{B}_{i_1}\mathcal{B}_{i_2}$ are equivalent. In fact:

$$\begin{aligned}
\mathcal{B}_{i_1}\mathcal{B}_{i_2} &= \left[S_i^{min}\right]^{1,1}[S_i]^{0,max-min} \\
&= S_i^{min}S_i^0 \cup S_i^{min}S_i^1 \cup \ldots \cup S_i^{min}S_i^{max-min} \text{ (by def. of brick and brick concatenation)} \\
&= S_i^{min+0} \cup S_i^{min+1} \cup \ldots \cup S_i^{min+(max-min)} \text{ (since } S^iS^j = S^{i+j}\text{)} \\
&= S_i^{min} \cup S_i^{min+1} \cup \ldots \cup S_i^{max} \text{ (indices semplification)} \\
&= [S_i]^{min,max} = \mathcal{B}_i \text{ (by def. of brick)}
\end{aligned}$$

Let us see an example. Consider $\mathcal{B} = [a]^{2,5} = \{aa, aaa, aaaa, aaaaa\}$. Using rule $\boldsymbol{\mathcal{R}_5}$ we obtain $\mathcal{B}_1 = [a^2]^{1,1} = [aa]^{1,1} = \{aa\}$ and $\mathcal{B}_2 = [a]^{0,3} = \{\epsilon, a, aa, aaa\}$. $\mathcal{B}_1\mathcal{B}_2 = \{aa\}\{\epsilon, a, aa, aaa\} = \{aa, aaa, aaaa, aaaaa\} = \mathcal{B}$.

This rule, although increasing the number of blocks of the representation, reduces their indices.

To see an example of the normalizing process, consider the representation $[l]^{1,1}[o]^{1,1}[o]^{1,1}[k]^{1,1}$ for the string $look$. Applying rule $\boldsymbol{\mathcal{R}_2}$ to the first two bricks ($[l]^{1,1}[o]^{1,1}$) we obtain:

$$[lo]^{1,1}[o]^{1,1}[k]^{1,1}$$

Then, we can apply $\boldsymbol{\mathcal{R}_2}$ again to $[lo]^{1,1}[o]^{1,1}$:

$$[loo]^{1,1}[k]^{1,1}$$

And finally we apply $\boldsymbol{\mathcal{R}_2}$ to $[loo]^{1,1}[k]^{1,1}$, thus obtaining:

$$[look]^{1,1}$$

This representation is the fixpoint of the application of our rules; in fact we cannot apply any more rules to it. This example helps us understand that, given a constant string s, its best representation is $[s]^{1,1}$: the number of blocks is the smallest possible (1 block), its indices are the simplest ($min = max = 1$) and the set of strings of the block contains only one element (the string itself, s).

We can also see a little more complex example of application of our rules. Consider the representation:

$$\mathcal{L} = [\mathcal{B}_1; \mathcal{B}_2; \mathcal{B}_3; \mathcal{B}_4; \mathcal{B}_5]$$

where:

$$\mathcal{B}_1 = [a]^{1,1} = \{a\}$$

$$\mathcal{B}_2 = [\emptyset]^{0,0} = \{\epsilon\}$$

$$\mathcal{B}_3 = [\{a, b\}]^{2,3} = \{aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\}$$

$$\mathcal{B}_4 = [\{a, b\}]^{0,1} = \{\epsilon, a, b\}$$

$$\mathcal{B}_5 = [c]^{3,3} = \{ccc\}$$

We start applying rule $\boldsymbol{\mathcal{R}_1}$: we remove $\mathcal{B}_2$, since it only represents ϵ. We obtain:

$$\mathcal{L}_1 = [\mathcal{B}_1; \mathcal{B}_3; \mathcal{B}_4; \mathcal{B}_5]$$

Rule $\boldsymbol{\mathcal{R}_2}$ cannot be applied now, because there are not two consecutive blocks with $min = max = 1$. We can apply $\boldsymbol{\mathcal{R}_3}$ instead: block $\mathcal{B}_5$ can be simplified to a new block with $min = max = 1$:

$$\mathcal{B}_5 = [c]^{3,3} \Rightarrow \mathcal{B}_{5_{new}} = [c^3]^{1,1} = [ccc]^{1,1}$$

The new representation is:

$$\mathcal{L}_2 = [\mathcal{B}_1; \mathcal{B}_3; \mathcal{B}_4; \mathcal{B}_{5_{new}}]$$

Now we can apply $\boldsymbol{\mathcal{R}_4}$, merging $\mathcal{B}_3$ and $\mathcal{B}_4$ (since they are made of the same strings set):

$$\mathcal{B}_3 = [\{a, b\}]^{2,3}, \mathcal{B}_4 = [\{a, b\}]^{0,1} \Rightarrow \mathcal{B}_{34} = [\{a, b\}]^{2+0,3+1} = [\{a, b\}]^{2,4}$$

The new representation is:

$$\mathcal{L}_3 = [\mathcal{B}_1; \mathcal{B}_{34}; \mathcal{B}_{5_{new}}] = [a]^{1,1}[\{a, b\}]^{2,4}[ccc]^{1,1}$$

Now we can apply $\boldsymbol{\mathcal{R}_5}$ to block $\mathcal{B}_{34}$, obtaining:

$$\mathcal{B}_{34} = [\{a, b\}]^{2,4} \Rightarrow \mathcal{B}_{34_1} = [\{a, b\}^2]^{1,1}, \mathcal{B}_{34_2} = [\{a, b\}]^{0,4-2}$$

$$\mathcal{B}_{34_1} = [\{a, b\}^2]^{1,1} = [\{aa, ab, ba, bb\}]^{1,1}$$

$$\mathcal{B}_{34_2} = [\{a, b\}]^{0,2}$$

Thus reaching the representation:

$$\mathcal{L}_4 = [\mathcal{B}_1; \mathcal{B}_{34_1}; \mathcal{B}_{34_2}; \mathcal{B}_{5_{new}}] = [a]^{1,1}[\{aa, ab, ba, bb\}]^{1,1}[\{a, b\}]^{0,2}[ccc]^{1,1}$$

Now we can apply rule $\boldsymbol{\mathcal{R}_2}$ to the first two blocks, $\mathcal{B}_1$ and $\mathcal{B}_{34_1}$, obtaining $\mathcal{B}_{1_{new}}$:

$$\mathcal{B}_{1_{new}} = [\{a\}\{aa, ab, ba, bb\}]^{1,1} = [\{aaa, aab, aba, abb\}]^{1,1}$$

The new representation is:

$$\mathcal{L}_5 = [\mathcal{B}_{1_{new}}; \mathcal{B}_{34_2}; \mathcal{B}_{5_{new}}] = [\{aaa, aab, aba, abb\}]^{1,1}[\{a, b\}]^{0,2}[ccc]^{1,1}$$

We cannot apply any more rules. This means that we have reached the normal form of the representation $\mathcal{L}$. It can be seen that $\mathcal{L}$ and $\mathcal{L}_5$ model the exactly same set of strings.

Finally, we can do some considerations about the normalized form of our domain elements. In a normalized element of our domain, there cannot be two successive blocks with both $min = max = 1$. In fact, they would be merged into one single block by the $\boldsymbol{\mathcal{R}_2}$ rule. Moreover, we cannot have a block with $min \geq 1, max > min$, since it would be simplified by $\boldsymbol{\mathcal{R}_3}$ or $\boldsymbol{\mathcal{R}_5}$ the rule. Also, we cannot have blocks with indices $min = 0$ and $max = 0$: they would be removed by rule $\boldsymbol{\mathcal{R}_1}$. Thus, every block of the normalized list will have one of the following two forms:

- $min = 0 \wedge max > 0$
- $min = 1 \wedge max = 1$

For this reason, our lists are made of blocks like $[T]^{1,1}$ or $[T]^{0,max>0}$, where T is a generic set of strings.

8.3 Lattice

We are now ready to formally define our lattice. Our abstract domain based on regular expressions is the following:

$$\mathcal{A}_6 = \mathcal{B}^{\vec{+}}$$

that is the set of all finite lists composed of bricks (elements in $\mathcal{B}$). An element of $\mathcal{A}_6$ is a list of bricks.

To define a partial order on this domain (which means a partial order on lists of bricks) we have to define a **partial order on bricks** first. We start considering two kinds of special bricks, $\mathcal{B}^{\top}$ and $\mathcal{B}^{\perp}$.

Given an alphabet of characters (K), the definition of $\mathcal{B}^{\top}$ is the following one:

$$\boxed{\mathcal{B}^{\top} = [K]^{0,+\infty}}$$

This bricks clearly represents all the possible strings, since it represents the strings which can be built taking all the characters in K and repeating them an arbitrary number of times (from 0 to $+\infty$).

The definition of $\mathcal{B}^{\perp}$ is the following one:

$$\boxed{\mathcal{B}^{\perp} = [\emptyset]^{(m,M)\neq(0,0)} \vee \left([S]^{min,max}, max < min\right) \vee [S \neq \emptyset]^{0,0}}$$

The three possible definitions are all bricks which do not represent *any* string. They are invalid bricks and they correspond to $\emptyset$.

Thus, given two bricks $\mathcal{B}_1 = [C_1]^{min_1,max_1}$ and $\mathcal{B}_2 = [C_2]^{min_2,max_2}$, we have:

$$\boxed{\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_2 \Leftrightarrow (\ C_1 \subseteq C_2 \wedge min_1 \geq min_2 \wedge max_1 \leq max_2\ ) \vee \mathcal{B}_2 = \mathcal{B}^{\top} \vee \mathcal{B}_1 = \mathcal{B}^{\perp}}$$

The second part of the formula on the right ($\mathcal{B}_2 = \mathcal{B}^{\top}$) takes in consideration the special brick $\mathcal{B}^{\top}$, which is, by definition, greater or equal than any other brick. The third part of the formula on the right ($\mathcal{B}_1 = \mathcal{B}^{\perp}$) takes into consideration the other special brick, $\mathcal{B}^{\perp}$, which is, by definition, less or equal than any other brick.

We are now going to see some examples of this partial order. We will consider couples of bricks ($\mathcal{B}_1$ and $\mathcal{B}_2$) and their concretizations $\gamma(\mathcal{B}_i)$ (which we already informally talked about but we will formalize later). We will check if $\mathcal{B}_1 \leq \mathcal{B}_2$; in such case we will show that $\gamma(\mathcal{B}_1) \subseteq \gamma(\mathcal{B}_2)$:

- Consider $\mathcal{B}_1 = [\{'mo','de'\}]^{1,1},\ \gamma(\mathcal{B}_1) = \{\ mo, de\ \}$ and $\mathcal{B}_2 = [\{'mo','de'\}]^{1,2},\ \gamma(\mathcal{B}_2) = \{\ mo, de, momo, dede, mode, demo\ \}$; $[\{'mo','de'\}]^{1,1} \leq_{\mathcal{B}} [\{'mo','de'\}]^{1,2}$ because $C_1 = C_2$, $min_1 = min_2$ and $max_1 < max_2$; in fact $\{\ mo, de\ \} \subseteq \{\ mo, de, momo, dede, mode, demo\ \}$;
- Consider $\mathcal{B}_1 = [\{'mo'\}]^{1,2},\ \gamma(\mathcal{B}_1) = \{\ mo, momo\}$ and $\mathcal{B}_2 = [\{'mo','de'\}]^{1,2},\ \gamma(\mathcal{B}_2) = \{\ mo, de, momo, dede, mode, demo\ \}$; $[\{'mo'\}]^{1,2} \leq_{\mathcal{B}} [\{'mo','de'\}]^{1,2}$ because $C_1 \subseteq C_2$, $min_1 = min_2$ and $max_1 = max_2$; in fact $\{\ mo, momo\} \subseteq \{\ mo, de, momo, dede, mode, demo\ \}$;
- Consider $\mathcal{B}_1 = [\{'ab'\}]^{2,3},\ \gamma(\mathcal{B}_1) = \{abab, ababab\}$ and $\mathcal{B}_2 = [\{'ab'\}]^{1,4}$, $\gamma(\mathcal{B}_2) = \{ab, abab, ababab, abababab\}$; $[\{'ab'\}]^{2,3} \leq_{\mathcal{B}} [\{'ab'\}]^{1,4}$ because $C_1 = C_2$, $min_1 > min_2$ and $max_1 < max_2$; in fact $\{abab, ababab\} \subseteq \{ab, abab, ababab, abababab\}$;

- Consider $\mathcal{B}_1 = [\{'c'\}]^{0,0},\ \gamma(\mathcal{B}_1) = \emptyset$ and $\mathcal{B}_2 = [\{'ab'\}]^{0,2},\ \gamma(\mathcal{B}_2) = \{\epsilon, ab, abab\}$; $[\{'c'\}]^{0,0} \leq_{\mathcal{B}} [\{'ab'\}]^{0,2}$ because $\mathcal{B}_1 = \mathcal{B}^{\perp}$; in fact $\emptyset \subseteq \{\epsilon, ab, abab\}$;
- Consider $\mathcal{B}_1 = [\{'mo', 'de'\}]^{1,2},\ \gamma(\mathcal{B}_1) = \{\, mo, de, mode, demo, dede, momo\}$ and $\mathcal{B}_2 = [\{'mo', 'de', 'a'\}]^{1,1},\ \gamma(\mathcal{B}_2) = \{\, mo, de, a\}$; $[\{'mo', 'de'\}]^{1,2} \nleq_{\mathcal{B}} [\{'mo', 'de', 'a'\}]^{1,1}$ because $max_1 > max_2$; $[\{'mo', 'de', 'a'\}]^{1,1} \nleq_{\mathcal{B}} [\{'mo', 'de'\}]^{1,2}$ because $C_2 \nsubseteq C_1$; in fact, neither $\{\, mo, de, mode, demo, dede, momo\} \subseteq \{\, mo, de, a\}$ nor $\{\, mo, de, a\} \subseteq \{\, mo, de, mode, demo, dede, momo\}$.

Now we can define an order on lists of bricks, the elements of our lattice. Given two lists $\mathcal{L}_1$ and $\mathcal{L}_2$, first of all we "augment" the shorter list with empty bricks ($\mathrm{E} = [\emptyset]^{0,0}$) in order to have lists of the same size. Then, we proceed extracting one block from each list and comparing the two blocks, up till the end of the two lists. Note the difference between E and $\mathcal{B}^{\perp}$: the first (E) is a valid brick which corresponds only to the empty string ϵ. As we will see later, $\gamma(\mathrm{E}) = \{\epsilon\}$. The second ($\mathcal{B}^{\perp}$) is instead an invalid brick, which does not correspond to any string, that is $\gamma(\mathcal{B}^{\perp}) = \emptyset$.

More formally, given two lists $\mathcal{L}_1$ and $\mathcal{L}_2$, we make them having the same size n adding empty bricks E to the shorter one, thus obtaining $\mathcal{L}_1'$ and $\mathcal{L}_2'$. Then:

$$\boxed{\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2 \Leftrightarrow (\mathcal{L}_1'[i] \leq_{\mathcal{B}} \mathcal{L}_2'[i]\ \forall i \in [1, n]) \vee (\mathcal{L}_2 = \mathcal{L}^{\top}) \vee (\mathcal{L}_1 = \mathcal{L}^{\perp})}$$

where $\mathcal{L}^{\top}$ is a list with only one element, $\mathcal{B}^{\top}$. Since $\mathcal{B}^{\top}$ represents all the strings, $\mathcal{L}^{\top}$ does too. $\mathcal{L}^{\perp}$ is the empty list (it does not represent any string at all, not even the empty string) or any list which contains at least one $\mathcal{B}^{\perp}$ (an invalid element).

Let us see an example. Consider the two lists:

$$\mathcal{L}_1 = ['SELECT\ ']^{1,1}[K]^{0,\infty}['\ FROM\ teachers']^{1,1}['\ WHERE\ name = Paul']^{0,1}$$

$$\mathcal{L}_2 = ['SELECT\ ']^{1,1}['age']^{1,1}['\ FROM\ teachers']^{1,1}$$

where K is our alphabet and $[K]^{0,\infty} = \mathcal{B}^{\top}$.

First of all, we make the second list the same size as the first one (4), adding one empty brick at its end:

$$\mathcal{L}'_1 = \mathcal{L}_1$$

$$\mathcal{L}'_2 = \mathcal{L}_2 + \mathrm{E}$$

Then, we compare each couple of bricks in the same position:

- $['SELECT\ ']^{1,1} \leq_{\mathcal{B}} ['SELECT\ ']^{1,1}$ (they are the same brick);
- $['age']^{1,1} \leq_{\mathcal{B}} [K]^{0,\infty} = \mathcal{B}^{\top}$ since any brick is less or equal than $\mathcal{B}^{\top}$;
- $['\ FROM\ teachers']^{1,1} \leq_{\mathcal{B}} ['\ FROM\ teachers']^{1,1}$ (they are the same brick);
- $\mathrm{E} \leq_{\mathcal{B}} ['\ WHERE\ name = Paul']^{0,1}$ because the second brick contains the empty string;

Since $\mathcal{L}'_2[i] \leq_{\mathcal{B}} \mathcal{L}'_1[i]\ \forall i \in [1,4]$, we can say that $\mathcal{L}_2 \leq_{\mathcal{A}_6} \mathcal{L}_1$.

Before going ahead and define the other details of our domain (top, bottom, least upper bound, etc…), we need to stop for a moment and make some considerations about our partial order. First of all, we want to be sure that it is truly a partial order. To do this, we must prove three properties:

- Reflexivity ($a \leq a$)
- Transitivity ($a \leq b \land b \leq c \Rightarrow a \leq c$)
- Antisymmetry ($a \leq b \land b \leq a \Rightarrow a = b$)

The first two properties (reflexivity and transitivity) characterize a **preorder**. Thus, a **partial order** can be thought of as an antisymmetric preorder.

Since our order $\leq_{\mathcal{A}_6}$ strongly relies on the order of bricks, $\leq_{\mathcal{B}}$, we are going to prove that $\leq_{\mathcal{B}}$ is a partial order first. Then we are going to prove that $\leq_{\mathcal{A}_6}$is a partial order, too.

8.3.1 Reflexivity of $\leq_{\mathcal{B}}$

Given a brick $\mathcal{B} = [S]^{min,max}$, we must prove that $\mathcal{B} \leq_{\mathcal{B}} \mathcal{B}$. We have three cases:

- If $\mathcal{B} = \mathcal{B}^{\top}$, then $\mathcal{B} \leq_{\mathcal{B}} \mathcal{B}$ for definition of $\leq_{\mathcal{B}}$ (every brick is less or equal than the top brick)
- If $\mathcal{B} = \mathcal{B}^{\perp}$, then $\mathcal{B} \leq_{\mathcal{B}} \mathcal{B}$ for definition of $\leq_{\mathcal{B}}$ (the bottom brick is less or equal than every other brick)
- Otherwise, we have that $S \subseteq S$, $min \geq min$ and $max \leq max$; then, for definition of $\leq_{\mathcal{B}}$, we also have that $\mathcal{B} \leq_{\mathcal{B}} \mathcal{B}$

We proved the reflexivity of $\leq_{\mathcal{B}}$.

8.3.2 Transitivity of $\leq_{\mathcal{B}}$

Given three bricks $\mathcal{B}_1$, $\mathcal{B}_2$ and $\mathcal{B}_3$, we must prove that:

$$\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_2 \wedge \mathcal{B}_2 \leq_{\mathcal{B}} \mathcal{B}_3 \Rightarrow \mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_3$$

Considering the relation $\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_2$, we have three cases:

- $\mathcal{B}_1 = \mathcal{B}^{\perp}$; then, by definition of $\leq_{\mathcal{B}}$, we know that $\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_3$, since $\mathcal{B}^{\perp}$ is smaller than any other element in the domain;
- $\mathcal{B}_2 = \mathcal{B}^{\top}$; then, by definition of $\leq_{\mathcal{B}}$ and since $\mathcal{B}_2 \leq_{\mathcal{B}} \mathcal{B}_3$ for hypothesis, $\mathcal{B}_3$ must be $\mathcal{B}^{\top}$ too. Then, we know that $\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_3$, since $\mathcal{B}^{\top}$ is greater than any other element in the domain;
- $\mathcal{B}_1 = [S_1]^{m_1,M_1}$ and $\mathcal{B}_2 = [S_2]^{m_2,M_2}$; here, considering the relation $\mathcal{B}_2 \leq_{\mathcal{B}} \mathcal{B}_3$, we have two cases:
    - $\mathcal{B}_3 = \mathcal{B}^{\top}$; then surely $\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_3$ since $\mathcal{B}^{\top}$ is greater than any other element in the domain;
    - $\mathcal{B}_3 = [S_3]^{m_3,M_3}$. Here we have:

- $S_1 \subseteq S_2$ and $S_2 \subseteq S_3$; then we also have $S_1 \subseteq S_3$
- $m_1 \geq m_2$ and $m_2 \geq m_3$; then we also have $m_1 \geq m_3$
- $M_1 \leq M_2$ and $M_2 \leq M_3$; then we also have $M_1 \leq M_3$
- Combining $S_1 \subseteq S_3$, $m_1 \geq m_3$, $M_1 \leq M_3$ we obtain that $\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_3$.

We proved the transitivity of $\leq_{\mathcal{B}}$.

8.3.3 Antisymmetry of $\leq_{\mathcal{B}}$

Given two bricks, $\mathcal{B}_1$ and $\mathcal{B}_2$, we must prove that:

$$\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_2 \wedge \mathcal{B}_2 \leq_{\mathcal{B}} \mathcal{B}_1 \Rightarrow \mathcal{B}_1 = \mathcal{B}_2$$

The cases in which one between $\mathcal{B}_1$ and $\mathcal{B}_2$ corresponds to $\mathcal{B}^{\top}$ or $\mathcal{B}^{\perp}$ are trivial. Less trivial is the case in which $\mathcal{B}_1 = [S_1]^{m_1,M_1}$ and $\mathcal{B}_2 = [S_2]^{m_2,M_2}$. From the hypothesis $\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_2 \wedge \mathcal{B}_2 \leq_{\mathcal{B}} \mathcal{B}_1$ and from the definition of $\leq_{\mathcal{B}}$ we deduce that:

- $S_1 \subseteq S_2$ but also $S_2 \subseteq S_1$; then we can say that $S_1 = S_2$;
- $m_1 \geq m_2$ but also $m_2 \geq m_1$; then $m_1 = m_2$;
- $M_1 \leq M_2$ but also $M_2 \leq M_1$; then $M_1 = M_2$;
- Thus $\mathcal{B}_1 = [S_1]^{m_1,M_1} = [S_2]^{m_2,M_2} = \mathcal{B}_2$.

We proved the antisymmetry of $\leq_{\mathcal{B}}$. Thus, $\leq_{\mathcal{B}}$ is a partial order. Now we are going to prove that $\leq_{\mathcal{A}_6}$ is a partial order too.

8.3.4 Reflexivity of $\leq_{\mathcal{A}_6}$

Given a list of bricks $\mathcal{L}$, we have three possible cases:

- $\mathcal{L} = \mathcal{L}^{\top}$ in which case, for definition of $\leq_{\mathcal{A}_6}$, $\mathcal{L} \leq_{\mathcal{A}_6} \mathcal{L}$ (every element of the domain is less or equal than the top element)
- $\mathcal{L} = \mathcal{L}^{\perp}$ in which case, for definition of $\leq_{\mathcal{A}_6}$, $\mathcal{L} \leq_{\mathcal{A}_6} \mathcal{L}$ (the bottom element is less or equal than every other element in the domain)

- $\mathcal{L} = [\mathcal{B}_1; \dots; \mathcal{B}_n] = [S_1]^{m_1,M_1} \dots [S_n]^{m_n,M_n}$; in this case, $\mathcal{L} \leq_{\mathcal{A}_6} \mathcal{L} \Leftrightarrow (\forall i \in [1,n] : \mathcal{L}[i] \leq_{\mathcal{B}} \mathcal{L}[i]) \Leftrightarrow \forall i \in [1,n] : [S_i]^{m_i,M_i} \leq_{\mathcal{B}} [S_i]^{m_i,M_i}$. Since we proved that $\leq_{\mathcal{B}}$ satisfies the *reflexivity* property, we know that $\forall i \in [1,n] : [S_i]^{m_i,M_i} \leq_{\mathcal{B}} [S_i]^{m_i,M_i}$ holds and this case is concluded.

We proved that $\mathcal{L} \leq_{\mathcal{A}_6} \mathcal{L}\ \ \forall \mathcal{L} \in \mathcal{A}_6$. The order $\leq_{\mathcal{A}_6}$ is then reflexive.

8.3.5 Transitivity of $\leq_{\mathcal{A}_6}$

Consider three lists of bricks $\mathcal{L}_a, \mathcal{L}_b, \mathcal{L}_c$, and suppose that $\mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_b$ and $\mathcal{L}_b \leq_{\mathcal{A}_6} \mathcal{L}_c$. Considering in particular $\mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_b$, we have three possible cases:

- $\mathcal{L}_b = \mathcal{L}^{\top}$; in this case $\mathcal{L}_c$ can only be $\mathcal{L}^{\top}$ too (since $\mathcal{L}_b \leq_{\mathcal{A}_6} \mathcal{L}_c$ and $\mathcal{L}_b = \mathcal{L}^{\top}$); then we also have, for definition of $\leq_{\mathcal{A}_6}$, that $\mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_c$ (every element of the domain is less or equal than top);
- $\mathcal{L}_a = \mathcal{L}^{\perp}$ in which case, no matter the shape of $\mathcal{L}_c$, we certainly have $\mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_c$ (bottom is less or equal than every other element);
- $\mathcal{L}_a = [\mathcal{B}_{1a}; \dots; \mathcal{B}_{na}] = [S_{1a}]^{m_{1a},M_{1a}} \dots [S_{na}]^{m_{na},M_{na}}$ and $\mathcal{L}_b = [\mathcal{B}_{1b}; \dots; \mathcal{B}_{nb}] = [S_{1b}]^{m_{1b},M_{1b}} \dots [S_{nb}]^{m_{nb},M_{nb}}$; in this case, since we know that $\mathcal{L}_b \leq_{\mathcal{A}_6} \mathcal{L}_c$, $\mathcal{L}_c$ could be:
  - $\mathcal{L}_c = \mathcal{L}^{\top}$, in which case we also have $\mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_c$ for definition of $\leq_{\mathcal{A}_6}$;
  - $\mathcal{L}_c = [\mathcal{B}_{1c}; \dots; \mathcal{B}_{nc}] = [S_{1c}]^{m_{1c},M_{1c}} \dots [S_{nc}]^{m_{nc},M_{nc}}$; in this case we know that $\forall i \in [1,n] : \mathcal{B}_{ia} \leq_{\mathcal{B}} \mathcal{B}_{ib}$ and $\forall i \in [1,n] : \mathcal{B}_{ib} \leq_{\mathcal{B}} \mathcal{B}_{ic}$; we have to prove that $\forall i \in [1,n] : \mathcal{B}_{ia} \leq_{\mathcal{B}} \mathcal{B}_{ic}$; in other words, we have to prove that $\forall i \in [1,n] : \mathcal{B}_{ia} \leq_{\mathcal{B}} \mathcal{B}_{ib} \wedge \mathcal{B}_{ib} \leq_{\mathcal{B}} \mathcal{B}_{ic} \Rightarrow \mathcal{B}_{ia} \leq_{\mathcal{B}} \mathcal{B}_{ic}$. We already proved this, since it is the property of transitivity of $\leq_{\mathcal{B}}$.

We proved that $\forall \mathcal{L}_a, \mathcal{L}_b, \mathcal{L}_c \in \mathcal{A}_6 : \mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_b \wedge \mathcal{L}_b \leq_{\mathcal{A}_6} \mathcal{L}_c \Rightarrow \mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_c$. The order $\leq_{\mathcal{A}_6}$ is then transitive.

Since we proved the reflexive and transitivity property, we can state that $\leq_{\mathcal{A}_6}$ is a **preorder**. In order for $\leq_{\mathcal{A}_6}$ to be a **partial order**, it has to satisfy another condition, antisymmetry.

8.3.6 Antisymmetry of $\leq_{\mathcal{A}_6}$

Consider two lists of bricks $\mathcal{L}_a, \mathcal{L}_b$, and suppose that $\mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_b$ and $\mathcal{L}_b \leq_{\mathcal{A}_6} \mathcal{L}_a$. Considering in particular $\mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_b$, we have three possible cases:

- $\mathcal{L}_b = \mathcal{L}^\top$; in this case, since the top element is bigger than every other element of the domain (except itself) and for hypothesis $\mathcal{L}_b \leq_{\mathcal{A}_6} \mathcal{L}_a$, we have to conclude that $\mathcal{L}_a = \mathcal{L}^\top$ too. Then $\mathcal{L}_b = \mathcal{L}_a = \mathcal{L}^\top$.
- $\mathcal{L}_a = \mathcal{L}^\perp$; in this case, since the bottom element is smaller than every other element of the domain (except itself) and for hypothesis $\mathcal{L}_b \leq_{\mathcal{A}_6} \mathcal{L}_a$, we have to conclude that $\mathcal{L}_b = \mathcal{L}^\perp$ too. Then $\mathcal{L}_b = \mathcal{L}_a = \mathcal{L}^\perp$.
- $\mathcal{L}_a = [\mathcal{B}_{1a}; \dots; \mathcal{B}_{na}] = [S_{1a}]^{m_{1a},M_{1a}} \dots [S_{na}]^{m_{na},M_{na}}$ and $\mathcal{L}_b = [\mathcal{B}_{1b}; \dots; \mathcal{B}_{nb}] = [S_{1b}]^{m_{1b},M_{1b}} \dots [S_{nb}]^{m_{nb},M_{nb}}$; in this case, for hypothesis and for definition of $\leq_{\mathcal{A}_6}$, we have $\forall i \in [1,n] : \mathcal{B}_{ia} \leq_{\mathcal{B}} \mathcal{B}_{ib} \wedge \mathcal{B}_{ib} \leq_{\mathcal{B}} \mathcal{B}_{ia}$. Then, we know that $\forall i \in [1,n] : \mathcal{B}_{ia} = \mathcal{B}_{ib}$ (for the antisymmetry property of $\leq_{\mathcal{B}}$). Thus, $\mathcal{L}_a = [\mathcal{B}_{1a}; \dots; \mathcal{B}_{na}] = [\mathcal{B}_{1b}; \dots; \mathcal{B}_{nb}] = \mathcal{L}_b$.

We proved that $\mathcal{L}_a \leq_{\mathcal{A}_6} \mathcal{L}_b \wedge \mathcal{L}_b \leq_{\mathcal{A}_6} \mathcal{L}_a \Rightarrow \mathcal{L}_a = \mathcal{L}_b$. The order $\leq_{\mathcal{A}_6}$ is then antisymmetric. Thus we can conclude that $\leq_{\mathcal{A}_6}$ is indeed a partial order.

Now, we want to know if the concretization function is monotone with respect to our order. The monotonicity of the concretization function is a fundamental property and can be formalized as follows:

$$\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2 \Rightarrow \gamma(\mathcal{L}_1) \subseteq \gamma(\mathcal{L}_2)$$

Another interesting property that our order could satisfy is completeness. The **completeness** property is formalized as follows:

$$\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2 \Leftarrow \gamma(\mathcal{L}_1) \subseteq \gamma(\mathcal{L}_2)$$

Thus, if our partial order were complete and the concretization function were monotone we could write:

$$\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2 \Leftrightarrow \gamma(\mathcal{L}_1) \subseteq \gamma(\mathcal{L}_2)$$

In order to verify those two properties, we must formalize the **concretization function**, γ. Remember that such function maps an element $a \in \mathcal{A}_6$ to an element $d \in \mathcal{D}$. Since our concrete domain $\mathcal{D}$ is made of string sets, an abstract value maps to a set of possible strings. Our abstract values are lists of bricks. Each brick represents a certain set of strings. The list of bricks thus represents all the string built through the concatenation of strings which can be made from the bricks of the list (taken in the correct order). More formally, we define the strings represented by a *single* brick as:

$$\gamma(\mathcal{B}) = \gamma\left([S]^{min,max}\right) = \left(\bigcup_{j=min}^{max} \underbrace{(SS \ldots S)}_{j \text{ times}}\right)$$

where $\underbrace{(SS \ldots S)}_{j \text{ times}} = S^j$ is the concatenation between set of strings (defined in Chapter 2, in the paragraph about regular expressions). To account for the case in which $j = 0$, we impose $S^0 = \{\epsilon\}$.

Let us make an example to be clearer about this definition. Consider the brick $[\{'a','b'\}]^{1,3}$. Then, $S = \{'a','b'\}$, $min = 1$ and $max = 3$. In the following table we consider all the possible values of j and the set of string corresponding to such value:

j	**Concatenation**	**Resulting strings**
1	S	$S = \{a, b\}$
2	SS	$SS = \{a, b\}\{a, b\} = \{aa, ab, ba, bb\}$

3	SSS	$SSS = S(SS) = \{a,b\}(SS)$ $= \{a,b\}\{aa,ab,ba,bb\}$ $= \{aaa,aab,aba,abb,baa,bab,bba,bbb\}$

On the whole:

$$\begin{aligned}\gamma([\{'a','b'\}]^{1,3}) &= S \cup SS \cup SSS \\ &= \{a,b\} \cup \{aa,ab,ba,bb\} \\ &\cup \{aaa,aab,aba,abb,baa,bab,bba,bbb\} \\ &= \{a,b,aa,ab,ba,bb,aaa,aab,aba,abb,baa,bab,bba,bbb\}\end{aligned}$$

Note that, if min had been 0 instead of 1, the result would have been:

$$\gamma([\{'a','b'\}]^{0,3}) = S^0 \cup S^1 \cup S^2 \cup S^3 = \{\epsilon\} \cup S \cup SS \cup SSS$$

Some special cases:

- $\gamma([\emptyset]^{0,0}) = \gamma(\mathrm{E}) = \{\epsilon\}$
- $\gamma\left([\emptyset]^{(m,M)\neq(0,0)}\right) = \gamma(\mathcal{B}^{\perp}) = \emptyset$
- $\gamma([S \neq \emptyset]^{0,0}) = \gamma(\mathcal{B}^{\perp}) = \emptyset$
- $\gamma\left(\left([S]^{min,max}, max < min\right)\right) = \gamma(\mathcal{B}^{\perp}) = \emptyset$
- $\gamma([K]^{0,+\infty}) = \gamma(\mathcal{B}^{\top}) = \bigcup_{j=0}^{\infty} K^j = K^*$ (the top brick represents all possible strings).

Our concretization function is finally the following one:

$$\boxed{\gamma(\mathcal{L}) = \gamma(\mathcal{B}_1\mathcal{B}_2 \ldots \mathcal{B}_n) = \{\, s : s \text{ is a string} \wedge s = b_1 + b_2 + \cdots + b_n \wedge \forall i \in [1,n] : b_i \in \gamma(\mathcal{B}_i) \,\}}$$

where "+" represents the operator of string concatenation.

Now that we have formally defined our concretization function γ, we can try and verify the two interesting properties cited above with regards to our order $\leq_{\mathcal{A}_6}$.

8.3.7 Monotonicity of the concretization function

The concretization function is monotone if the following condition holds for each $\mathcal{L}_1, \mathcal{L}_2 \in \mathcal{A}_6$:

$$\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2 \Rightarrow \gamma(\mathcal{L}_1) \subseteq \gamma(\mathcal{L}_2)$$

In order to prove it, we will need to prove some lemmas first. Then, with the help of such lemmas, we will be able to conclude that our concretization function γ is monotone with respect to $\leq_{\mathcal{A}_6}$.

8.3.7.1 LEMMA 1

Lemma 1: Consider $S_1, \dots, S_n, T_1, \dots, T_n$ strings sets such that $\forall i \in [1, n] : S_i \subseteq T_i$. Let $S_1 \dots S_n$ be the set of the strings s such that $s = s_1 + \dots + s_n$ where $\forall i \in [1, n] : s_i \in S_i$ and $+$ is the string concatenation. Then:

$$S_1 \dots S_n \subseteq T_1 \dots T_n$$

Proof: A set is subset of another set if each element of the first set is an element of the second set too:

$$S \subseteq T \Leftrightarrow \forall s \in S : s \in T$$

Thus, we have to prove that each element of $S_1 \dots S_n$ is an element of $T_1 \dots T_n$.

An element of $S_1 \dots S_n$ is a string s such that $s = s_1 + \dots + s_n$ and $\forall i \in [1, n] : s_i \in S_i$. Since $\forall i \in [1, n] : S_i \subseteq T_i$, we can also write s as $s_1 + \dots + s_n$ where $\forall i \in [1, n] : s_i \in T_i$. But this is exactly the definition of the elements of $T_1 \dots T_n$. Then, $s \in T_1 \dots T_n$. We proved that each element of $S_1 \dots S_n$ is also an element of $T_1 \dots T_n$, thus we proved that $S_1 \dots S_n \subseteq T_1 \dots T_n$. This concludes our proof. □

8.3.7.2 LEMMA 2

Lemma 2: Consider two strings sets, S and T. If $S \subseteq T$ then $S^n \subseteq T^n$ (where $S^n = \underbrace{(SS \dots S)}_{n \text{ times}}$, that is the concatenation between set of strings).

Proof: This is just a special case of **Lemma 1**, in which $\forall i \in [1, n] : S_i = S$ and $\forall i \in [1, n] : T_i = T$. □

8.3.7.3 LEMMA 3

Lemma 3: Let $A_1, \dots, A_n, B_1, \dots, B_n$ be sets such that $\forall i \in [1, n] : A_i \subseteq B_i$. Then:

$$A_1 \cup \dots \cup A_n \subseteq B_1 \cup \dots \cup B_n$$

Proof: As in the proof of Lemma 1, we have to prove that each element of $A_1 \cup \dots \cup A_n$ is also an element of $B_1 \cup \dots \cup B_n$. This is quite trivial.

If $a \in (A_1 \cup \dots \cup A_n)$ then $a \in A_i$ for some $i \in [1, n]$. Then, since $A_i \subseteq B_i$ for hypothesis, $a \in B_i$ too. Then, $a \in (B_1 \cup \dots \cup B_n)$. This concludes our proof. □

8.3.7.4 LEMMA 4 (Monotonicity of $\gamma_{\mathcal{B}}$)

Lemma 4: Given two blocks $\mathcal{B}_1$ and $\mathcal{B}_2$:

$$\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_2 \Rightarrow \gamma(\mathcal{B}_1) \subseteq \gamma(\mathcal{B}_2)$$

Proof: If $\mathcal{B}_1 \leq_{\mathcal{B}} \mathcal{B}_2$ then, for definition of $\leq_{\mathcal{B}}$, one of the following cases must hold:

- $\mathcal{B}_1 = \mathcal{B}^{\perp}$
    - In this case, $\gamma(\mathcal{B}_1) = \emptyset$ and $\emptyset \subseteq S \forall S$, so $\emptyset \subseteq \gamma(\mathcal{B}_2)$. Then $\gamma(\mathcal{B}_1) \subseteq \gamma(\mathcal{B}_2)$ and we trivially conclude.
- $\mathcal{B}_2 = \mathcal{B}^{\top}$

    - In this case, $\gamma(\mathcal{B}_2) = K^*$ and $S \subseteq K^*$ $\forall$ string set S, so $\gamma(\mathcal{B}_1) \subseteq K^*$. Then $\gamma(\mathcal{B}_1) \subseteq \gamma(\mathcal{B}_2)$ and we trivially conclude.
- $\mathcal{B}_1 = [C_1]^{min_1,max_1}$, $\mathcal{B}_2 = [C_2]^{min_2,max_2}$ and $C_1 \subseteq C_2, min_1 \geq min_2, max_1 \leq max_2$
    - This case is the last one and the most complex. For the definition of γ , we can write: $\gamma(\mathcal{B}_1) = C_1^{min_1} \cup C_1^{min_1+1} \cup \ldots \cup C_1^{max_1}$ and $\gamma(\mathcal{B}_2) = C_2^{min_2} \cup C_2^{min_2+1} \cup \ldots \cup C_2^{max_2}$. Since $min_1 \geq min_2, max_1 \leq max_2$ we can re-write $\gamma(\mathcal{B}_2)$ as: $\gamma(\mathcal{B}_2) = C_2^{min_2} \cup \ldots \cup C_2^{min_1} \cup \ldots \cup C_2^{max_1} \cup \ldots \cup C_2^{max_2}$. Since we also know that $C_1 \subseteq C_2$, for **Lemma 2** we have that $C_1^i \subseteq C_2^i$ $\forall i \in [min_1, max_1]$. Then, for **Lemma 3** we have that $C_1^{min_1} \cup \ldots \cup C_1^{max_1} \subseteq C_2^{min_1} \cup \ldots \cup C_2^{max_1}$. For definition of set union, $C_2^{min_1} \cup \ldots \cup C_2^{max_1} \subseteq C_2^{min_2} \cup \ldots \cup C_2^{min_1} \cup \ldots \cup C_2^{max_1} \cup \ldots \cup C_2^{max_2}$. Thus, $\gamma(\mathcal{B}_1) = C_1^{min_1} \cup \ldots \cup C_1^{max_1} \subseteq C_2^{min_1} \cup \ldots \cup C_2^{max_1} \subseteq C_2^{min_2} \cup \ldots \cup C_2^{min_1} \cup \ldots \cup C_2^{max_1} \cup \ldots \cup C_2^{max_2} = \gamma(\mathcal{B}_2)$. Then $\gamma(\mathcal{B}_1) \subseteq \gamma(\mathcal{B}_2)$ and we conclude this case.

In each case above we proved that $\gamma(\mathcal{B}_1) \subseteq \gamma(\mathcal{B}_2)$, therefore we concluded our proof. □

8.3.7.5 THEOREM (Monotonicity of $\gamma_{\mathcal{A}_6}$)

Theorem: The concretization function γ is monotone with respect to $\leq_{\mathcal{A}_6}$, that is, given $\mathcal{L}_1 = [\mathcal{B}_{11}; \ldots; \mathcal{B}_{1n}]$ and $\mathcal{L}_2 = [\mathcal{B}_{21}; \ldots; \mathcal{B}_{2n}]$

$$\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2 \Rightarrow \gamma(\mathcal{L}_1) \subseteq \gamma(\mathcal{L}_2)$$

Note that we can safely assume that the two lists have the same size (n). In fact, when they have different sizes, we can add the appropriate number of empty bricks E at the end of the shorter list.

Proof: For definition of $\leq_{\mathcal{A}_6}$, $\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2 \Rightarrow \mathcal{B}_{1i} \leq_{\mathcal{B}} \mathcal{B}_{2i}\ \forall i \in [1, n]$. Moreover, for **Lemma 4** (correctness of $\leq_{\mathcal{B}}$), $\mathcal{B}_{1i} \leq_{\mathcal{B}} \mathcal{B}_{2i} \Rightarrow \gamma(\mathcal{B}_{1i}) \subseteq \gamma(\mathcal{B}_{2i})$.

For definition of γ we also have:

$$\gamma(\mathcal{L}_1) = \gamma([\mathcal{B}_{11}; \mathcal{B}_{12}; \dots; \mathcal{B}_{1n}]) = \{\, s : s \text{ is a string} \wedge s = b_1 + b_2 + \dots + b_n \wedge b_i \in \gamma(\mathcal{B}_{1i})\ \forall i \in [1, n] \,\}$$

which, using another notation (concatenation between set of strings), can also be written like:

$$\gamma(\mathcal{L}_1) = \gamma([\mathcal{B}_{11}; \mathcal{B}_{12}; \dots; \mathcal{B}_{1n}]) = \gamma(\mathcal{B}_{11})\gamma(\mathcal{B}_{12}) \dots \gamma(\mathcal{B}_{1n})$$

$$\gamma(\mathcal{L}_2) = \gamma([\mathcal{B}_{21}; \mathcal{B}_{22}; \dots; \mathcal{B}_{2n}]) = \gamma(\mathcal{B}_{21})\gamma(\mathcal{B}_{22}) \dots \gamma(\mathcal{B}_{2n})$$

Since $\gamma(\mathcal{B}_{1i}) \subseteq \gamma(\mathcal{B}_{2i})\ \forall i \in [1, n]$, we can use **Lemma 1** to assert that $\gamma(\mathcal{B}_{11})\gamma(\mathcal{B}_{12}) \dots \gamma(\mathcal{B}_{1n}) \subseteq \gamma(\mathcal{B}_{21})\gamma(\mathcal{B}_{22}) \dots \gamma(\mathcal{B}_{2n})$. Then, $\gamma(\mathcal{L}_1) \subseteq \gamma(\mathcal{L}_2)$ and this concludes our proof. □

8.3.8 Completeness property

The completeness property is the following:

$$\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2 \Leftarrow \gamma(\mathcal{L}_1) \subseteq \gamma(\mathcal{L}_2)$$

This means that, if the concretization of an abstract value ($\mathcal{L}_1$) is a subset of the concretization of another abstract value ($\mathcal{L}_2$), then $\mathcal{L}_1$ must be smaller, in our order, than $\mathcal{L}_2$.

Unfortunately, our order does not satisfy this property. To prove this, we only need to show a counterexample. Consider the following abstract values (both normalized):

$$\mathcal{L}_1 = [abb]^{1,1}$$

$$\mathcal{L}_2 = [a]^{1,1}[\{b, c\}]^{0,2}$$

The first abstract value is a list with only one element. Its concretization is the following:

$$\gamma(\mathcal{L}_1) = \{abb\}$$

The second abstract value is a list with two elements. Given that $\gamma([a]^{1,1}) = \{a\}$ and $\gamma([\{b, c\}]^{0,2}) = \{b, c\}^0 \cup \{b, c\}^1 \cup \{b, c\}^2 = \{\epsilon\} \cup \{b, c\} \cup \{bb, bc, cb, cc\} = \{\epsilon, b, c, bb, bc, cb, cc\}$, the concretization of $\mathcal{L}_2$ is the following:

$$\gamma(\mathcal{L}_2) = \{a, ab, ac, abb, abc, acb, acc\}$$

It can be easily seen that $\gamma(\mathcal{L}_1) \subseteq \gamma(\mathcal{L}_2)$: $\{abb\} \subseteq \{a, ab, ac, abb, abc, acb, acc\}$. However, it is not true that $\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2$. In fact, to compare the two list, we stuff the first one with an empty brick: $\mathcal{L}_1{}' = \mathcal{L}_1 + \mathrm{E}$. Then we proceed comparing pairs of blocks taken from the two list: $[abb]^{1,1} \nleq_{\mathcal{B}} [a]^{1,1}$, $\mathrm{E} \leq_{\mathcal{B}} [\{b, c\}]^{0,2}$. Since not all the blocks from the first list are smaller than the corresponding block from the second list, we cannot say that $\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2$:

$$\mathcal{L}_1 \nleq_{\mathcal{A}_6} \mathcal{L}_2$$

We have thus proved that our concretization function is monotone with respect to $\leq_{\mathcal{A}_6}$, but such order $\leq_{\mathcal{A}_6}$ does <u>not</u> satisfy the completeness property.

Now we can go on describing the abstract domain $\mathcal{A}_6$: top element, bottom element, least upper bound operator and so on.

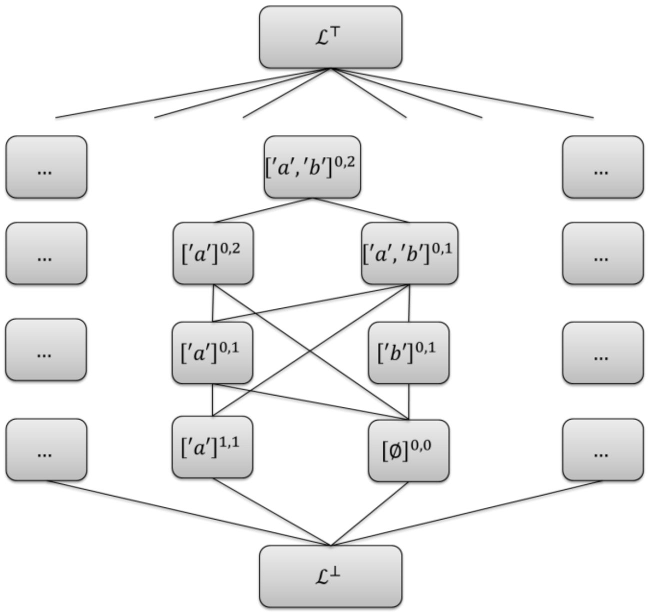
The **bottom** element of this domain is $\bot = \mathcal{L}^{\bot}$. For definition of $\leq_{\mathcal{A}_6}$:

$$\boxed{\mathcal{L}^{\bot} \leq_{\mathcal{A}_6} \mathcal{L}\ \forall \mathcal{L} \in \mathcal{A}_6}$$

The **top** element is $\top = \mathcal{L}^{\top}$. For definition of $\leq_{\mathcal{A}_6}$:

$$\boxed{\mathcal{L} \leq_{\mathcal{A}_6} \mathcal{L}^{\top}\ \forall \mathcal{L} \in \mathcal{A}_6}$$

The lattice of $\mathcal{A}_6$ is the following (for visual clarity, we only pictured lists of size one and we considered $K = \{'a', 'b'\}$):



We want to define a least upper bound operator on our lattice. As we did for the partial order, we are going to work with single bricks at first, and then we will extend our results to lists of bricks. The **least upper bound** operator on single bricks is the following:

$$\boxed{\bigsqcup(\mathcal{B}_1, \mathcal{B}_2) = \bigsqcup([S_1]^{m_1,M_1}, [S_2]^{m_2,M_2}) = [S_1 \cup S_2]^{m,M}}$$

where $m = \min(m_1, m_2)$ and $M = \max(M_1, M_2)$.

Let us see some example of how this operator works:

- Consider $['a', 'b']^{1,2} = \{a, b, aa, bb, ab, ba\}$ and $['a', 'c']^{0,2} = \{\epsilon, a, c, aa, ac, ca, cc\}$; the least upper bound between them is

$\coprod(['a','b']^{1,2},['a','c']^{0,2}) = ['a','b','c']^{0,2} = \{\epsilon, a, b, c, aa, ab, ac, bb, ba, bc, cc, ca, cb\}$;

- Consider $['a']^{3,4} = \{aaa, aaaa\}$ and $[\emptyset]^{0,0} = \{\ \epsilon\ \}$; their least upper bound is $\coprod(['a']^{3,4},[\emptyset]^{0,0}) = ['a']^{0,4} = \{\ \epsilon, a, aa, aaa, aaaa\ \}$;

In both examples we saw that the least upper bound of two bricks generally contains *more* strings than the simple union of the original strings represented by the single bricks. This approximation is caused by the combination of the structures of the domain $\mathcal{A}_6$ and the lub operator $\sqcup$.

To compute the least upper bound between elements of our domain (lists of bricks) we proceed exactly as we did to define the partial order $\leq_{\mathcal{A}_6}$. Given two lists $\mathcal{L}_1$ and $\mathcal{L}_2$, we make them have the same size n adding empty bricks (E) to the shorter one, thus obtaining $\mathcal{L}_1'$ and $\mathcal{L}_2'$. Then:

$$\boxed{\coprod(\mathcal{L}_1,\mathcal{L}_2) = \coprod(\mathcal{L}_1',\mathcal{L}_2') = \mathcal{L}_R[1]\mathcal{L}_R[2]\dots\mathcal{L}_R[n]}$$

where $\mathcal{L}_R[i] = \coprod(\mathcal{L}_1'[i],\mathcal{L}_2'[i])\ \ \forall i \in [1,n]$.

The **greatest lower bound** operator works very similarly to the least upper bound one. The GLB on single bricks is defined as follows:

$$\boxed{\prod(\mathcal{B}_1,\mathcal{B}_2) = \prod([S_1]^{m_1,M_1},[S_2]^{m_2,M_2}) = [S_1 \cap S_2]^{m,M}}$$

where $m = \max(m_1, m_2)$ and $M = \min(M_1, M_2)$.

Let us see some examples:

- Consider $['a','b']^{1,2} = \{a, b, aa, bb, ab, ba\}$ and $['a','c']^{0,2} = \{\epsilon, a, c, aa, ac, ca, cc\}$; the greatest lower bound between them is $\prod(['a','b']^{1,2},['a','c']^{0,2}) = ['a']^{1,2} = \{a, aa\}$;

- Consider $['b']^{0,2} = \{\epsilon, b, bb\}$ and $['a','b']^{2,3} = \{aa, bb, ab, ba, aaa, aab, aba, abb, baa, bab, bba, bbb\}$; the greatest lower bound between them is $\prod(['a','b']^{2,3}, ['b']^{0,2}) = ['b']^{2,2} = \{bb\}$;
- Consider $['a','b']^{0,2} = \{\epsilon, a, b, aa, bb, ab, ba\}$ and $[\emptyset]^{0,0} = \{\epsilon\}$; the greatest lower bound between them is $\prod(['a','b']^{0,2}, [\emptyset]^{0,0}) = [\emptyset]^{0,0} = \{\epsilon\}$

Other examples are the following:

- Consider $['a','b']^{1,2} = \{a, b, aa, bb, ab, ba\}$ and $['c']^{0,2} = \{\epsilon, c, cc\}$; the greatest lower bound between them is $\prod(['a','b']^{1,2}, ['c']^{0,2}) = [\emptyset]^{1,2} = \mathcal{B}^{\perp}$ (since $(1,2) \neq (0,0)$);
- Consider $['a','b']^{1,2} = \{a, b, aa, bb, ab, ba\}$ and $[\emptyset]^{0,0} = \{\epsilon\}$; the greatest lower bound between them is $\prod(['a','b']^{1,2}, [\emptyset]^{0,0}) = [\emptyset]^{1,0} = \mathcal{B}^{\perp}$ (since $max < min$);
- Consider $['a']^{1,2} = \{a, aa\}$ and $['a']^{3,4} = \{aaa, aaaa\}$; the greatest lower bound between them is $\prod(['a']^{1,2}, ['a']^{3,4}) = ['a']^{3,2} = \mathcal{B}^{\perp}$ (since $max < min$);

In these last three examples, the result was always an invalid brick, for example because the max index was less than the min one. In these cases, the result was $\mathcal{B}^{\perp}$. The outcome is correct, since in each case the two bricks had nothing (i.e. no represented strings) in common.

To conclude the description of the GLB operator, we have to define how it works with lists of bricks (the elements of our domain). Given two lists $\mathcal{L}_1$ and $\mathcal{L}_2$, we make them have the same size n adding empty bricks (E) to the shorter one, thus obtaining $\mathcal{L}_1'$ and $\mathcal{L}_2'$. Then:

$$\boxed{\prod(\mathcal{L}_1, \mathcal{L}_2) = \prod(\mathcal{L}_1', \mathcal{L}_2') = \mathcal{L}_R[1]\mathcal{L}_R[2] \dots \mathcal{L}_R[n]}$$

where $\mathcal{L}_R[i] = \prod(\mathcal{L}'_1[i], \mathcal{L}'_2[i])\ \forall i \in [1, n]$. If any element of $\mathcal{L}_R$ corresponds to $\mathcal{B}^{\perp}$, then the entire resulting list should be set to $\mathcal{L}^{\perp}$.

Now we can proceed to define the abstraction function, always remembering that our concrete domain is made of *set of strings*. A single string can have multiple abstractions. In the example of $'look'$ we saw various possible representations: $[l]^{1,1}[o]^{1,1}[ok]^{1,1}$, $[lo]^{1,1}[o]^{1,1}[k]^{1,1}$, $[l]^{1,1}[o]^{1,1}[o]^{1,1}[k]^{1,1}$, etc. We decided that the normalized representation was $[look]^{1,1}$, which is also the briefest and simplest one. Thus, the function which abstracts a single string is the following:

$$\alpha'_6(s) = [s]^{1,1}$$

The **abstraction function** has to abstract a set of concrete strings S (that is, an element of $\mathcal{D}$) into an element of $\mathcal{A}_6$. The solution to this problem is simple: we can abstract all the strings in the set and then compute the least upper bound of them in $\mathcal{A}_6$:

$$\boxed{\alpha_6(S) = \coprod_{s \in S} \alpha'_6(s) = [S]^{1,1}}$$

The abstracted value $\alpha_6(S)$ is really simple: a list composed by only a single brick, containing the whole set of strings S and having indices $min = 1$ and $max = 1$ (each string can be used only one time). Note that $\alpha_6(S)$ already is in a normalized form.

The **widening operator** is the following:

$$\nabla : (\mathcal{A}_6, \mathcal{A}_6) \to \mathcal{A}_6$$

$$\nabla(\mathcal{L}_1, \mathcal{L}_2) = \begin{cases} \top \text{ if } \left(\mathcal{L}_1 \not\leq_{\mathcal{A}_6} \mathcal{L}_2 \wedge \mathcal{L}_2 \not\leq_{\mathcal{A}_6} \mathcal{L}_1\right) \vee (\exists i \in [1,2] : len(\mathcal{L}_i) > k_L) \\ \mathrm{w}(\mathcal{L}_1, \mathcal{L}_2) \text{ otherwise} \end{cases}$$

We return the $\top$ element of our domain in two cases:

1. if the two abstract values are not comparable with respect to our order ($\mathcal{L}_1 \nleq \mathcal{L}_2 \wedge \mathcal{L}_2 \nleq \mathcal{L}_1$);
2. if the length of the list associated to one abstract value is greater than a certain constant k_L ($\exists i \in [1,2] : len(\mathcal{L}_i) > k_L$). The constant k_L can be considered a parameter of the domain.

Otherwise, the two lists are comparable and they have a bounded length. In such case we return $\mathrm{w}(\mathcal{L}_1, \mathcal{L}_2)$. Now we have to define what the function $\mathrm{w}(\cdot,\cdot)$ does. Let us assume that $\mathcal{L}_1 \leq \mathcal{L}_2$ ($\mathcal{L}_1$ is the smaller value) and that $len(\mathcal{L}_1) = len(\mathcal{L}_2) = n$. If the two lists were not of the same length, we could always add a proper number of empty blocks (E) at the end of the shorter list. The definition of $\mathrm{w}(\cdot,\cdot)$ is thus the following:

$$\mathrm{w}(\mathcal{L}_1, \mathcal{L}_2) = \mathcal{L}_{new} = [\mathcal{B}_1^{new}; \mathcal{B}_2^{new}; \dots; \mathcal{B}_n^{new}]$$

where:

$$\mathcal{B}_i^{new} = \Omega(\mathcal{L}_1[i], \mathcal{L}_2[i])$$

This means that the new abstract value is a list ($\mathcal{L}_{new}$) in which each element ($\mathcal{B}_i^{new}$) is the combination of the two corresponding elements of the two initial lists ($\mathcal{L}_1[i]$ and $\mathcal{L}_2[i]$). How is this combination formed?

$$\Omega(\mathcal{B}_1, \mathcal{B}_2) = \Omega([\mathrm{S}_1]^{\mathrm{m}_1,\mathrm{M}_1}, [S_2]^{m_2,M_2}) = \begin{cases} \mathcal{B}^\top \text{ if } |S_1 \cup S_2| > k_S \vee \mathcal{B}_1 = \mathcal{B}^\top \vee \mathcal{B}_2 = \mathcal{B}^\top \\ [S_1 \cup S_2]^{0,+\infty} \text{ if } (M - m) > k_I \\ [S_1 \cup S_2]^{m,M} \text{ otherwise} \end{cases}$$

where $m = \min(m_1, m_2)$ and $M = \max(M_1, M_2)$. We find other two constants, k_S and k_I, which can be considered parameters of the domain together with k_L.

Let us briefly explain why this widening operator is correct.

First of all, the result of a widening between two values must be greater or equal than both values:

$$\Omega(\overline{s_1}, \overline{s_2}) = \overline{s} \Rightarrow \overline{s_1} \leq \overline{s} \wedge \overline{s_2} \leq \overline{s}$$

In our domain, the result of the widening between $\mathcal{L}_1$ and $\mathcal{L}_2$ can be $\top$ or $\mathrm{w}(\mathcal{L}_1, \mathcal{L}_2)$. If it is $\top$, the conclusion is immediate ($\mathcal{L}_1 \leq_{\mathcal{A}_6} \top$ and $\mathcal{L}_2 \leq_{\mathcal{A}_6} \top$). In the other case, we know that $\mathcal{L}_1 \leq_{\mathcal{A}_6} \mathcal{L}_2$ or vice versa (for argument's sake, we assume that $\mathcal{L}_1$ is the smaller value). Thus, the result of the widening is a new list in which each element $\mathcal{B}_i^{new}$ is the combination of $\mathcal{L}_1[i]$ and $\mathcal{L}_2[i]$. Given the formulation of our order $\leq_{\mathcal{A}_6}$, to prove that $\mathcal{L}_1 \leq_{\mathcal{A}_6} \Omega(\mathcal{L}_1, \mathcal{L}_2)$ and $\mathcal{L}_2 \leq_{\mathcal{A}_6} \Omega(\mathcal{L}_1, \mathcal{L}_2)$, we just need to prove that $\mathcal{L}_1[i] \leq_{\mathcal{B}} \mathcal{B}_i^{new}$ and $\mathcal{L}_2[i] \leq_{\mathcal{B}} \mathcal{B}_i^{new}$ $\forall i \in [1, len(\mathcal{L}_1)]$. The combination of $\mathcal{L}_1[i]$ and $\mathcal{L}_2[i]$ is one of the following three possibilities:

1. $\mathcal{B}^{\top}$; in this case it is immediate to show that $\mathcal{L}_1[i] \leq_{\mathcal{B}} \mathcal{B}^{\top}$ and $\mathcal{L}_2[i] \leq_{\mathcal{B}} \mathcal{B}^{\top}$
2. $[S_1 \cup S_2]^{0,+\infty}$; in this case $\mathcal{L}_1[i] = [\mathrm{S}_1]^{\mathrm{m}_1,\mathrm{M}_1} \leq_{\mathcal{B}} [S_1 \cup S_2]^{0,+\infty}$ since $\mathrm{S}_1 \subseteq S_1 \cup S_2$, $\mathrm{m}_1 \geq 0$ and $\mathrm{M}_1 \leq +\infty$. The same happens for $\mathcal{L}_2[i]$: $[S_2]^{m_2,M_2} \leq_{\mathcal{B}} [S_1 \cup S_2]^{0,+\infty}$ since $\mathrm{S}_2 \subseteq S_1 \cup S_2$, $\mathrm{m}_2 \geq 0$ and $\mathrm{M}_2 \leq +\infty$.
3. $[S_1 \cup S_2]^{m,M}$ where $m = \min(m_1, m_2)$ and $M = \max(M_1, M_2)$. In this case we have $\mathcal{L}_1[i] = [\mathrm{S}_1]^{\mathrm{m}_1,\mathrm{M}_1} \leq_{\mathcal{B}} [S_1 \cup S_2]^{m,M}$ since $\mathrm{S}_1 \subseteq S_1 \cup S_2$, $\mathrm{m}_1 \geq m = \min(m_1, m_2)$ and $\mathrm{M}_1 \leq M = \max(M_1, M_2)$. The same happens for $\mathcal{L}_2[i]$.

Secondly, the widening operator must be convergent. In other words, given an ascending chain $\overline{s_n}$, the sequence $\overline{t_{n+1}} = \Omega(\overline{t_n}, \overline{s_n})$ is ultimately stationary. In our case, a value of an ascending chain can increase along three axes:

1. The length of the brick list
2. The indices range of a certain block
3. The strings contained in a certain block

The growth of an abstract value is bounded along each axis with the help of the three constants, k_L, k_S and k_I. After the list has reached k_L elements, the entire abstract value is approximated to $\top$, stopping its possible growth altogether. If the range of a certain block becomes larger than k_I, the block is approximated to $[S]^{0,+\infty}$, stopping the indices possible growth. If the strings set of a certain block reaches k_S elements, the block is approximated to $\mathcal{B}^{\top}$, stopping its possible growth altogether.

Now that we have finished presenting and explaining the widening operator, we are going to see the **semantics** of the string operators, with the usual notation.

8.3.9 String(char[] data)

This is the string constructor. As we already said, given a concrete string s, its normal form is $[s]^{1,1}$. Consider for example the following code:

```
char[] data = {'a', 'b', 'c'};
String s = new String(data);
```

The semantics is:

$$S\big[[new\ String, \mathrm{arrayChars}]\big] = [\mathrm{arrayChars}]^{1,1}$$

In our example above, the resulting abstract value is $[\mathrm{abc}]^{1,1}$.

Obviously if we don’t know what `data` contains (think about user input), then the semantics will be:

$$S\big[[\mathrm{new\ String, arrayChars}]\big] = \mathcal{L}^{\top} = [K]^{0,+\infty}$$

8.3.10 concat(String str)

This operator takes two strings and concatenates them. For example, if `s1 = “hello ”` and `s2 = “world”` then `s1.concat(s2) = “hello world”`. Suppose that $\overline{s1}$ is the abstract representation of `s1` and that $\overline{s2}$ is the abstract representation of `s2`. The semantics of this operator is:

$$S\left[\left[\text{concat}, \overline{s1}, \overline{s2}\right]\right] = normalize\left(concatList\left(\overline{s1}, \overline{s2}\right)\right)$$

where $concatList$ represents the concatenation between lists (defined in paragraph 2.2). Even supposing that $\overline{s1}$ and $\overline{s2}$ are in normalized form, their concatenation could not be in normalized form. For this reason, after having concatenated the two lists, we proceed to normalize the result. In the example cited above, we have:

$$\overline{s1} = [\text{"}hello\text{ "}]^{1,1}$$

$$\overline{s2} = [\text{"}world\text{"}]^{1,1}$$

The abstract value resulting from concatenation of `s1` and `s2` is:

$$\overline{s_{res}} = norm\left(\overline{s1} + \overline{s2}\right) = normalize([\text{"}hello\text{ "}]^{1,1}[\text{"}world\text{"}]^{1,1})$$

Applying rule $\boldsymbol{\mathcal{R}_2}$, we can merge the two blocks into one:

$$normalize([\text{"}hello\text{ "}]^{1,1}[\text{"}world\text{"}]^{1,1}) = [\text{"}hello\ world\text{"}]^{1,1}$$

We can see that computing the result of the `concat` operation we did not lose any information. The result of the operation is the best we could have hoped.

8.3.11 indexOf(int c)

This operator returns the index within the string of the first occurrence of the specified character. In order to define the semantics of such operator, we must remember the considerations we made at the end of paragraph 8.2 about the normalized form of our domain elements. We showed that, in a normalized element of our domain, every block of the list would have had the shape of $[T]^{1,1}$ or $[T]^{0,max}$, where T is a generic set of strings. In our case:

- If the first block of the list is $[T]^{0,max}$, then we have too much uncertainty on how the string begins: we cannot have precise information about the first index of a certain character.

- If the first block of the list is $[T]^{1,1}$, though, we are sure that the string will begin with any of the strings in T. We can check if all the strings in T contain the character we are searching for, and, if that is the case, in which indices.

Otherwise, we can always return > -1 if we find a block (in any position of the list) of the form $[T]^{1,1}$ and each string in T contains the character we are looking for. In fact, we are certain that such character will be present in the string represented by the abstract value, even though we do not know at which index exactly.

The semantics of this operator in our domain will thus be:

$$S[[\text{indexof}, \bar{s}, \text{c}]] = \begin{cases} search(T,c) & \text{if } \bar{s}[0] = [T]^{(1,1)} \\ > -1 & \text{if } \exists\, \mathcal{B} \in \bar{s} : \mathcal{B} = [T]^{1,1} \wedge \left(\forall t \in T : t.Contains(c)\right) \\ \top & \text{otherwise} \end{cases}$$

where $search(T, c)$ is as follows (in pseudo-code):

```
search(T,c)
{
      if (∃ s ∈ T : c ∉ s)
            return Top;
      else
            return { i : ∃s ∈ T t.c. indexOf(s,c) = i };
}
```

This semantics is possible if the numerical domain we are working in collaboration with supports sets of values. Note that, if T contains only one string and such string contains the character c, we return a singleton set with the exact value of the index we are searching for.

Some examples to clarify the semantics we have presented:

- Let $\overline{s} = [abc]^{1,1}[\{d,e\}]^{0,3}[b]^{1,1}$ and $\mathrm{c} = 'b'$; then $S\big[[\mathrm{indexof}, \overline{s}, \mathrm{c}]\big] = search(\{abc\}, 'b') = \{1\}$, since `indexOf('abc', 'b') = 1`.
- Let $\overline{s} = [\{abc, bac, cab\}]^{1,1}[\{d,e\}]^{0,3}[b]^{1,1}$ and $\mathrm{c} = 'b'$; then $S\big[[\mathrm{indexof}, \overline{s}, \mathrm{c}]\big] = search(\{abc, bac, cab\}, 'b') = \{0,1,2\}$. In fact, the first index in which $'b'$ will appear in the string is among 0 (if the string begins with bac), 1 (if the string begins with abc) and 2 (if the string begins with cab).
- Let $\overline{s} = [acd]^{1,1}[\{d,e\}]^{0,3}[b]^{1,1}$ and $\mathrm{c} = 'b'$; then $S\big[[\mathrm{indexof}, \overline{s}, \mathrm{c}]\big] > -1$, since there exists a block ($[b]^{1,1}$) with the features we require.
- Let $\overline{s} = [acd]^{1,1}[\{d,e\}]^{0,3}[f]^{1,1}$ and $\mathrm{c} = 'b'$; then $S\big[[\mathrm{indexof}, \overline{s}, \mathrm{c}]\big] = \top$.
- Let $\overline{s} = [\{d,e\}]^{0,3}[b]^{1,1}$ and $\mathrm{c} = 'b'$; then $S\big[[\mathrm{indexof}, \overline{s}, \mathrm{c}]\big] > -1$.

8.3.12 lastIndexOf(int c)

This operator returns the index within the string of the last occurrence of the specified character. Here we will have to make considerations similar to the ones we made in the previous paragraph.

First of all, we can return > -1 if the abstract value contains at least one block of the form $[T]^{1,1}$ and each string in T contains the character we are looking for. But we can be more precise than that. If the abstract value is a list made by *only one* block of the form $[T]^{1,1}$, then we can check if the interesting character appears in all the strings in T and (if that is the case) in which indices.

The semantics is thus the following:

$$S\big[[\mathrm{lastIndexOf}, \overline{s}, \mathrm{c}]\big] = \begin{cases} search(T,c) & \text{if } len(\overline{s}) = 1 \land \overline{s}[0] = [T]^{(1,1)} \\ > -1 & \text{if } len(\overline{s}) > 1 \land \exists\, \mathcal{B} \in \overline{s} : \mathcal{B} = [T]^{1,1} \land (t.Contains(c)\ \forall t \in T) \\ \top & \text{otherwise} \end{cases}$$

where $search(T,c)$ is the following (in pseudo-code):

```
search(T,c)
```

```
{
      if (∄ s ∈ T : c ∈ s) //no string in T contains c
            return −1;
      if (∃ s ∈ T : c ∉ s) //some strings in T don't contain c
            return Top;

      //all the strings in T contain c
      return { i : ∃s ∈ T t.c. lastindexOf(s,c) = i };
}
```

Also in this case the semantics is possible if the numerical domain we are working in collaboration with supports sets of values. Some examples to clarify the semantics we have presented:

- Let $\overline{s} = [abc]^{1,1}[\{d,e\}]^{0,3}[b]^{1,1}$ and $\text{c} = 'b'$; then $S\big[[\text{lastIndexOf}, \overline{s}, \text{c}]\big] > -1$, since the list contains more than one element, but there is a block ($[b]^{1,1}$) with the features we require.
- Let $\overline{s} = [\{babc, bacb, cbab\}]^{1,1}$ and $\text{c} = 'b'$; then $S\big[[\text{lastIndexOf}, \overline{s}, \text{c}]\big] = search(\{babc, bacb, cbab\}, 'b') = \{2,3\}$. In fact, the last index in which $'b'$ will appear in the string is between 2 ($babc$) and 3 ($cbab$ and $bacb$).
- Let $\overline{s} = [\{bacd, ca\}]^{1,1}[\{d,e\}]^{0,3}[\{b,f\}]^{1,1}$ and $\text{c} = 'b'$; then $S\big[[\text{lastIndexOf}, \overline{s}, \text{c}]\big] = \top$ because there exist two blocks of the form $[T]^{1,1}$ but in neither one all the strings in T contain $'b'$.

8.3.13 substring(int beginIndex, int endIndex)

This operator returns a new string (`s2`) that is a substring of this string (`s1`). The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`.

If the first block of our abstract value is $[T]^{0,max}$, then we have too much uncertainty on how the string begins: we cannot compute a substring based on start and end indices. If the first block is $[T]^{1,1}$, though, we are sure that the string will begin with any of the strings in T. If all the strings in T are long enough

$(len(t) \geq endIndex)$ we can pack all the possible substrings in a new abstract value, which we will return.

The semantics will thus be:

$$S\left[\left[\text{substring}, \overline{s1}, \text{beginIndex,endIndex}\right]\right] = \begin{cases} [T']^{1,1} & \text{if } \overline{s}[0] = [T]^{(1,1)} \wedge len(t) \geq \text{endIndex } \forall t \in T \\ \top & \text{otherwise} \end{cases}$$

where

$$T' = \{\, t.substring(\text{beginIndex, endIndex}) \; \forall t \in T \,\}$$

Some examples to clarify the semantics we have presented:

- Let $\overline{s1} = [\{abcde, qwertyu, qazwsxedc\}]^{1,1}[b]^{0,3}$, $\text{beginIndex} = 3$ and $\text{endIndex} = 5$; then $S\left[\left[\text{substring}, \overline{s1}, \text{beginIndex,endIndex}\right]\right] = [\{de, rt, ws\}]^{1,1}$, since all the strings of the first block contain at least 5 characters, $abcde.substring(3{,}5) = de$, $qwertyu.substring(3{,}5) = rt$ and $qazwsxedc.substring(3{,}5) = ws$.
- Let $\overline{s1} = [\{abcde, qwertyu, qazwsxedc\}]^{1,1}[b]^{0,3}$, $\text{beginIndex} = 3$ and $\text{endIndex} = 7$; then $S\left[\left[\text{substring}, \overline{s1}, \text{beginIndex,endIndex}\right]\right] = \top$, because not all the strings of the first block contain at least 7 characters.
- Let $\overline{s1} = [\{bacd, ca\}]^{0,1}[\{d, e\}]^{0,3}[\{b, f\}]^{1,1}$, $\text{beginIndex} = 0$ and $\text{endIndex} = 2$; then $S\left[\left[\text{substring}, \overline{s1}, \text{beginIndex,endIndex}\right]\right] = \top$ because the first block of the abstract value $\overline{s1}$ has not $min = max = 1$.

8.3.14 contains(CharSequence seq)

This operator returns true if and only if the string (`s1`) contains the specified sequence of char values (`seq`). If the sequence of characters `seq` appears in *all* the strings of a certain block with indices $min = max = 1$, then we can return `true`. We can return `false`, instead, if we are sure that some character of `seq` does not appear in *any* block. Otherwise, we will have to return ⊤. In conclusion, the results of this method will be the following:

$$S\left[\left[\text{contains}, \overline{s1}, \text{seq}\right]\right]$$
$$= \begin{cases} \text{true} \;\; \text{if} \, \exists \mathcal{B} \in \overline{s1} : \mathcal{B} = [T]^{1,1} \wedge \left(\forall t \in T : t.Contains(\text{seq})\right) \\ \text{false} \;\; \text{if} \, \exists c \in seq : \left(!\, t.Contains(c) \; \forall \mathcal{B} = [T]^{min,max} \in \overline{s1}\,, \forall t \in T\right) \\ \quad \top \;\; \text{otherwise} \end{cases}$$

Some examples of the semantics of this operator:

- Let $\bar{s} = [abc]^{1,1}[\{d,e\}]^{0,3}[b]^{1,1}$ and $\text{seq} = {}'ab'$; then $S\left[\left[\text{contains}, \overline{s1}, \text{seq}\right]\right] = \text{true}$, since there is a block ($[abc]^{1,1}$) with the features we require ($min = max = 1$ and each string of the block contains the sequence).
- Let $\bar{s} = [\{babc, bacb, cbab\}]^{1,1}$ and c = $'ba'$; then $S\left[\left[\text{contains}, \overline{s1}, \text{seq}\right]\right] = \text{true}$. In fact, there is a block ($[\{babc, bacb, cbab\}]^{1,1}$) with the features we require ($min = max = 1$ and each string of the block contains the sequence $'ba'$).
- Let $\bar{s} = [\{bacd, ca\}]^{1,1}[\{d,e\}]^{0,3}[\{b,f\}]^{1,1}$ and c = $'bcf'$; then $S\left[\left[\text{contains}, \overline{s1}, \text{seq}\right]\right] = \top$ because it does not exists a block with the feature we require (so we cannot return `true`), plus each character of the sequence appears in some block (so we cannot return `false`).
- Let $\bar{s} = [\{bacd, ca\}]^{1,1}[\{d,e\}]^{0,3}[\{b,f\}]^{1,1}$ and c = $'gbac'$; then $S\left[\left[\text{contains}, \overline{s1}, \text{seq}\right]\right] = \text{false}$ because the character $'g' \in \text{seq}$ does not appear in any block of the list.

9 Abstract domain based on type graphs

In the first three domains presented in the thesis ($\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$), the only focus was character inclusion. In the next two domains ($\mathcal{A}_4, \mathcal{A}_5$) we also considered order, but limited at the beginning (prefix) or at the end (suffix) of the string. In the last domain ($\mathcal{A}_6$) we considered (like in $\mathcal{A}_4$ and $\mathcal{A}_5$) both inclusion and order among characters, but this time it was not limited to the beginning or the end of the string. In $\mathcal{A}_6$ a string was approximated as a list of bricks. Each brick represented a set of strings. The strings corresponding to a list of bricks were thus the concatenation of all possible strings of each brick in the list. For example, we were able to track information like '*this string begins with "SELECT ", then it has some characters which we do not know, then it continues with " FROM TABLE T1" and then it ends with some other characters which we do not know* '. Unfortunately, the operators on the domain (the widening operator, mainly) are not so precise. In such domain we can lose information very quickly.

The new abstract domain we are going to present in this chapter tries to track a kind of information similar to the one tracked by $\mathcal{A}_6$ (inclusion plus order), but with the improvement of undoubtedly better operators (order and widening). This domain exploits *type graphs* (a data structure which represents tree automata), adapting them to represent set of strings.

9.1 Type graphs

Type graphs were introduced in 1992 by (Janssens & Bruynooghe, Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation, 1992) when they developed a method for obtaining descriptions of possible values of program variables (extended modes or a kind of type information). Their method was based upon a framework for abstract interpretation.

A *type graph* T is a triple (N, A_F, A_B) where $T_r = (N, A_F)$ is a rooted tree whose arcs in A_F are called *forward* arcs, and A_B is a restricted class of arcs, *backward* arcs, superimposed on T_r. T_r is called the *underlying* rooted tree of the type graph

T. In a type graph T, $TANC^*(n)$ [respectively, $TANC^+(n)$] is the set of ancestors [respectively, proper ancestors] of the node n in the underlying rooted tree T_r. The backward arcs (n, m) in A_B, have the property that $m \in TANC^*(n)$. $TDESC^*(n)$ [respectively, $TDESC^+(n)$] is the set of descendants [respectively, proper descendants] of the node n in T_r. $GDESC^*(n)$ [respectively, $GDESC^+(n)$] is the set of descendants [respectively, proper descendants] of the node n in the type graph T. Note that $GDESC^*(n) = \{n\} \cup GDESC^+(n)$ (likewise for $TDESC$ and $TANC$). Observe that the set of forward descendants, $TDESC$, only takes into account the arcs A_F in T_r whereas the set of descendants, $GDESC$, takes into account $A_F \cup A_B$. A forward path is a path composed of forward arcs. The depth of a node n, denoted by $\mathrm{depth}(n)$, is the length of the shortest path from the root of the type graph to n.

Each node n of a type graph has a label, denoted by $lb(n)$, indicating the kind of term it describes, and the nodes are divided into three classes:

- *Simple nodes* have a label from the set $\{max, \perp, Int, Real, \ldots\}$, and their outdegree is 0. Some of them have a specific name: *max*-node, ⊥-node.
- *Functor nodes* are labeled with a functor f/k and have outdegree k with $k \geq 0$ (a constant has arity 0).
- OR *nodes* have the label OR and have outdegree k with $k \geq 0$.

We use the convention that n/i denotes the ith son of node n, and the set of sons of a node n is then denoted as $\{n/1, \ldots, n/k\}$ with $k = \mathrm{outdegree}(n)$.

Definition TG1. A type graph $T(N, A_F, A_B)$ has the following characteristics:

1. There is exactly one node, called the root n_0, with no incoming forward arc.
2. All the nodes, except the root, have exactly one incoming forward arc.
3. For each node, except for the root, there is exactly one path consisting of forward arcs and connecting the root with the node.

4. For each backward arc $(n, m) \in A_B$, $m \in TANC^*(n)$.
5. Simple nodes are leaves, functor nodes labeled f/k have outdegree k, and the outdegree of OR nodes is ≥ 0.

The type graphs satisfying Definition TG1 are called *rigid types*.

The graphical representation of the type graphs is straightforward. The nodes of a type graph are represented by their labels and the OR node is encircled. The direction of the arc is indicated by its arrow: forward arcs are drawn downwards, backward arcs upwards. The root of the type graph is the topmost node. Examples are shown in Figure 1 (took from (Janssens & Bruynooghe, Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation, 1992)). More convenient names are *List* for T_2, *ListOne* for T_3, *Tree* for T_4, and *TreeOne* for T_5.

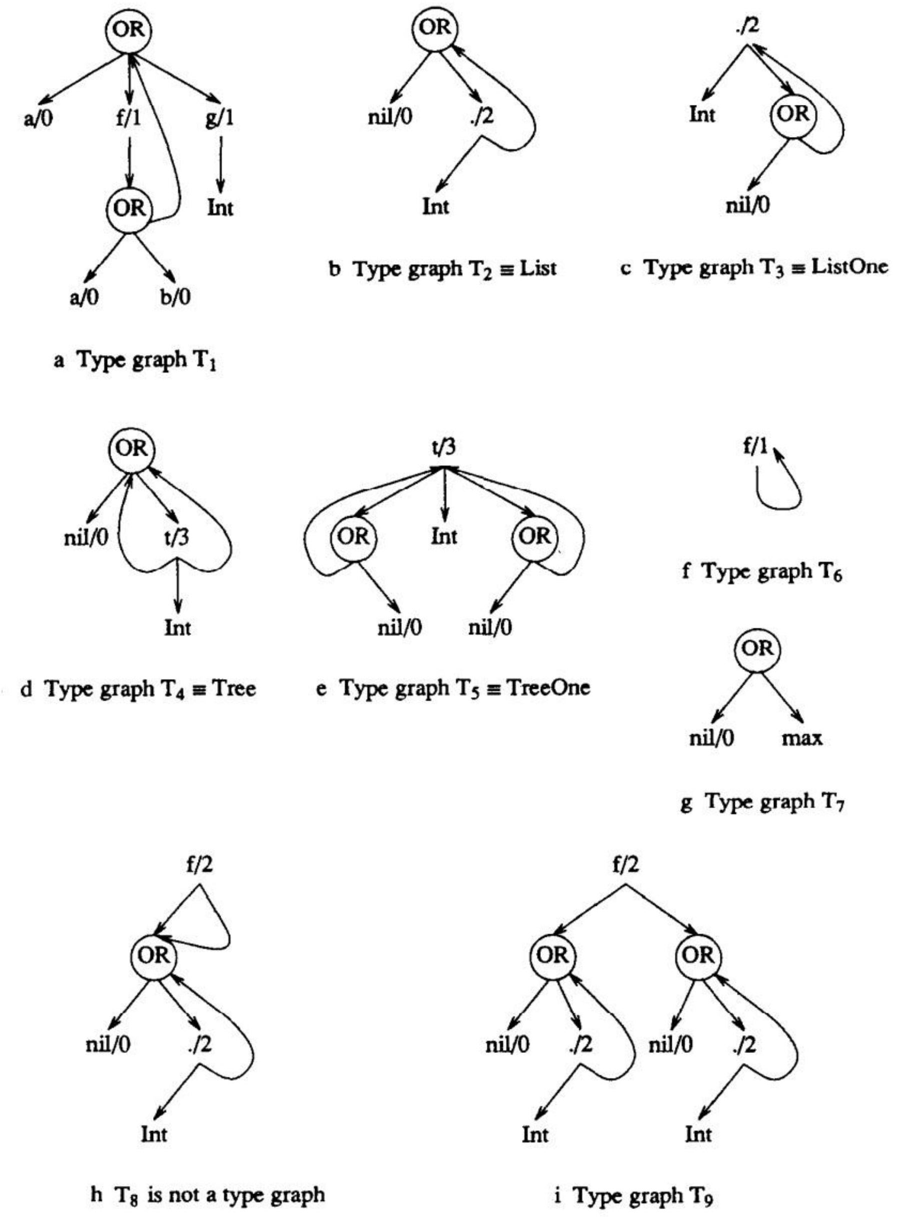

a Type graph T_1
b Type graph $T_2 \equiv$ List
c Type graph $T_3 \equiv$ ListOne
d Type graph $T_4 \equiv$ Tree
e Type graph $T_5 \equiv$ TreeOne
f Type graph T_6
g Type graph T_7
h T_8 is not a type graph
i Type graph T_9
OR
a/0
f/1
g/1
b/0
Int
nil/0
./2
t/3
max
f/2

Figure 1 - Type graphs

The structure of the type graph together with the labels of its nodes determines the set of represented terms. Let S_V be a set of variables, and S_{max} the set of all the terms constructed with the functors and the constants in the program together with the variables from S_V, and the terms in the primitive types (e.g. Int, $Real$, …). We assume that each primitive type P represents a set S_P of ground terms with depth one and that these sets are mutually disjoint. The set of finite terms represented by a node n in the type graph T is said to be the *denotation* of the node n, $\mathbb{D}(n)$.

Definition TG2. The denotation of a node n in a type graph, $\mathbb{D}(n)$ is defined as follows:

- if $lb(n) = max$ then $\mathbb{D}(n) = S_{max}$
- else if $lb(n) = \perp$ then $\mathbb{D}(n) = \emptyset$
- else if $lb(n)$ is a primitive type P then $\mathbb{D}(n) = S_P$
- else if ($lb(n) = f/k$ and $n/1$, … , n/k are its sons) then $\mathbb{D}(n) = \{ f(t_1, \dots, t_k) \mid t_i \text{ is finite} \land t_i \in \mathbb{D}(n/i)\ \forall i \in [1,k] \}$
- else $\mathbb{D}(n) = \bigcup_{i=1}^{k} \mathbb{D}(n/i)$, as $lb(n) = OR$ and $n/1, \dots, n/k$ are its sons.

Note that the order of the sons of a functor node is important because they correspond to the arguments, whereas the order of the sons of an OR-node is irrelevant. Observe that if the condition "t_i is finite" were dropped from the rule for $lb(n) = f/k$, then infinite terms could be included in the set due to backward arcs. $\mathbb{D}(n)$ can be $\emptyset$ or a (finite or infinite) set of finite terms. With n_0 the root of type graph T, we use $\mathbb{D}(T)$ as a synonym for $\mathbb{D}(n_0)$.

For example, referring to figure 1, we have that $\mathbb{D}(List) = \mathbb{D}(T_2)$ is the set of all list of integers.

Context-free grammars provide an alternative way for describing sets of terms. It is straightforward to derive a context-free grammar from a type graph. This correspondence can be formalized by associating a non-terminal symbol T_n with each node n. The grammar rule associated with an OR-node n with successors $n_1, \dots, n_k$ is simply:

$$T_n \coloneqq T_{n_1} | \dots | T_{n_k}$$

The rule associated with a functor node having f as functor and $n_1, \dots, n_k$ as successors is simply:

$$T_n \coloneqq f\left(T_{n_1}, \dots, T_{n_k}\right)$$

The initial symbol of the regular tree grammar is the nonterminal symbol associated with the root of the graph.

From the type graphs T_1, T_2, T_3, T_4 and T_5 of figure 1, we can derive:

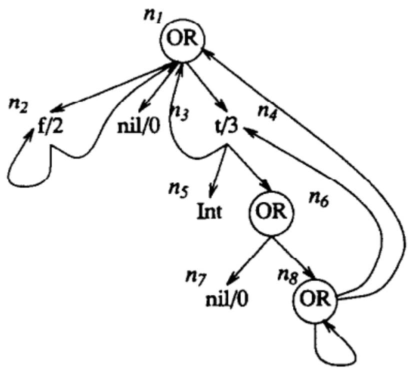
- $T_1 \coloneqq a \mid f(a|b|T_1) \mid g(Int)$
- $T_2 \coloneqq nil \mid .(Int, T_2)$
- $T_3 \coloneqq .(Int, nil \mid T_3)$
- $T_4 \coloneqq nil \mid t(T_4, Int, T_4)$
- $T_5 \coloneqq t(nil \mid T_5, Int, nil \mid T_5)$

Also, deriving a type graph from a context-free grammar is rather straightforward. However, the most obvious derivation does not necessarily result in a type graph, because the underlying trees are explicit in the type graphs, while they are only implicit in a context-free grammar.

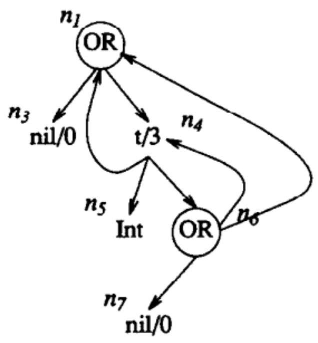
The type graphs are very suitable for representing a set of terms. However, several distinct type graphs can have the same denotation. The existence of superfluous nodes and arcs makes operations needed during abstract interpretation, such as the $\leq$-operation, quite complex and inefficient, because of the increased number of nodes and arcs in the type graph, and also because detecting which set of terms is represented by a node heavily depends on the structure of the type graph. In order to reduce this variety of type graphs, additional restrictions are imposed. In a first step, **compact type graphs** are defined. The expressive power of type graphs is preserved under this restriction. The formal definitions of compact type graphs and of the algorithm of compaction can be found in (Janssens & Bruynooghe, Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation, 1992). We are just going to informally list the main characteristics of a compact type graph:

- Each node (except the root) must have a denotation different than ∅
- An OR-node must have strictly more than one son and each son must *not* be a max-node
- For each forward arc (n, m), if the label of n and m are both OR then $\text{indegree}(m) > 1$
- For each backward arc (n, m), n must be different than m and the forward path $(m, \ldots, n)$ contains at least one functor node

A type graph before compaction is the following:



The same type graph (the denotation is preserved) after the $compact$ algorithm has been executed:



Notice that compact type graphs are not the most economical representation. Nodes in different branches can have the same denotation. In particular, different sons of an OR node may have overlapping, even identical denotations. This makes testing whether a particular term is in the denotation of a compact type graph and the comparison of the denotations of two type graphs inefficient, so we impose a further restriction. Such restriction limits the expressive power of the type graphs but is necessary to achieve efficient operations. First we introduce two functions, $prnd$ and $prlb$. The function $prnd(n)$ denotes the set of *principal nodes* of a node n, and $prlb(n)$ its set of *principal labels*.

$$prnd(n) = \begin{cases} \bigcup_{i=1}^{k} prnd(n/i) \ \text{ if } lb(n) = OR \wedge k = \text{outdegree}(n) \\ \{n\} \text{ else} \end{cases}$$

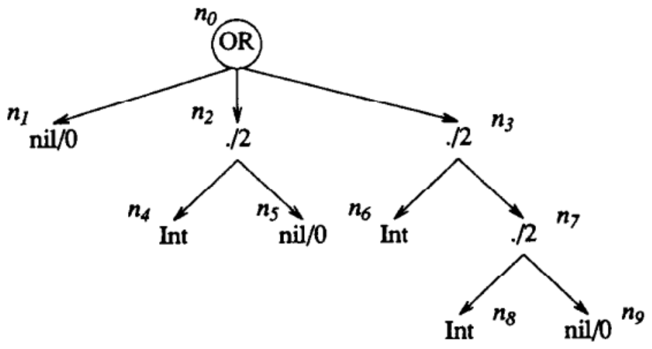
$$prlb(n) = \{\, lb(n_i) \mid n_i \in prnd(n) \,\}$$

The compactness of the type graph assures us that if $lb(n) \neq max$ then $max \neq prlb(n)$. Two sets of principal labels are overlapping if their intersection is nonempty.

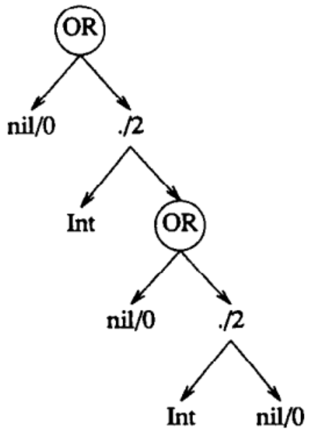
Definition TG3. The *principal label restriction* states that each pair of sons of an OR node must have non-overlapping sets of principal labels.

Normal type graphs are compact type graphs satisfying the principal label restriction. In (Janssens & Bruynooghe, Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation: Definition and Proofs, 1990) it can be found the definition of a normalization algorithm, $normalize(T)$, which takes in input a compact type graph and returns in output the corresponding normal type graphs.

An example of compact (but not normal) type graph is the following:



The corresponding normal type graph obtained through the $normalize$ algorithm is the following:



The principal label restriction limits the expressiveness of the type graph: type graphs violating this restriction sometimes have to be replaced by a type graph denoting a larger set of terms.

9.2 String graphs

We are going to define a modified version of type graphs, which represent strings (instead of types). For this reason, we name them *string graphs*. String graphs have the same basic structure as type graphs. The following differences distinguish string graphs from type graphs:

- **Simple nodes** have labels from the set $\{ max, \bot, \epsilon \} \cup K$; this means that the leaves of our trees can represent:
  - All possible strings, K^* (if the node has label max);
  - No strings, $\emptyset$ (if the node has label $\bot$);
  - The empty string (if the node has label ϵ);
  - Any character taken from our alphabet K, otherwise.
- The only functor we consider is concat (with its obvious meaning of string concatenation). Thus, **functor nodes** are labeled with $concat/k$.
- $\boldsymbol{OR}$**-nodes** remain unchanged.

Let us see some examples of string graphs (the graphical representation is roughly the same as the one for type graphs, the only difference is that we encircle every node).

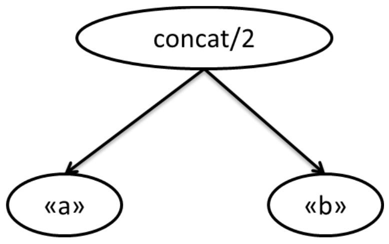


Figure 2 - SG1

The strings graph $SG1$ represents only one string, which is built concatenating a and b. Thus, $\mathbb{D}(SG1) = \{ "ab" \}$. The context free grammar corresponding to $SG1$ is the following (supposing that the nodes are called N_i in breadth-first order):

$$N_1 = concat("a", "b")$$

Another example:

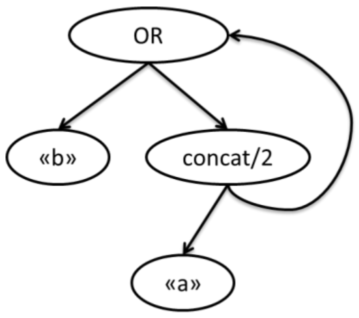


Figure 3 - SG2

This string graph represents an infinite set of strings, that is the set of strings which start with an indefinite number of “a” (even 0) and surely end with a “b”. The corresponding grammar is the following:

$$N_1 = "b" \mid N_2$$

$$N_2 = concat("a", N_1)$$

Using a “regular expression notation”, we can write

$$\mathbb{D}(SG2) = a^*b = \{ "b", "ab", "aab", "aaab", \dots \}$$

One last example:

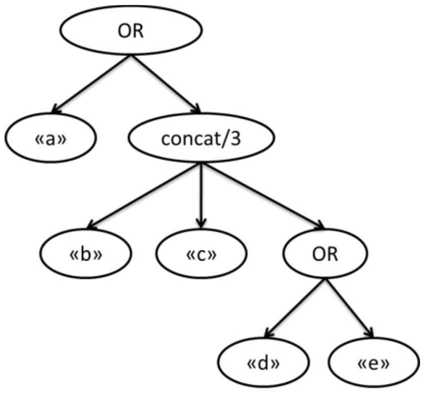


Figure 4 - SG3

This string graph represents a finite number of strings, precisely $\mathbb{D}(SG3) = \{"a", "bcd", "bce"\}$. The corresponding grammar is the following:

$$N_1 = "a" \mid N_2$$

$$N_2 = concat("b", "c", N_3)$$

$$N_3 = "d" \mid "e"$$

Using the notation of (Janssens & Bruynooghe, Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation, 1992), we are going to consider **normal string graphs**, that is compact string graphs which satisfy the principal label restriction. The following string graph does not satisfy the principal label restriction:

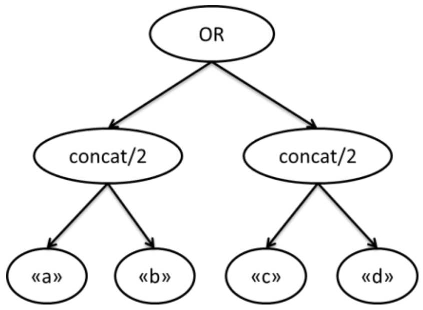


Figure 5 – SG4, not normal

The string graph $SG4$ does not satisfy the principal label restriction, since the two sons of the root node have the same label: $concat/2$. The normalization algorithm produces the following results:

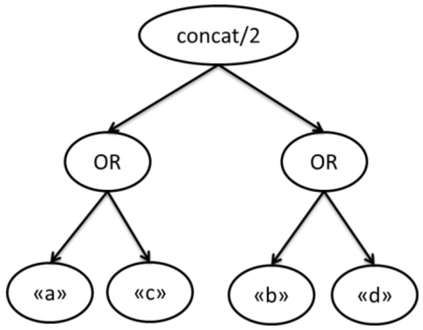


Figura 6 - SG5, normal

The string graph $SG5$ is normal. Its denotation is $\mathbb{D}(SG5) = \{"ab", "ad", "cb", "cd"\}$. We can see that $\mathbb{D}(SG5)$ is a larger set than $\mathbb{D}(SG4) = \{"ab", "cd"\}$. In fact, the normalization process makes us lose the information that, when the first character of the string is a, then the second is always b (and the same for c and d).

Our normal string graph must satisfy, besides the principal label restriction, other four restrictions, which we are now going to introduce.

9.2.1 Normalization Rule 1

The first rule simplifies some naïve occurrences of the functor $concat/k$. In fact, when $concat$ has only one son ($k = 1$), the result of its application is the argument itself. We thus discard every $concat/1$ node, replacing it with its argument.

$\boldsymbol{\mathcal{R}_1}$: Given a node n with label $concat/1$ and $n/1$ as successor, replace n with $n' = n/1$. Any backward arc (m, n) should be replaced with the arc (m, n').

Let us see an example. Consider the following string graph:

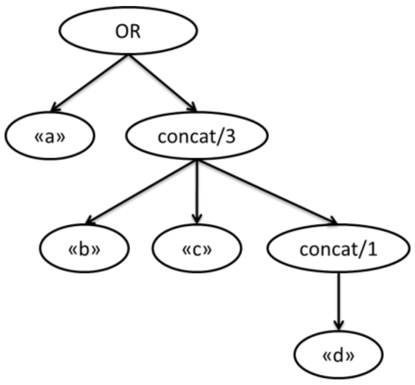


Figure 7 - SG6

After applying normalization rule $\boldsymbol{\mathcal{R}_1}$, it simply becomes:

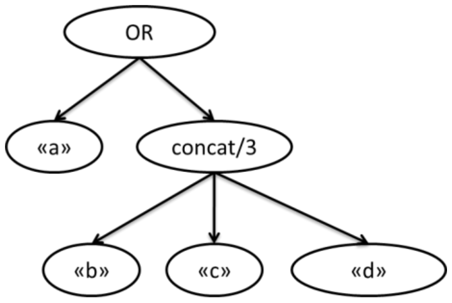


Figure 8 - SG7

We can see that the node $concat/1$ of $SG6$ has been replaced in $SG7$ by its son, the simple node with label “d”.

9.2.2 Normalization Rule 2

The second rule simplifies a node with label $concat/k$ and which successors n/i all have the label max. In fact, the concatenation of *all possible strings* with *all possible strings* gives us *all possible string*, again.

$\boldsymbol{\mathcal{R}_2}$: Given a node n with label $concat/k$ such that $n/i = max\ \forall i \in [1, k]$, replace n with $n' = max$.

Let us see an example. Consider the following string graph:

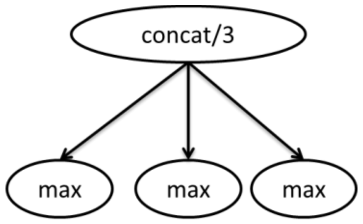


Figure 9 – SG8

After applying normalization rule $\boldsymbol{\mathcal{R}_2}$, it becomes:

Figure 10 - SG9

9.2.3 Normalization Rule 3

The third rule merges two successive sons of a $concat$-node, which labels are both $concat$. In fact, if we concat some characters obtaining s_1, then we concat some other characters obtaining s_2, and finally we concat s_1 and s_2, we obtain the same result as concatenating *all* the characters in the first place.

$\boldsymbol{\mathcal{R}_3}$: Given a node n with label $concat/k$ such that $\exists i : n/i = concat/k_1 \wedge n/(i+1) = concat/k_2$, $\text{indegree}(n/i) = 1$ and $\text{indegree}(n/(i+1)) = 1$, replace n/i and $n/(i+1)$ with a single new node $n' = concat/(k_1 + k_2)$ whose sons are $n'/j = \begin{cases} (n/i)/j & \text{if } j \leq k_1 \\ (n/(i+1))/(j-k_1) & \text{otherwise} \end{cases}$ where $j \in [1, k_1 + k_2]$.

Let us see an example. Consider the following string graph:

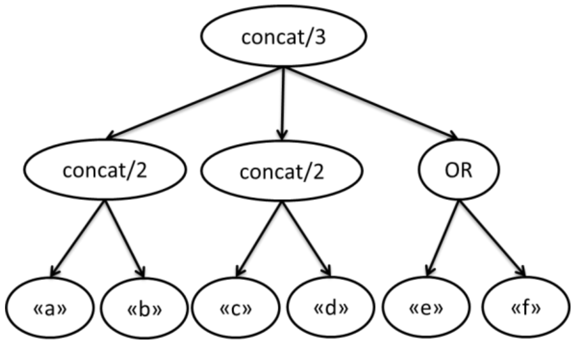


Figure 11 - SG10

In $SG10$ we have two successive sons of a $concat$, which are $concat$ themselves. Thus applying normalization rule $\boldsymbol{\mathcal{R}_3}$, we obtain:

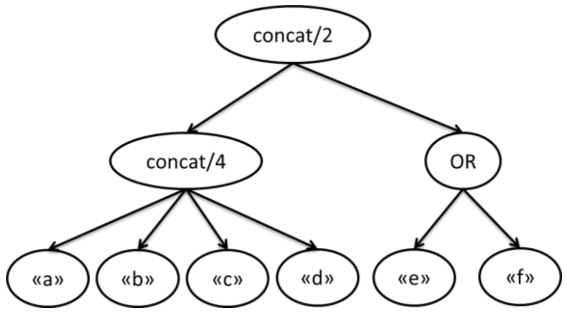


Figure 12 - SG11

In $SG11$ the first two sons of the root have been merged into one single node ($concat/4$), which arity (4) is the sum of the initial arities (2 and 2). Note that, after such simplification, the arity of the root has decreased by one (now it is 2 instead of 3).

9.2.4 Normalization Rule 4

The fourth rule imposes that the sons of a $concat$ node must be simple nodes (leaves), OR-nodes or $concat$ nodes with in-degree > 1. In fact, if a $concat$ node (T_1) has a $concat$ son (T_2) with $\text{indegree} = 1$, we replace T_2 with all the its sons, thus increasing the arity of T_1.

$\boldsymbol{\mathcal{R}_4}$: Given a node n with label $concat/k$ such that $\exists i : n/i = concat/k_1 \wedge \text{indegree}(n/i) = 1$, replace n/i with k_1 nodes such that $n_j = (n/i)/j \; \forall j \in [1, k_1]$.

Let us see an example. Consider the string graph $SG11$ in Figure 12. The root is a $concat$ node with a $concat$ son which indegree is 1. Applying normalization rule $\boldsymbol{\mathcal{R}_4}$, we obtain:

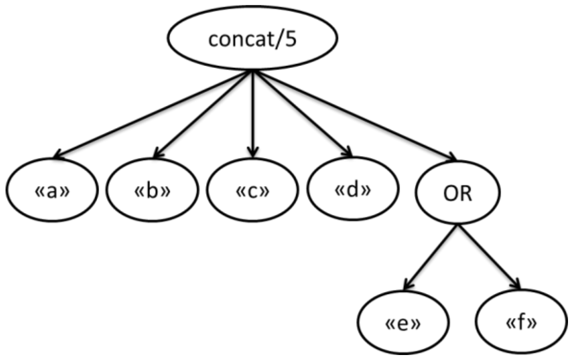


Figure 13 - SG11 bis

The denotation of the string graph has not changed.

Now we are going to prove that such normalization rules do not affect the expressiveness of the string graphs. In fact, the denotation of a string graph does not change after the application of one of our four rules.

9.2.4.1 Theorem: soundness of the normalization rules

Theorem: Given a normalization rule $\mathcal{R}_i$ ($i \in [1,4]$) and a string graph T, suppose that T_1 is the string graph resulting from the application of $\mathcal{R}_i$ to T. Then, $\mathbb{D}(T) = \mathbb{D}(T_1)$.

Proof: Let us consider separately the four cases (one for each normalization rule). Moreover, since our rules always involve a single node and its sons, we can prove that the denotation of such node remains the same after the application of the rule. Then, also the denotation of the entire graph will remain the same.

<u>Rule $\mathcal{R}_1$</u>

Suppose that n is the node with label $concat/1$ and m is its only son. Then, $\mathbb{D}(n) = concat\big(\mathbb{D}(m)\big) = \mathbb{D}(m)$ (since the concatenation of a single string results in the string itself). Rule $\mathcal{R}_1$ replaces n with m, and we proved that the denotation of the two nodes are the same.

<u>Rule $\mathcal{R}_2$</u>

Suppose that n is the node with label $concat/k$ and n/i has label max for each $i \in [1, k]$. Then, $\mathbb{D}(n) = concat(\underbrace{K^*, K^*, \dots K^*}_{k\ times})$. Since K^* represents "all possible strings", we can write $concat\left(\underbrace{K^*, K^*, \dots K^*}_{k\ times}\right) = K^*$ (the concatenation of all possible strings k times results in all possible strings). Then:

$$\mathbb{D}(n) = concat\left(\underbrace{K^*, K^*, \dots K^*}_{k\ times}\right) = K^* = \mathbb{D}(m)$$

where m is a node with label max and replaces node n in T_1.

<u>Rule $\mathcal{R}_3$</u>

Suppose that n is the node with label $concat/k$ and $m_1 = (n/i)$ and $m_2 = (n/(i+1))$ are its two sons with label, respectively, $concat/k_1$ and $concat/k_2$. The denotation of n is:

$$\mathbb{D}(n) = concat(\mathbb{D}(n/1), \dots, \mathbb{D}(n/(i-1)), \mathbb{D}(m_1), \mathbb{D}(m_2), \mathbb{D}(n/(i+2)) \dots, \mathbb{D}(n/k))$$

Assume that m' is the node which replaces m_1 and m_2 in the new string graph T_1. Then, the denotation of the node n' (the node of T_1 corresponding to n in T) is:

$$\mathbb{D}(n') = concat(\mathbb{D}(n/1), \dots, \mathbb{D}(n/(i-1)), \mathbb{D}(m'), \mathbb{D}(n/(i+2)), \dots, \mathbb{D}(n/k))$$

Since string concatenation is an associative operation, to prove that $\mathbb{D}(n) = \mathbb{D}(n')$ we just have to prove that $concat(\mathbb{D}(m_1), \mathbb{D}(m_2)) = \mathbb{D}(m')$. m' is a node with label $concat$ and $k_1 + k_2$ sons, which are the sons of m_1 and of m_2, in this order. Then:

$$\mathbb{D}(m') = concat(\mathbb{D}(m_1/1), \dots, \mathbb{D}(m_1/k_1), \mathbb{D}(m_2/1), \dots, \mathbb{D}(m_2/k_2))$$

Because of the associativity of $concat$, we can say that $concat(a_1, \dots, a_n, b_1, \dots, b_m) = concat\big(concat(a_1, \dots, a_n), concat(b_1, \dots, b_m)\big)$. Thus, in our case:

$$\mathbb{D}(m') = concat(concat(\mathbb{D}(m_1/1), \dots, \mathbb{D}(m_1/k_1)), concat(\mathbb{D}(m_2/1), \dots, \mathbb{D}(m_2 /k_2)))$$

But $concat(\mathbb{D}(m_1/1), \dots, \mathbb{D}(m_1/k_1)) = \mathbb{D}(m_1)$ and $concat(\mathbb{D}(m_2/1), \dots, \mathbb{D}(m_2/k_2)) = \mathbb{D}(m_2)$. Therefore: $\mathbb{D}(m') = concat(\mathbb{D}(m_1), \mathbb{D}(m_2))$ which is exactly what we wanted to prove.

Rule $\mathcal{R}_4$

Suppose that n is a $concat/k$ node and m is one if its sons. m has label $concat/k_1$ and $\text{indegree}(m) = 1$. We know that

$$\mathbb{D}(n) = concat(\mathbb{D}(n/1), \dots, \mathbb{D}(m), \dots, \mathbb{D}(n/k))$$

In T_1 we replace m with all of its sons. The denotation of n' (the corresponding node of n in T_1) is the following:

$$\mathbb{D}(n') = concat(\mathbb{D}(n/1), \dots, \mathbb{D}(m/1), \dots, \mathbb{D}(m/k_1), \dots, \mathbb{D}(n/k))$$

For the associativity of string concatenation, to prove that $\mathbb{D}(n) = \mathbb{D}(n')$ we simply need to prove that $concat(\mathbb{D}(m/1), \dots, \mathbb{D}(m/k_1)) = \mathbb{D}(m)$, but this is the exact definition of $\mathbb{D}(m)$.

We proved our thesis for each possible normalization rule. This concludes our proof. □

After having verified the soundness of our normalization rules, we can see a more complete example of the normalizing process. Consider the following string graph:

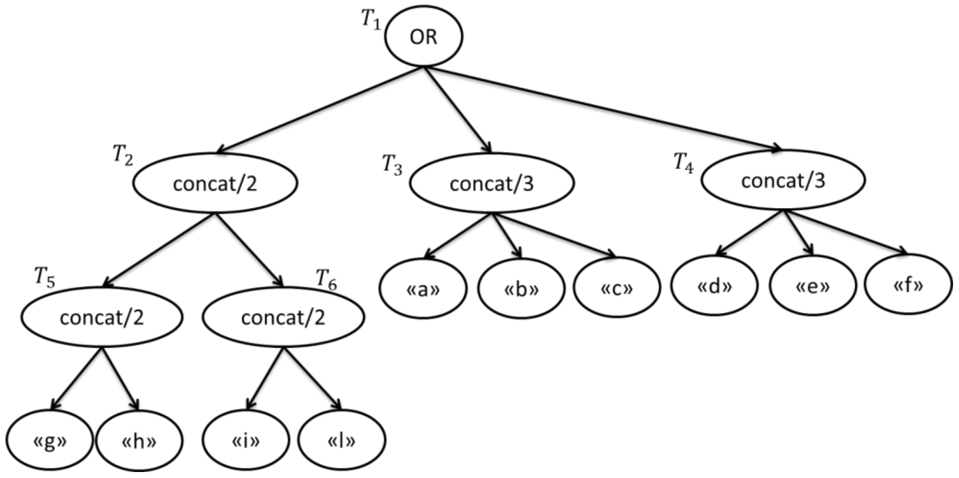


Figure 14 - SG12

The denotation of $SG12$ is $\mathbb{D}(SG12) = \{"ghil", "abc", "def"\}$.

First of all, we can apply $\boldsymbol{\mathcal{R}_3}$ to T_2 and its sons (T_5 and T_6). In fact, T_2 is a functor node with label $concat$ and all of its sons are $concat$ functor too. Applying the normalization rule, we merge T_5 and T_6 in a single node (called T_7), obtaining:

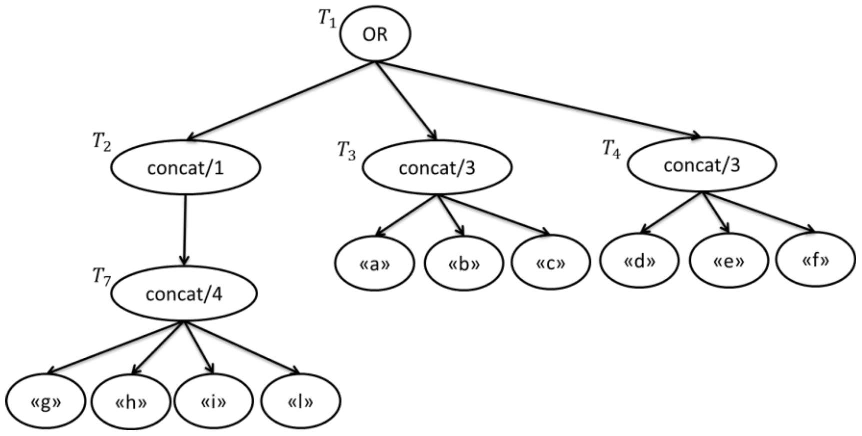


Figure 15 - SG13

The denotation of $SG13$ is not changed with respect to $SG12$, in fact we have $\mathbb{D}(SG13) = \{"ghil", "abc", "def"\}$. Because of the normalization rule applied, though, the arity k of T_2 has decreased, becoming 1. Then, we have to apply $\boldsymbol{\mathcal{R}_1}$ and replace T_2 with its only son, T_7. We obtain:

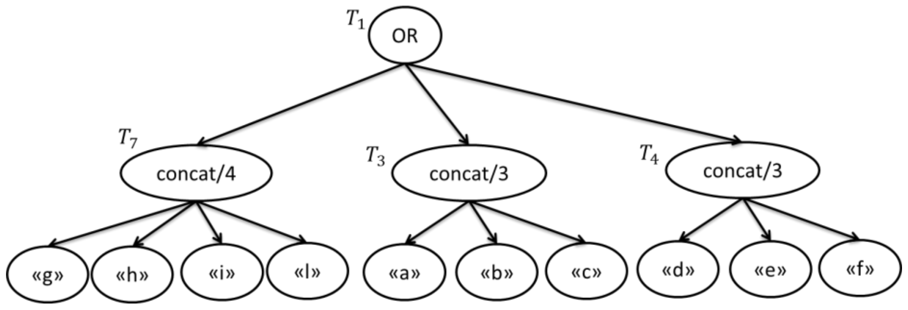


Figure 16 - SG14

The denotation has remained the same:

$$\mathbb{D}(SG14) = \mathbb{D}(SG13) = \mathbb{D}(SG12) = \{"ghil", "abc", "def"\}$$

Note that applying $\boldsymbol{\mathcal{R}_4}$ to T_2 two times would have yield the same results than the successive application of rules $\boldsymbol{\mathcal{R}_3}$ and $\boldsymbol{\mathcal{R}_1}$. In fact, the only order we need to respect while applying our rules is that $\boldsymbol{\mathcal{R}_2}$ comes first (otherwise we could have two, or more, successive sons of a $concat$ node with max label, and this would be anti-economical). The order of application of the other three rules is not relevant.

We have obtained a compact string graph which satisfies $\boldsymbol{\mathcal{R}_1}, \boldsymbol{\mathcal{R}_2}, \boldsymbol{\mathcal{R}_3}$ and $\boldsymbol{\mathcal{R}_4}$. However, the string graph is not normal yet, since it does not satisfy the principal label restriction. In fact, two sons (T_3 and T_4) of an OR-node (T_1) have the exact same label ($concat/3$). We have to merge them in a single node (T_8), obtaining the normal string graph $SG15$:

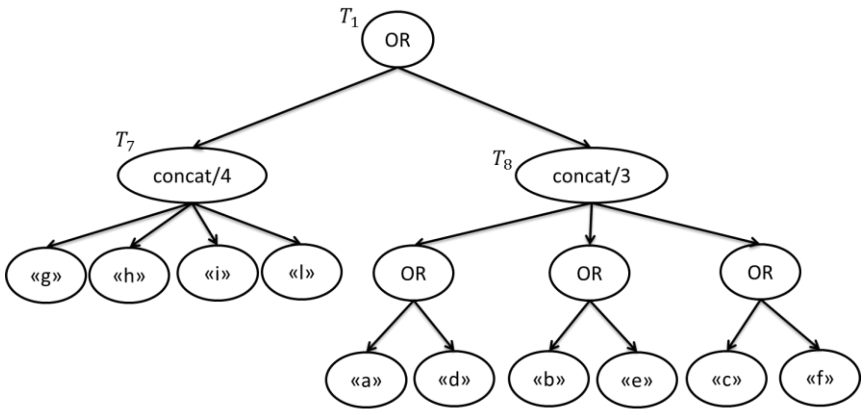


Figure 17 - SG15

The denotation is now increased:

$$\mathbb{D}(SG15) = \{"ghil", "abc", "abf", "aec", "aef", "dbc", "dbf", "dec", "def"\}$$

This representation is the fixpoint of the application of our normalization rules; in fact we cannot apply any more rules to it.

9.3 Lattice

We are now ready to formally define our lattice. Our abstract domain based on string graphs is the following:

$$\mathcal{A}_7 = NSG$$

that is the set of all normal string graphs. An element of $\mathcal{A}_7$ is a normal string graph.

The **bottom** element of this domain is a string graph made by only one node, a $\perp$-node (which denotation is $\emptyset$). The **top** element is a string graph made by only one node, a max-node (which denotation is K^*).

To define the partial order of the domain we can exploit an algorithm defined in (Janssens & Bruynooghe, Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation, 1992). The algorithm $\leq (n, m, \emptyset)$ compares $\mathbb{D}(n)$ with $\mathbb{D}(m)$ and returns true if $\mathbb{D}(n) \subseteq \mathbb{D}(m)$, which is exactly what we want. Thus, given two string graphs T_1 and T_2, to check if $T_1 \leq T_2$ we will compute $\leq (n_0, m_0, \emptyset)$ where n_0 is the root of T_1 and m_0 is the root of T_2:

$$\boxed{T_1 \leq_{\mathcal{A}_7} T_2 \Leftrightarrow \leq \big((n_0, m_0, \emptyset) : \ n_0 = \mathrm{root}(T_1) \wedge m_0 = \mathrm{root}(T_2)\big) \vee \mathrm{root}(T_1) = \bot}$$

The algorithm $\leq (n, m, S^C)$ is the following:

```
if $(n,m) \in S^C$ then true
else if $lb(m) = max$ then true
else if ( $lb(n) = lb(m) = f/k \ \wedge \ k > 0$ )
        then $\forall i \in [1,k] : \leq (n/i, m/i, S^C \cup \{(n,m)\})$
else if ( $lb(n) = lb(m) = OR$ with $k = \mathrm{outdegree}(n)$ )
        then $\forall i \in [1,k] : \leq (n/i, m, S^C \cup \{(n,m)\})$
else if ( $lb(m) = OR \wedge \exists m_d \in \mathrm{prnd}(m) : lb(m_d) = lb(n)$ )
        then $\leq (n, m_d, S^C \cup \{(n,m)\})$
else $lb(n) = lb(m)$
```

The **least upper bound** between two type graphs T_1 and T_2 can be simply computed: we can create a new string graph T which root is an OR-node. The root has two sons, precisely T_1 and T_2. Then, we can apply a compaction algorithm that will transform T in a normal string graph (enforcing the restrictions which define a normal string graph).

$$\bigsqcup(T_1, T_2) = normalize\big(OR(T_1, T_2)\big)$$

For example, consider the two following string graphs:

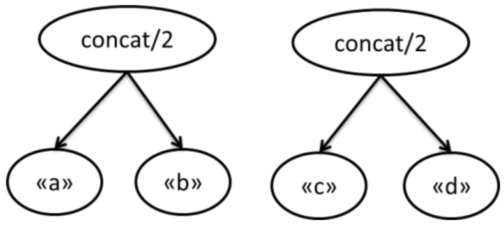


Figure 18 - Two string graphs before the LUB operation

The first step of the least upper bound operation is to create the following string graph:

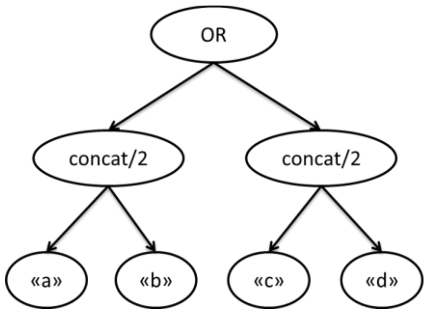


Figure 19 - The first step of the LUB operation

Then, the normalization algorithm has to be applied. We obtain:

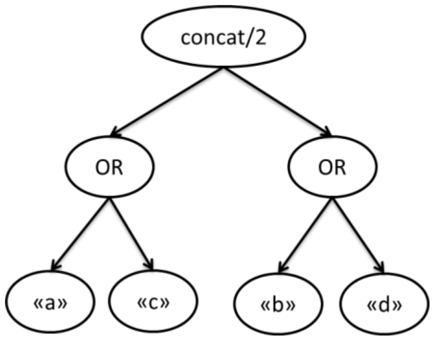


Figure 20 - The final step of the LUB operation

The **greatest lower bound** operator is described in the appendix of (Janssens & Bruynooghe, Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation, 1992). They present an algorithm, $intersection(n_1, n_2)$, which computes the type graph T_{12}, whose denotation is the intersection of the denotations of the type graphs with roots n_1 and n_2. Their strategy to deal with this kind of problem is to leave the old type graphs unchanged and to construct the new type graph step by step. The initialization creates the root l_0 of T_{12} whose required denotation is defined in terms of the nodes n_1 and n_2. At this point the root l_0 is called an *unexpanded leaf*. They define the function "is" which associates at every step in the construction of T_{12} with each node in T_{12} a set of nodes from the given type graphs such that the second function on the nodes of T_{12}, $\mathbb{D}$-is, specifies for each node l of T_{12} its intended denotation.

$$\mathbb{D}\text{-is}(l) = \bigcap_{n \in is(l)} \mathbb{D}(n)$$

Each step extends T_{12} without decreasing the denotation of its nodes. This is done by transforming one of the unexpanded leaves l of T_{12} into a usual node (after the transformation, l is called a *safe node*), and new unexpanded leaves may be added as sons of l. The nodes of T_{12}, in each step of its construction, belong either to S^{ul}, the set of unexpanded leaves, or to S^{sn}, the set of safe nodes.

Now we can proceed to define the abstraction function, always remembering that our concrete domain is made of *set of strings*. A single string s can be seen as a $concat$ node with k sons, where k is the number of characters in s. Each son of the root is a simple node with the corresponding character in the string. For example, "$look$" would become a string graph which root is a $concat/4$ node. The four sons of the root would be simple nodes with label (respectively) "l", "o", "o", "k". The abstraction of a single string is thus:

$$\alpha(s) = concat/k \; \{ \; s[i] : i \in [0, k-1] \; \}$$

where $k = len(s)$.

The **abstraction function** has to abstract a set of concrete strings S (that is, an element of $\mathcal{D}$) into an element of $\mathcal{A}_7$. The solution to this problem is simple: we can abstract all the strings in the set and then compute the least upper bound of them in $\mathcal{A}_7$:

$$\boxed{\alpha(S) = \coprod_{s \in S} \alpha(s) = normalize(\ OR \ \{\ \alpha(s) : s \in S\}\ )}$$

We already defined the **concretization function**, γ. In fact:

$$\boxed{\gamma(T) = \mathbb{D}(T)}$$

We repeat here the definition of $\mathbb{D}(\cdot)$, adapted to *string graphs* instead of type graphs. The denotation of a node n in a string graph, $\mathbb{D}(n)$ is defined as follows:

- if $lb(n) = max$ then $\mathbb{D}(n) = K^*$
- else if $lb(n) = \bot$ then $\mathbb{D}(n) = \emptyset$
- else if $lb(n) \in K \vee lb(n) = \epsilon$ then $\mathbb{D}(n) = \{\ lb(n)\ \}$
- else if ($lb(n) = concat/k$ and $n/1, \ \ldots \ , n/k$ are its sons) then $\mathbb{D}(n) = \{\ concat(t_1, \ldots, t_k) \mid t_i \text{ is finite} \wedge t_i \in \mathbb{D}(n/i)\ \forall i \in [1, k]\ \}$
- else $\mathbb{D}(n) = \bigcup_{i=1}^{k} \mathbb{D}(n/i)$, as $lb(n) = OR$ and $n/1, \ldots, n/k$ are its sons.

As for the **widening operator**, we can exploit the one defined in (van Hentenryck, Cortesi, & Le Charlier, 1995). The authors of the paper argue that such operator leads to accurate results and is effective in keeping the graph sizes small. The widening operator is always applied to an old graph g_{old} and a new graph g_{new} to produce a new graph g_{res}. The main idea behind the widening operator of (van Hentenryck, Cortesi, & Le Charlier, 1995) is to consider two graphs:

$$g_0 = g_{old} \text{ and } g_n = g_{old} \sqcup g_{new}$$

and exploit the topology of the graphs to guess where g_n is growing compared to g_0. The key notion is the concept of *topological clash* which occurs in situations where:

- an OR-node v_0 in g_0 corresponds to an OR-node v_n in g_n where $prlb(v_0) \neq prlb(v_n)$;
- an OR-node v_0 in g_0 corresponds to an OR-node v_n in g_n where $depth(v_0) < depth(v_n)$;

In these cases the widening operator tries to prevent the graph from growing by introducing a cycle in g_n. Given a clash (v_0, v_n) the widening searches for an ancestor v_a to v_n such that $prlb(v_n) \subseteq prlb(v_a)$. If such an ancestor is found and if $v_a \geq v_n$, a cycle can be introduced.

When no ancestor with a suitable $prlb$-set can be found, the widening operator simply allows the graph to grow. Termination will be guaranteed because this growth necessarily adds along the branch of a $prlb$-set which is not a subset of any existing $prlb$-set in the branch. This case happens frequently in early iterations of the fixpoint. Letting the graph grow in this case is of great importance to recover the structure of the type in its entirety.

The last case to consider appears when there is an ancestor v_a with a suitable $prlb$-set, but unfortunately, $v_a \geq v_n$ is false. In this case, introducing a cycle would produce a graph T_r whose denotation may not include the denotation of T_n, and hence the widening cannot perform cycle introduction. Instead, the operation replaces v_a by a new OR-node which is an upper bound to v_a and v_n but decreases the overall size of the graph. The widening is then applied again on the resulting graph.

In conclusion, such widening operator can be viewed as a sequence of transformations on T_n which are of two types: cycle introduction and node replacement, until no more topological clashes can be resolved.

Now we are going to see the **semantics** of the string operators, with the usual notation.

9.3.1 String(char[] data)

This is the string constructor. As we already said, given a concrete string s, its abstraction is a $concat$ node with all the characters that compose the string as sons. The semantics is thus:

$$S\big[[new\ String, \text{arrayChars}]\big] = concat/k\ \{\ \text{arrayChars}[i] : i \in [0, k-1]\ \}$$

where $k = len(\text{arrayChars})$. Consider for example the following code:

```
char[] data = {'a', 'b', 'c'};
String s = new String(data);
```

The resulting abstract value is $concat/3\ \{"a", "b", "c"\}$.

Obviously if we don't know what `data` contains (think about user input), then the semantics will be:

$$S\big[[\text{new String, arrayChars}]\big] = \top$$

9.3.2 concat(String str)

This operator takes two strings and concatenates them. For example, if `s1 = "hello "` and `s2 = "world"` then `s1.concat(s2) = "hello world"`. Suppose that $\overline{s1}$ is the abstract representation of `s1` and that $\overline{s2}$ is the abstract representation of `s2`. The semantics of this operator is:

$$S\left[\left[\text{concat}, \overline{s1}, \overline{s2}\right]\right] = normalize\big(concat/2\ \{\overline{s1}, \overline{s2}\}\big)$$

We create a new string graph, which root is a $concat$ node with two sons. The two sons are the two input abstract values. Then we need to normalize the result, to be sure that it is a normal string graph. In the example cited above, we have:

$$\overline{s1} = concat/6\,\{\,"h","e","l","l","o","\ "\}$$

$$\overline{s2} = concat/5\,\{\,"w","o","r","l","d"\}$$

The abstract value resulting from concatenation of s1 and s2 is:

$$\overline{s_{res}} = normalize\big(concat/2\,\{\overline{s1},\overline{s2}\}\big) =$$

$$concat/2\,\{concat/6\,\{\,"h","e","l","l","o","\ "\}, concat/5\,\{\,"w","o","r","l","d"\}\}$$

Applying rule $\boldsymbol{\mathcal{R}_3}$ and then rule $\boldsymbol{\mathcal{R}_1}$, we obtain:

$$\overline{s_{res}} = concat/11\,\{\,"h","e","l","l","o","\ ","w","o","r","l","d"\}$$

Note that, instead of applying $\boldsymbol{\mathcal{R}_3}$ and then $\boldsymbol{\mathcal{R}_1}$, we could have applied $\boldsymbol{\mathcal{R}_4}$ two times; the result would have been the same.

9.3.3 indexOf(int c)

This operator returns the index within the string of the first occurrence of the specified character. In order to define the semantics of such operator, we must do some considerations about the normalized form of our domain elements. The root of a normal string graph could be an OR-node or a $concat/k$ node. If it is an OR-node, there is too much uncertainty on how the string begins and thus we cannot know the index of the specified character c. We are forced to return $\top$ (or -1 if the character does not appear in the string graph and there are not max-nodes: in this case, such character is certainly not in the string). If the root is a $concat/k$ node, instead, we know that (considering the restrictions we imposed on normal string graphs) its sons could be OR-nodes, simple nodes or $concat$ nodes with $\text{indegree} > 1$. If the first k_1 nodes of the root are simple nodes, then we can look for the character c in such k_1 nodes. If we find it, we can also return its exact index. If the character does not appear in such first k_1 nodes, then we cannot know its index (we will return ⊤ or -1). In fact, the presence of OR-nodes or backward arcs make the shape of the string too uncertain. Note that if the first son of the root is an OR-node or a $concat$ node with $\text{indegree} > 1$ then $k_1 = 0$.

The semantics of this operator in our domain will thus be:

$$S[\![\text{indexof}, \overline{s}, \text{c}]\!] = \begin{cases} i & \text{if } r = \text{root}(\overline{s}) = concat/k \ \wedge lb(r/i) = c \wedge (\forall j \in [0, i-1] : \ lb(r/j) \in K \wedge lb(r/j) \neq c) \\ -1 & \text{if } \nexists n \in \overline{s} : lb(n) = max \vee lb(n) = c \\ > -1 & \text{if } \nexists n \in \overline{s} : lb(n) = max \wedge \exists m \in \overline{s} : (lb(m) = c \wedge OR \notin \text{path}(root, m)) \\ \top & \text{otherwise} \end{cases}$$

where the notation $OR \notin \text{path}(root, m)$ means that the forward path from the root to node m does not contain any OR-node.

Let us see an example to clarify the semantics we have presented.

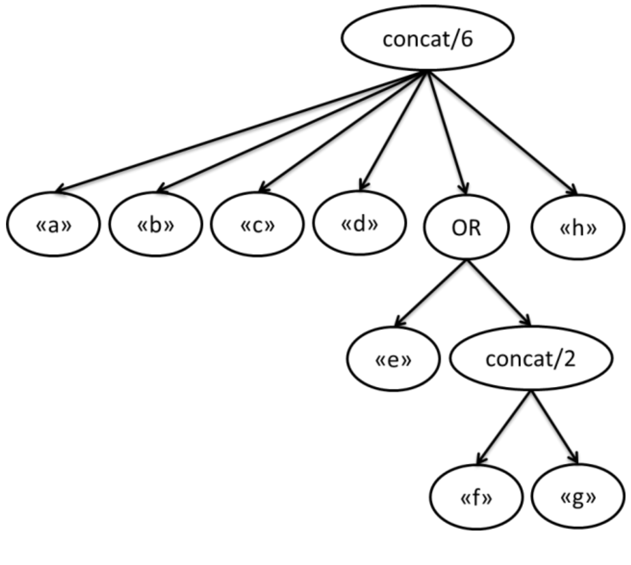


Figura 21 - SG16

Supposing that $\overline{s} = \alpha(SG16)$, then we have:

- $S[[\text{indexof}, \bar{s}, \text{'b'}]] = 1$ since the root node is a $concat$ node, which first $k_1 = 4$ nodes are simple nodes, and in the second son we find the specified character.
- $S[[\text{indexof}, \bar{s}, \text{'m'}]] = -1$ since no node in the string graph contains such character ($'m'$) and there is no node with max label.
- $S[[\text{indexof}, \bar{s}, \text{'h'}]] = > -1$ since there is no node with max label, there is a node which contains our character ($'h'$) and the path from the root to such node does not contain any OR-node.
- $S[[\text{indexof}, \bar{s}, \text{'e'}]] = \top$ since there is a node which contains our character ($'e'$) but the path from the root to such node contains an OR-node.

9.3.4 lastIndexOf(int c)

This operator returns the index within the string of the last occurrence of the specified character. Here we will have to make considerations similar to the ones we made in the previous paragraph. If the root has label $concat$ and all of its sons are leaves (simple nodes) then we can look for the specified character and (if we find it) return the exact index of its last occurrence. Otherwise:

- if the character does not appear in any node of the string graph and the string graph does not contain any max node, we can return -1; we are sure that such character is not in the string;
- if the string graph does not contain any max node and the character appears in a node which path from the root does not contain any OR-node, then we return > -1; in fact, we know that the string will contain such character, but we do not know in which position;
- in all other cases, we return $\top$; we do not have sufficient information to give a more precise answer.

The semantics is thus the following:

$$S[\![\text{lastIndexOf}, \overline{s}, \text{c}]\!] = \begin{cases} i \ \text{if}\, r = \text{root}(\overline{s}) = concat/k \ \wedge lb(r/i) = c \wedge (\forall j \in [0, k-1] : lb(r/j) \in K) \ \wedge (\forall j \in [i+1, k-1] : lb(r/j) \ \neq c) \\ -1 \ \text{if} \nexists n \in \overline{s} : lb(n) = max \vee lb(n) = c \\ > -1 \ \text{if} \nexists n \in \overline{s} : lb(n) = max \wedge \exists m \in \overline{s} : (lb(m) = c \wedge OR \notin \text{path}(root, m)) \\ \top \ \text{otherwise} \end{cases}$$

Let us see an example to clarify the semantics we have presented. Supposing that $\overline{s} = \alpha(SG16)$, then we have:

- $S[\![\text{lastIndexof}, \overline{s}, \text{'m'}]\!] = -1$ since no node in the string graph contains such character ($'m'$) and there is no node with max label.
- $S[\![\text{lastIndexof}, \overline{s}, \text{'h'}]\!] = > -1$ since there is no node with max label, there is a node which contains our character ($'h'$) and the path from the root to such node does not contain any OR-node.
- $S[\![\text{lastIndexof}, \overline{s}, \text{'e'}]\!] = \top$ since there is a node which contains our character ($'e'$) but the path from the root to such node contains an OR-node.

9.3.5 substring(int beginIndex, int endIndex)

This operator returns a new string (`s2`) that is a substring of this string (`s`). The substring begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`.

If the root of our string graph is a $concat$ node and its first $endIndex$ sons are simple nodes (leaves), then we can return the exact substring. Otherwise, we are going to return ⊤.

The semantics will thus be:

$$S[\![\text{substring}, \overline{s}, \text{bI,eI}]\!] = \begin{cases} res \ \text{if}\, r = \text{root}(\overline{s}) = concat/k \ \wedge (lb(r/i) \in K \ \forall i \in [0, eI - 1]) \\ \top \ \text{otherwise} \end{cases}$$

where

$$res = concat/(eI - bI)\ \{\ (r/i)\ \forall i \in [bI, eI - 1]\ \}$$

Some examples of the semantics of this operator (always considering $\overline{s} = \alpha(SG16)$):

- $S[\![\text{substring}, \overline{s}, 0{,}3]\!] = \text{concat}/3\ \{\ \text{'a'}, \text{'b'}, \text{'c'}\ \}$ since the first three sons of the root (a $concat$ node) are leaves and they have label, respectively, 'a', 'b' and 'c'.
- $S[\![\text{substring}, \overline{s}, 2{,}5]\!] = \top$ since the fifth son of the root is an OR-node.

9.3.6 contains(CharSequence seq)

This operator returns true if and only if the string (s) contains the specified sequence of char values (seq). We can return `false` if we are sure that some character of seq does not appear in the string, that is there is no simple node labeled with such character and there is no max node. We can return true if we find the exact sequence seq reproduced by some successive sons (simple nodes) of a concat node m and the path from the root to m does not contain any OR-node. Otherwise, we will have to return $\top$. In conclusion, the results of this method will be the following:

$$S[\![\text{contains}, \overline{s}, \text{seq}]\!] = \begin{cases} \text{true} & \text{if } \exists m \in \overline{s} : m = concat/k\ \wedge OR \notin \text{path}(root, m) \wedge \exists i : (lb(m/(j+i)) = seq[j]\ \forall j \in [0, len(seq) - 1]) \\ \text{false} & \text{if } \exists c \in seq : (\nexists n \in \overline{s} : lb(n) = max \vee lb(n) = c) \\ \top & \text{otherwise} \end{cases}$$

Some examples of the semantics of this operator (always considering $\overline{s} = \alpha(SG16)$):

- $S[\![\text{contains}, \overline{s}, \text{"amc"}]\!] = \text{false}$ since no node in the string graph contains character $'m'$ and there is no node with max label.
- $S[\![\text{contains}, \overline{s}, \text{"abc"}]\!] = \text{true}$ since the first three sons of the root (a $concat$ node) contain exactly the sequence "abc".
- $S[\![\text{contains}, \overline{s}, \text{"cde"}]\!] = \top$ since we are not in any of the previous situations.

10 Domains comparison

In chapters 6, 7, 8 and 9 we have presented five domains:

1. (CC, MC). Strings are abstracted through a pair of sets which represent, respectively, the certainly contained (CC) characters and the maybe contained (MC) characters. An element of this domain represents all the strings which contain all the characters in CC and no character which is not in MC.
2. $Prefix$. Strings are abstracted through a known prefix followed by an unknown suffix ($p\ *$, where p is an array of characters and $*$ represents any string). An element of this domain represents all the strings which start with a certain sequence of characters (the prefix).
3. $Suffix$. Strings are abstracted through an unknown prefix followed by a known suffix ($*\ s$, where s is an array of characters and $*$ represents any string). An element of this domain represents all the string that end with a certain sequence of characters (the suffix).
4. $Bricks\ lists$. Strings are abstracted through a list of bricks. A brick is made by a strings set S and two integer indices min and max: $[S]^{min,max}$. A brick represents all the strings that can be built by concatenating the strings in S between min and max times altogether. If we call $\mathbb{D}([S]^{min,max})$ the denotation of a brick, then the denotation of a list of bricks $\mathcal{B}_i$ ($i \in [1, n]$) is the following: $\{\, b_1 + \cdots + b_n : b_i \in \mathbb{D}(\mathcal{B}_i) \,\}$ where $+$ is string concatenation.
5. $String\ graphs$. Strings are abstracted through string graphs, a variant of the so-called *type graphs*. A string graph is a rooted tree with, in addition, backwards arcs superimposed on it. The nodes have labels, which can be OR, $concat/k$ (where k is the arity of the operation) or a character from our alphabet. The last type of nodes (characters from K) are called simple nodes and are the leaves of the tree. A string graph can be easily transformed in the corresponding context free grammar.

In this chapter we are going to explore the relations between domains and try to build a lattice of domains. We want to understand the degree of precision of each representation and how we can go from one representation to another. The first consideration we can make is that the first three domains are certainly less precise than the last two. In fact, the first domain considers only character inclusion and does not trace their order; the second and the third domains consider also the order, but limiting themselves to a certain part of the string (beginning and end, respectively). It is immediate to see that such three domains have less information than the final two, which trace both inclusion and order along all the string. The relationship between the last two domains is more complicated and we will discuss it in details in a little bit.

10.1 String graphs to (CC, MC)

We argue that string graphs are more precise than the pair of sets of certainly contained and maybe characters. Thus, we can say that (CC, MC) is more abstract than SG (string graphs):

$$SG \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (CC, MC)$$

The function which takes in input a string graph and returns as output a pair of sets can be seen as an *abstraction function*, since it takes us from a certain domain to a more abstract one. On the opposite, the function which takes in input a pair of characters sets and returns as output a <u>set</u> of string graphs can be seen as a *concretization function*.

We are going to consider the **abstraction function** first. The input of such function is a string graph; such string graph is the abstraction of a set of strings. We want to abstract even more, building the pair of sets of certainly contained and maybe contained characters.

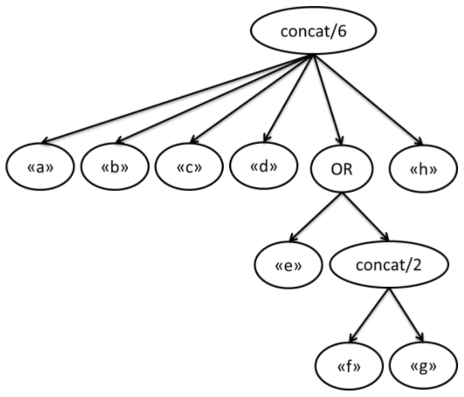
Suppose that T is the string graph and L is the set of leaves of T which label is a character (an element of K):

$$L = \{ \, l : l \in T \wedge lb(l) \in K \, \}$$

Then:

- the set of *maybe contained characters* is $MC = \{ \, c : c = lb(l) \wedge l \in L\}$, that is the characters of all the leaves in T; however, if any leave of the graph T has label max then $MC = K$.
- the set of *certainly contained characters* is $CC = \{ \, c : c = lb(l) \wedge l \in L \wedge OR \notin \mathrm{path}(\mathrm{root}(T), l)\}$, that is the characters of all the leaves in T and which path from the root to such leaf does not contain any OR-node ($OR \notin \mathrm{path}(\mathrm{root}(T), l)$).

Let us see an example. Consider the following string graph:



The result of the application of the abstraction function is:

$$\begin{cases} CC = \{a, b, c, d, h\} \\ MC = \{a, b, c, d, e, f, g, h\} \end{cases}$$

We have defined the abstraction function, that is how to go from a string graph to the pair of sets of certainly contained and maybe contained characters. Now we are going to define the **concretization function**: given the pair of sets (CC, MC) we have to build the set of corresponding string graphs. To enlighten the notation, we define two function, cn and un, which take a string graph as input and return a set of nodes as output:

$$un(T) = \{\, n \in T : lb(n) \in K \,\}$$

$$cn(T) = \{\, n \in T : lb(n) \in K \wedge OR \notin \mathrm{path}(\mathrm{root}(T), n) \,\}$$

The set returned by $un(T)$ contains the leaves of the string graph T which have labels in K (characters from our alphabet). They are the ***u**ncertain **n**odes*. The set returned by $cn(T)$, instead, contains the leaves of the string graph T which have labels in K (characters from our alphabet) and which path from the root does not contain any OR-node. They are the ***c**ertain **n**odes*.

We can also define the corresponding sets of labels:

$$unlb(T) = \{\, lb(l) : l \in un(T) \,\}$$

$$cnlb(T) = \{\, lb(l) : l \in cn(T) \,\}$$

Now we can define the concretization function:

$$\gamma(CC, MC) = \{\, T : unlb(T) = MC \wedge cnlb(T) = CC \,\}$$

This means that, given two sets CC and MC, their concretization is the set of string graphs such that CC corresponds to the labels of the certain nodes ($cnlb(T)$) and MC corresponds to the labels of the uncertain nodes ($unlb(T)$). If $MC = K$, then T is also allowed to have max-nodes.

10.2 String graphs to Prefix

We argue that string graphs are more precise than the prefix domain. Thus, $Prefix$ is more abstract than SG (string graphs):

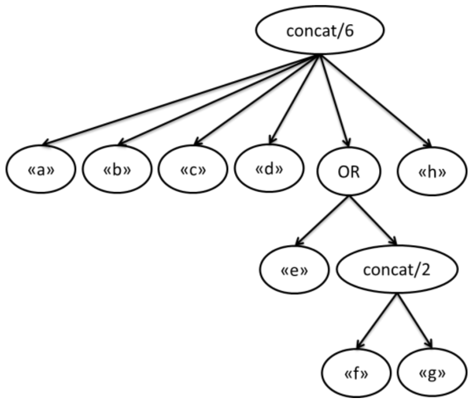
$$SG \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} Prefix$$

Let us define first the **abstraction function**, which takes in input a string graph and returns as output a prefix. If the root of the string graph (let us call it T) is an OR-node, then we do not know how the string begins, and we will have to return the empty prefix (an asterisk only: $*$). If the root is a $concat/k$ node, then we can take its first successive k_1 sons which have label $\in K$: they form the prefix. If the first son is an OR-node or another $concat$ node (with $\text{indegree} > 1$), then we will have to return $*$ even in this case (since $k_1 = 0$). The abstraction function is thus:

$$\alpha(T) = \begin{cases} P* & \text{if } r = \text{root}(T) = concat/k \wedge \exists k_1 > 0 : (lb(r/i) \in K \; \forall i \in [1, k_1] \wedge lb(r/(k_1+1)) \notin K) \\ * & \text{otherwise} \end{cases}$$

where P is an array of k_1 characters such that $P[i] = lb(r/(i+1)) \; \forall i \in [0, k_1 - 1]$.

Let us see an example. Consider the following string graph:



The result of the application of the abstraction function is:

$$P = "abcd"$$

On the opposite of the abstraction function, the **concretization function** takes in input a prefix and returns as output the following string graph:

$$\gamma(P*) = \{ T : (r = \mathrm{root}(T) = concat/k) \wedge (k > k_1) \wedge (\forall i \in [1, k_1] : lb(r/i) = P[i-1]) \}$$

where $k_1 = len(P)$.

10.3 String graphs to Suffix

This relation is very similar to the one between string graphs and prefixes, which we already explored. We argue that string graphs are more precise than the suffix domain. Thus, $Suffix$ is more abstract than SG (string graphs):

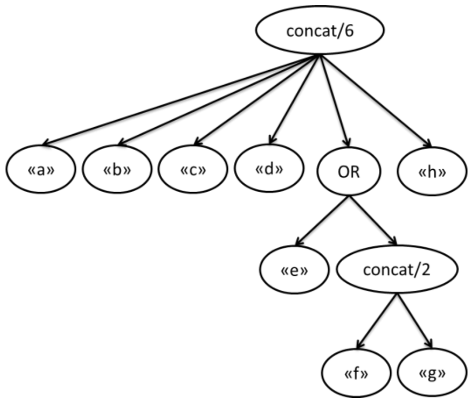
$$SG \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} Suffix$$

We start defining the **abstraction function**, which takes in input a string graph and returns as output a suffix. If the root of the string graph (let us call it T) is an OR-node, then we do not know how the string ends, and we will have to return the empty suffix (an asterisk only: $*$). If the root is a $concat/k$ node, then we can take its <u>last</u> successive k_1 sons which have label $\in K$: they form the suffix. If the last son is an OR-node or another $concat$ node (with $\mathrm{indegree} > 1$), then we will have to return $*$ even in this case (since $k_1 = 0$). The abstraction function is thus:

$$\alpha(T) = \begin{cases} *S \text{ if } r = \mathrm{root}(T) = concat/k \wedge \exists k_1 > 0 : (lb(r/(k-i)) \in K\ \forall i \in [0, k_1 - 1] \wedge lb(r/(k-k_1)) \notin K) \\ * \text{ otherwise} \end{cases}$$

where S is an array of k_1 characters such that $S[i] = lb(r/(k - k_1 + i + 1))\ \forall i \in [0, k_1 - 1]$.

Let us see an example. Consider the following string graph:



The result of the application of the abstraction function is:

$$S = "h"$$

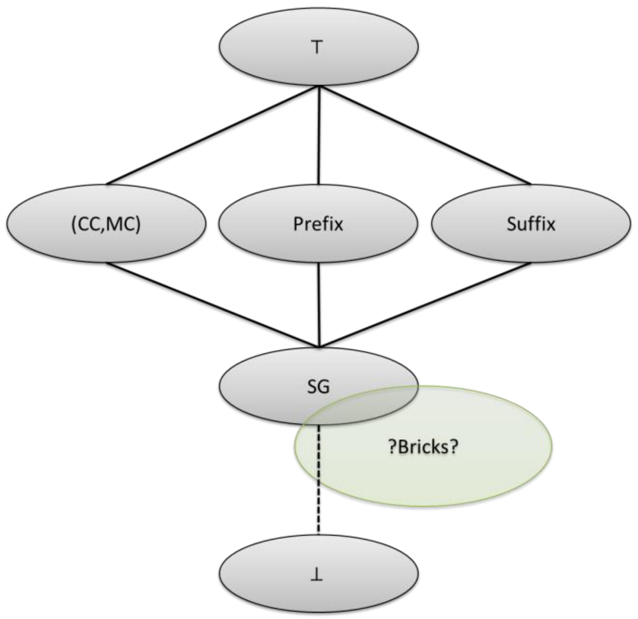
On the opposite of the abstraction function, the **concretization function** takes in input a suffix and returns as output the following string graph:

$$\gamma(*\,S) = \{\, T : (r = \mathrm{root}(T) = concat/k) \wedge (k > k_1) \wedge (lb(r/(i+1)) \\ = S[i-1]\ \forall i \in [1, k_1]) \,\}$$

where $k_1 = len(S)$.

10.4 Recap

Up until now we showed that the three domains (CC, MC), $Prefix$ and $Suffix$ are all more abstract than the domain of string graphs (SG). Thus we can start drawing our lattice of domains:



In the above figure we see that SG is closer to ⊥ (is more precise) than the other three domains and that it is connected to all three. Since we do not know the collocation of $Bricks$ yet, we represented it in transparency and we put a question mark on it. In the following paragraph we are going to explicit the relationship between SG and $Bricks$; after that we will be able to put the $Bricks$ domain in its right place in the lattice.

10.5 Bricks to String Graphs

The relationship between string graphs and the first three domains was relatively easy, since the string graphs traced more information than any of those domains. (CC, MC) traces only characters inclusion, while string graphs trace also order between characters. $Prefix$ and $Suffix$ trace both inclusion and order, but only for a limited part of the string (the beginning or the end), while string graphs trace information along all the string.

In the case of bricks versus string graphs, the comparison is more complex. In fact, bricks trace information about character inclusion and order along all the string, too. The primitives used by each domain are different: $Bricks$ domain uses concatenation of bricks ($\mathcal{B}$), integer indices to indicate repetitions (min, max) and sets of strings (S) as elements of bricks. $String\ graphs$ domain, instead, uses rooted tree with backwards arcs, two types of functors nodes (OR and $concat/k$) and characters as labels of simple nodes ($c \in K$). Their precision is almost the same, but each domain has its own way of obtaining it. The main differences are the following:

- string graphs has OR-nodes which bricks do not have; $Bricks$ domain can trace alternatives *inside* bricks (intra-bricks, thanks to the set of strings inside each single brick) but not *outside* (inter-bricks, like: "these three bricks **OR** these other two"). In this case, we could say that string graphs are more precise than bricks, since they can express information very difficult to express with bricks.
- bricks has the indices, min and max, which string graphs do not have; string graphs have backward arcs which allow repetitions of patterns, but they can be traversed how many times we want (even infinite times). With bricks, instead, we can indicate exactly how many times a certain pattern should be repeated (between min and max times). This makes bricks more expressive than string graphs. We could make up for this defect of string graphs (by "duplicating" the part of the tree which should be repeated, how many times is needed) but this would *vastly* expand the size of the tree used to approximate strings. The operations needed during abstract interpretation would become complex and inefficient, due to the increased number of arcs and nodes in the string graph.

Thus, we have seen that each domain has its merits and its flaws; they both trace a lot of information, but it is hardly comparable, due to the significant structural differences between them.

Since $Bricks$ domain is not comparable with $String\ Graphs$ domain, we need to compare it with the other three domains, in order to find its place in the lattice of domains.

10.6 Bricks to (CC, MC)

We argue that bricks are more precise than the pair of sets of certainly contained and maybe characters. Thus, we can say that (CC, MC) is more abstract than $Bricks$:

$$Bricks \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (CC, MC)$$

We are going to define the abstraction function, which takes in input a list of bricks and returns as output a pair of sets. Suppose that $\mathcal{L}$ is the list of bricks. In the paragraph about the $Bricks$ domain, we showed that, using the normalized representation, bricks could have indices $min = max = 1$ ($[S]^{1,1}$) or $min = 0$ and $max > min$ ($[S]^{0,M>0}$). Then we define:

$$cs(\mathcal{L}) = \{\, s : \big(\exists S : (s \in S \wedge [S]^{1,1} \in \mathcal{L} \wedge\ |S| = 1)\big)\,\}$$

$cs(\mathcal{L})$ is the set of the strings contained in bricks with indices $min = max = 1$ and with only one string in S. They are the strings which will certainly be contained in the string represented by $\mathcal{L}$ ($cs =$ "certain strings"). We also define:

$$us(\mathcal{L}) = \{\, s : (\exists S : s \in S \wedge [S]^{m,M} \in \mathcal{L})\,\}$$

$us(\mathcal{L})$ is the set of all the strings contained in bricks of $\mathcal{L}$. They are the strings which *could* be contained in the string represented by $\mathcal{L}$ ($us =$ "uncertain strings"). Then:

$$\alpha(\mathcal{L}) = \begin{cases} MC = \{\, c : \exists s \in us(\mathcal{L}) : s.contains(c)\,\} \\ CC = \{\, c : \exists s \in cs(\mathcal{L}) : s.contains(c)\,\} \end{cases}$$

Let us see an example. Consider the following list of bricks:

$$\mathcal{L}_1 = [\{a\}]^{0,2}[\{bc, de\}]^{1,1}[\{f\}]^{0,3}[\{gh\}]^{1,1}$$

Then, $cs(\mathcal{L}_1) = \{gh\}$ and $uc(\mathcal{L}_1) = \{a, bc, de, f, gh\}$. The result of the application of the abstraction function is:

$$\begin{cases} CC = \{g, h\} \\ MC = \{a, b, c, d, e, f, g, h\} \end{cases}$$

Now that we have defined the abstraction function, we can define the concretization function. The concretization function takes in input a pair of characters sets (CC, MC) and returns as output a <u>set</u> of lists of bricks. We define:

$$cc(\mathcal{L}) = \{\, c : \exists s \in cs(\mathcal{L}) : s.contains(c) \,\}$$

$$uc(\mathcal{L}) = \{\, c : \exists s \in us(\mathcal{L}) : s.contains(c) \,\}$$

Those two sets contain, respectively, all the characters which appear in strings of $cs(\mathcal{L})$ and all the characters which appear in strings of $us(\mathcal{L})$. Now we can easily define the concretization function:

$$\gamma(CC, MC) = \{\, \mathcal{L} : uc(\mathcal{L}) = MC \wedge cc(\mathcal{L}) = CC \,\}$$

This means that, given two sets CC and MC, their concretization is the set of lists of bricks such that CC corresponds to the characters contained in strings of $cs(\mathcal{L})$ (that is, $cc(\mathcal{L})$) and MC corresponds to the characters contained in strings of $us(\mathcal{L})$ (that is, $uc(\mathcal{L})$).

10.7 Bricks to Prefix

We argue that list of bricks are more precise than the prefix domain. Thus, $Prefix$ is more abstract than $Bricks$ (list of bricks):

$$\begin{tikzcd} Bricks \arrow[r, shift left, "\alpha"] & Prefix \arrow[l, shift left, "\gamma"] \end{tikzcd}$$

Let us define first the **abstraction function**, which takes in input a list of bricks and returns as output a prefix. If the first brick of the list ($\mathcal{B}_0$) has indices $min = 0 \wedge max > 0$, then we do not know how the string begins, and we will have to return the empty prefix (an asterisk only: $*$). If $\mathcal{B}_0$ is a brick with indices $min = 1 \wedge max = 1$, then we can take the longest common prefix of the strings inside the brick: that is the prefix. If the longest common prefix is empty then we will have to return $*$ even in this case. The abstraction function is thus:

$$\alpha(\mathcal{L}) = \begin{cases} P * & \text{if } \mathcal{L}[0] = [S]^{1,1} \wedge P = \text{longestCommonPrefix}(S) \\ * & \text{otherwise} \end{cases}$$

Let us see an example. Consider the following list of bricks:

$$\mathcal{L}_1 = [\{abc, abd, abe, abcd\}]^{1,1}[\{bc, de\}]^{0,2}[\{f\}]^{1,1}[\{gh\}]^{0,3}$$

The result of the application of the abstraction function is:

$$P = "ab"$$

On the opposite of the abstraction function, the **concretization function** takes in input a prefix and returns as output the following list of bricks:

$$\gamma(P *) = \{\, \mathcal{L} : (\mathcal{L}[0] = [\{P\}]^{1,1} \wedge len(\mathcal{L}) \geq 1) \,\}$$

Thus, if $P = "ab"$, then $[\{ab\}]^{1,1}[K]^{0,+\infty} \in \gamma(P *)$.

10.8 Bricks to Suffix

This relation is very similar to the one between list of bricks and prefixes, which we already explored. We argue that bricks are more precise than the suffix domain. Thus, $Suffix$ is more abstract than $Bricks$ (list of bricks):

$$Bricks \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} Suffix$$

We start defining the **abstraction function**, which takes in input a list of bricks and returns as output a suffix. If the last brick of the list ($\mathcal{B}_n$) has indices $min = 0 \wedge max > 0$, then we do not know how the string ends, and we will have to return the empty suffix (an asterisk only: $*$). If $\mathcal{B}_n$ is a brick with indices $min = 1 \wedge max = 1$, then we can take the longest common suffix of the strings inside the brick: that is the suffix. If the longest common suffix is empty then we will have to return $*$ even in this case. The abstraction function is thus:

$$\alpha(\mathcal{L}) = \begin{cases} *\,S \text{ if } len(\mathcal{L}) = n \wedge \mathcal{L}[n-1] = [C]^{1,1} \wedge S = \text{longestCommonSuffix}(C) \\ * \text{ otherwise} \end{cases}$$

Let us see an example. Consider the following list of bricks:

$$\mathcal{L}_1 = [\{bc, de\}]^{0,2}[\{f\}]^{1,1}[\{gh\}]^{0,3}[\{abcd, d, ed, fdd\}]^{1,1}$$

The result of the application of the abstraction function is:

$$S = "d"$$

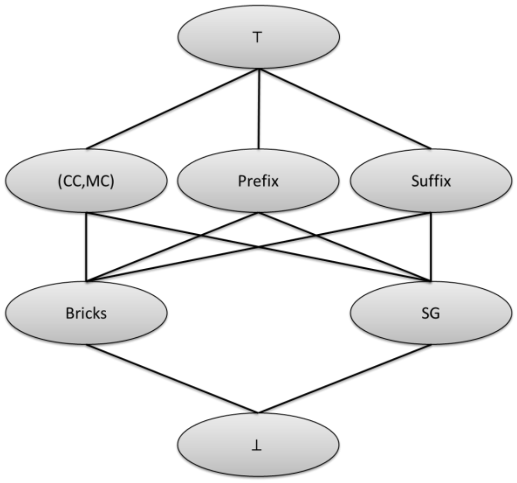
On the opposite of the abstraction function, the **concretization function** takes in input a suffix and returns as output the following list of bricks:

$$\gamma(*\,S) = \{\, \mathcal{L} : (\mathcal{L}[len(\mathcal{L}) - 1] = [\{S\}]^{1,1} \wedge len(\mathcal{L}) \geq 1)\}$$

Thus, if $S = "d"$, then $[K]^{0,+\infty}[\{d\}]^{1,1} \in \gamma(*\,S)$.

10.9 The domains lattice

We showed that $Bricks$ and SG are not comparable. We also showed that the three domains (CC, MC), $Prefix$ and $Suffix$ are all more abstract than the domain of list of bricks ($Bricks$). This information, combined with the one we already gathered before, takes us to the following lattice:



We confirmed the suspicions we described at the beginning of this chapter: the first three domains ($(CC, MC), Prefix, Suffix$) are simple and not so precise; the other two domains ($Bricks, SG$) are more complex but also more precise. $Bricks$ and SG are not comparable, due to some significant structural differences. $Bricks$ also suffers of some imprecise abstract operations (widening, mostly). Thus, we could say that SG is the best domain among all the five we built and described throughout the thesis.

11 Related work

The static determination of approximated values of string expressions has many potential applications. For instance, approximated string values may be used to check the validity and security of generated strings, as well as to collect the useful string properties. In fact, strings are used in many applications to build SQL queries, construct semi-structured Web documents, create XPath and JavaScript expressions, and so on. After being dynamically generated from user inputs, strings are sent to their respective processors. However, strings are not evaluated for their validity or security despite the potential usefulness of such metrics. For these reasons, string analysis has been widely studied.

Hosoya and Pierce (Hosoya & Pierce, 2003) designed a statically typed XML processing language called XDuce based on the theory of finite tree automata. XDuce is an ML-like language for building XML documents that are struct-like values statically typed with regular-expressions. Its sound type system ensures that dynamically generated documents conform to "templates" defined by the document types. In fact, its basic data values are XML documents, and its types (so-called *regular expression types*) directly correspond to document schemas. The correspondence between types and finite tree automata gives the language a powerful mathematical foundation, leading to a simple, clean, and flexible design. Finite tree automata have good mathematical properties including that it is decidable whether the language accepted by one tree automata is included in that accepted by another. XDuce also provides a flexible form of regular expression pattern matching, integrating conditional branching, tag checking, and subtree extraction, as well as dynamic type checking. The type system of XDuce has similarities with string analysis in the sense that it is based on the theory of formal languages. However, XDuce does not directly work on strings, but on trees representing XML documents. This work has some similarities with ours: the regular expression types recall our $Bricks$ domain, the tree automata recall our string graphs domain. However, there are also some great differences: they use type system instead of abstract interpretation and they are focused on building

XML documents, while our focus is on collecting possible values of generic string variables.

It is possible to formulate string analysis as a type system. A type system based on regular expressions was studied by **Tabuchi, Sumii, and Yonezawa** (Tabuchi, Sumii, & Yonezawa, 2002) for λ^{re}–calculus, a minimal functional language with string concatenation and pattern matching over strings. This calculus established a theoretical foundation of using regular expressions as types of strings in text processing languages. By adopting regular expressions as types, they could include rich operations over types in their type structure, and that made it possible to capture precisely the behavior of pattern matching over strings in their type system. This calculus allows in principle limited type inference (types of recursive functions must be given explicitly), but a full type inference algorithm was not given in the paper, due to a technical problem with recursive constraint solving. The paper refers to the Mohri-Nederhof algorithm as a possible venue for future work (which will subsequently be used by (Christensen, Møller, & Schwartzbach, 2003)). The major technical novelties in this paper (with respect to other work such as XDuce (Hosoya & Pierce, 2003)) are:

- the use of regular expression *effects* to statically analyze the shape of the output of an even diverging program;
- the treatment of as-patterns in non-tail positions.

Also in this case (as in XDuce), the approach is very different from ours, since it employs type system. The only resemblance regards regular expressions, which we use in the $Bricks$ domain.

Christensen, Møller and Schwartzbach (Christensen, Møller, & Schwartzbach, 2003) proposed a grammar-based string analysis (implemented in a tool called **JSA**, *Java String Analyzer*) to statically determine the values of string expressions in Java programs. Their algorithm for string analysis can be split into two parts:

1) a front-end that translates the given Java program into a flow graph;

2) a back-end that analyzes the flow graph and generates finite-state automata.

They start constructing flow graphs from Java class files. Then they convert the flow graph into a context free grammar (CFG) where each string variable corresponds to a non-terminal, and each string operation corresponds to a production rule. The size of the resulting grammar is linear in the size of the flow graph. Then, they convert this grammar to a regular language by computing an over-approximation (the regular language is guaranteed to contain all possible values for such string expression). Their approach thus abstracts the set of generated strings in a program into a regular grammar and then performs a grammar inclusion check between the regular grammar and the reference grammar (which is a context-free grammar). Precision loss occurs when the generated strings are abstracted into the regular grammar. This abstraction has to be made, because checking context-free grammar inclusion is costly. Thus, rather than check for context-free language inclusion, the flow equations are over approximated into a regular grammar, using a conversion due to *Mohri and Nederhof* (cited by (Tabuchi, Sumii, & Yonezawa, 2002) as possible future work). They further transform the regular grammars into multi-level finite automata M, which can be then used to generate a *DFA* (Deterministic Finite Automata). The size of the final *DFA* is worst case doubly exponential to the size of M. Queries about grammatical well-formedness are posed as regular expressions, and individual answers are decided by the *DFA*. If a program error is detected, examples of invalid strings are automatically produced. In the paper the authors present extensive benchmarks demonstrating that the analysis is efficient and produces results of useful precision. The potential applications include validity checking of dynamically generated XML, improved precision of call graphs for Java programs that use reflection, and syntax analysis of dynamically generated SQL expressions. However, they did not address context string replacement in this work. Furthermore, JSA tends to be adequate when every string value is stored in a local variable, but it falters when dealing with strings stored in heap variables.

Perhaps the method could be extended to deal with such variables, but not in a straightforward and immediate manner. Finally, their string analysis ensures that the generated object-programs are syntactically correct. However, it does not provide any semantic correctness guarantee of the object-programs.

Let us explain how JSA works with a brief example. The analyzer approximates the set of possible strings that the program may generate for a particular string variable at a particular program location of interest. These locations of interest are called *hotspots*. Consider the following program (P):

```
x = “a”;
while <cond> do
      x = “0” + x;
      x = x + “1”;
print x;                ← Hotspot
```

The possible values of the string variable x at the *hotspot* are expected as follows:

$$\{ 0^n a 1^n : n \geq 0 \}$$

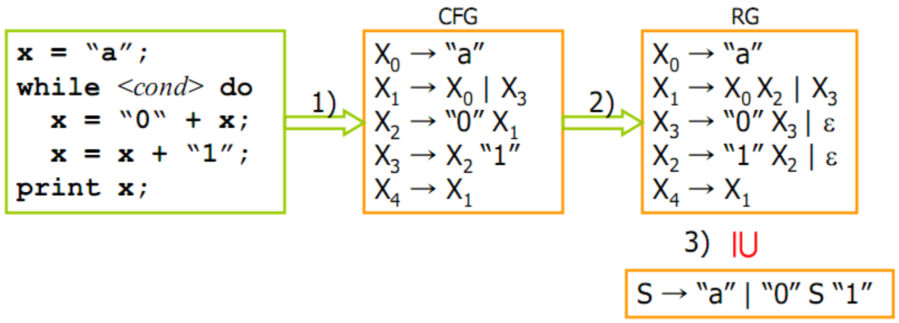
To obtain a sound approximation of the string values by static analysis of the program, we should follow the procedure:

1) Statically analyze the program and determine a context-free grammar G which represents the values of a variable x at the hotspot;
2) See if G is equivalent to the reference grammar: $S \rightarrow$ “a” | “0” S “1”.

However, the problem of checking if a context-free grammar is included in another context-free grammar is “undecidable”. So, JSA does the following:

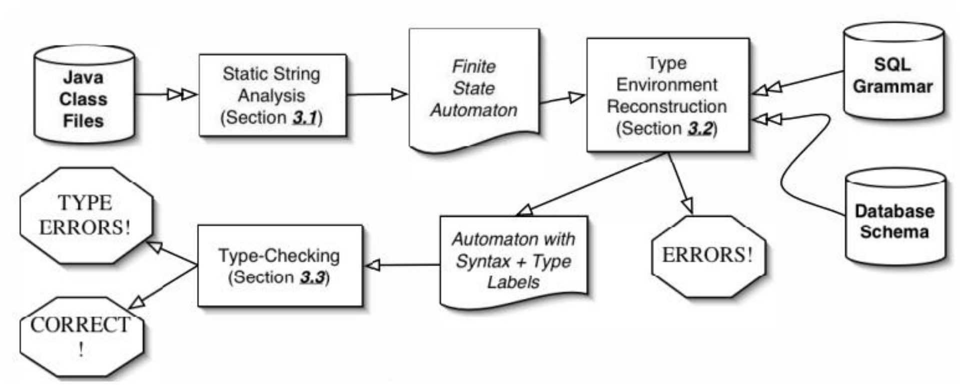
1) extract a context-free grammar G from the program;
2) determine the regular approximation RG of the extracted grammar G;
3) see if the regular grammar RG includes the reference grammar: $S \rightarrow$ “a” | “0” S “1”.

For the program considered above, we have:



This work has considerable similarities with our *String Graphs* domain. In fact, we showed that type/string graphs are closely related to context free grammar. However, JSA encounters a precision loss when the generated strings are abstracted into the regular grammar. In the example above, this happens at step 2). They obtain a regular grammar which *contains* the reference grammar, but they are *not* the same grammar. Preliminary test on our string graphs domain show that, for this example, string graphs are able to model precisely the reference grammar ($S \rightarrow$ "a" | "0" S "1"), without the need of any kind of approximation.

Gould, Su, and Devanbu (Gould, Su, & Devanbu, 2004) use Christensen et al.'s grammar-based string analysis technique to check for errors in dynamically generated SQL query strings in Java-based web applications. They introduce a sound, static, program analysis technique to verify the correctness of dynamically generated query strings. Their approach is based on a combination of automata-theoretic techniques, and a variant of the context-free language (CFL) reachability problem. As a first step, their analysis builds upon a static string analysis to build a conservative representation of the generated query strings as a finite-state automaton. Then, they statically check the finite-state automaton with a modified version of the context-free language reachability algorithm. Their analysis is sound in the sense that if it does not find any errors, then such errors do not occur at runtime. Their tool is able to detect some known and unknown errors in these programs. Furthermore, their analysis empirically appears quite precise, with a low false-positive error rate. Note that their analysis makes use of the string analysis of Java programs reported in (Christensen, Møller, & Schwartzbach, 2003). A scheme of their approach (taken from their paper) is the following:


Java Class Files
Static String Analysis (Section 3.1)
Finite State Automaton
Type Environment Reconstruction (Section 3.2)
SQL Grammar
Database Schema
TYPE ERRORS!
Type-Checking (Section 3.3)
Automaton with Syntax + Type Labels
ERRORS!
CORRECT !

This work is more specific than ours, since it focuses on checking the grammar of dynamically generated SQL queries. String analysis (which is what we do in this thesis) is a prerequisite of (Gould, Su, & Devanbu, 2004). The authors of this paper employ JSA, thus the considerations we made about (Christensen, Møller, & Schwartzbach, 2003) still apply here.

Christodorescu, Kidd, and Goh (Christodorescu, Kidd, & Goh, 2005) present an implementation of string analysis for executable programs for the x86 architecture. Their technique recovers semantic information from binary code (e.g., they perform string inference, alias analysis) and leverages the JSA infrastructure of (Christensen, Møller, & Schwartzbach, 2003) to perform the string analysis. This work is a specific implementation of JSA. We did not have time to give an implementation of our domains.

Thiemann (Thiemann, 2005) introduced a type system for string analysis based on context-free grammars and presented a type inference algorithm based on Earley’s parsing algorithm. In his type system, a program that may generate a document with an arbitrary depth can be type-checked against a context-free grammar. However, the grammar must be carefully crafted and a program must be written so that it is checked against it. The type system was designed for an applied lambda calculus with string concatenation, and it was not discussed how to deal with string operations other than concatenation (while in this thesis we show the semantics of various string operations, not only concatenation). His analysis is more precise than those based on regular expressions, but his type inference algorithm, though sound, is incomplete because its context-free language inclusion problem cannot be solved. The weak point is that the grammar must be written in terms of single characters rather than tokens.

To statically check the properties of Web pages generated dynamically by a server-side program, **Minamide** (Minamide, 2005) developed a static program analysis that approximates the string output of a program with a context-free grammar. His analyzer extracts a flow-equation set for a string expression, treating the equation set as if it were a context-free grammar. The approximation obtained by the analyzer can be used to check various properties of a server-side program and the pages it generates. Such approximation is conservative in the sense that it contains any possible output string that could be generated by the program. His analysis is based on the Java string analyzer (JSA) of (Christensen, Møller, & Schwartzbach, 2003). Both analyzers first extract a grammar with string operations from a program. After this phase, Minamide's string analyzer directly eliminates string operations in the grammar and transforms it into a context-free grammar. In this phase, a string operation is modeled with an automaton with output called a *transducer*. In fact, the novelty of his analysis is the application of finite-state-automata transducers to revise the flow equations due to string-update operations embedded in the program. The transducers are also used to sanitize suspect user input before it is injected into a dynamically generated document. Minamide also supports string-based replacement operations by escaping replace operations to finite-state transducers. Subsequently, Minamide's group developed exponential-time algorithms that validate a context-free grammar against a subclass of balanced context-free grammars, which can be used to validate dynamically generated XML and HTML documents.

Minamide’s analyzer differs from Christensen et al.’s analyzer in the following ways. Firstly, Minamide’s analysis is based on context-free languages instead of regular languages and thus his approximations can be more precise. The analysis is also simplified by removing the transformation of Mohri and Nederhof. Secondly, his analyzer approximates the whole output of a program representing a Web page and thus can be used to check properties of a dynamically generated Web page as a whole.

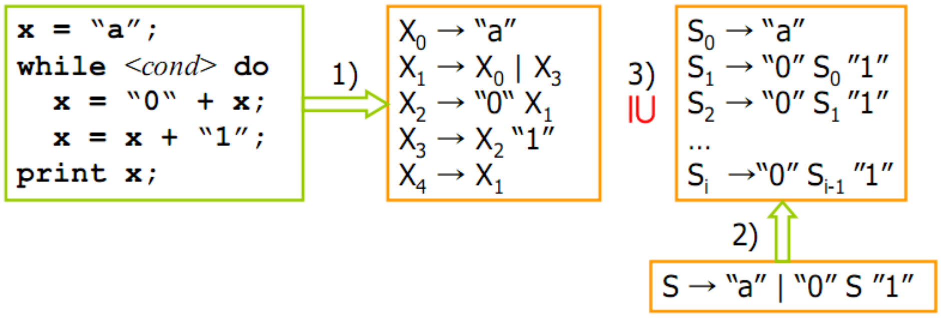
To summarize, Minamide’s analyzer works like this:

1) Extract a context-free grammar from the program;
2) Transform the reference grammar into a regular grammar by restricting the nesting depth of recursion;
3) See if the context-free grammar includes the reference grammar.

Let us considered a small program (P, the same we used to explain JSA):

```
x = “a”;
while <cond> do
      x = “0” + x;
      x = x + “1”;
print x;                  ← Hotspot
```

Minamide’s analysis is the following (the reference grammar is $S \rightarrow$ “a” | “0” S “1”):



```
x = "a";
while <cond> do
  x = "0" + x;
  x = x + "1";
print x;
```

1)
$X_0 \rightarrow$ "a"
$X_1 \rightarrow X_0 \mid X_3$
$X_2 \rightarrow$ "0" X_1
$X_3 \rightarrow X_2$ "1"
$X_4 \rightarrow X_1$
3) $\supseteq$
$S_0 \rightarrow$ "a"
$S_1 \rightarrow$ "0" S_0 "1"
$S_2 \rightarrow$ "0" S_1 "1"
...
$S_i \rightarrow$ "0" S_{i-1} "1"
2)
$S \rightarrow$ "a" | "0" S "1"

This work has some similarities (just like JSA) with our *String Graphs* domain. The preliminary test we already cited show that our work is more precise than (Minamide, 2005). In fact, we do not have the depth restriction, plus we do not need to know the reference grammar *a priori*, since we obtain exactly that grammar only through our domain.

Choi, Lee, Kim, and Doh (Choi, Lee, Kim, & Doh, 2006) used standard abstract-interpretation techniques with heuristic widening to devise a string analyzer that handles heap variables and context sensitivity. Until then, ascertaining widening operators for regular expressions had been believed to be difficult (Christensen, Møller, & Schwartzbach, 2003). However, by selecting a restricted subset of regular expressions as abstract domain, which results in limited loss of expressibility, and by using heuristics, Choi et al. were able to devise a precise widening operator. The widening operator is defined on strings and the widening of a set of strings is achieved by applying the widening operator pairwise to each string pair. String operators, such *concat*, *substring*, *trim* and *replace*, are treated uniformly. The abstract-interpretation framework enables the integration of the following tasks into their analyzer:

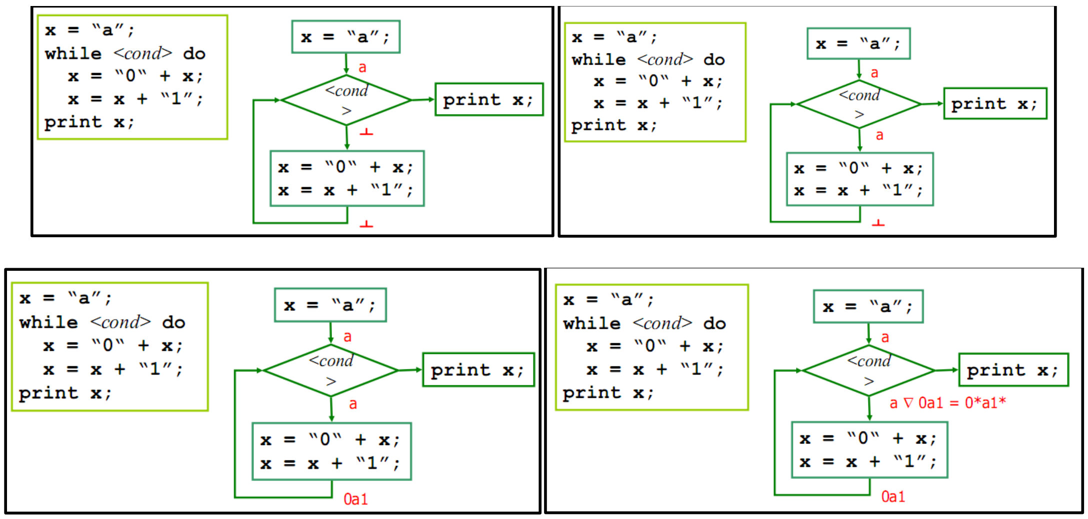
- Handle memory objects and their field variables
- Recognize context sensitivity
- Integrate with constant propagation for integers
- String-based replacement

Their string analyzer is implemented and tested. The results showed the proposed analyzer to be as precise as JSA in all cases, and even more precise for test programs dealing with memory objects and field variables. However, their analyzer lacks the expressibility required for checking the syntax of generated strings and for handling strings with escaped characters. The authors envisioned future work aimed to produce abstract string representations with more expression power while still employing the widening operator of their method.

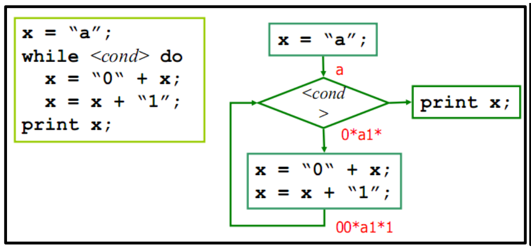
A summary of their method is the following:

1) A string value is abstracted as a “regular string” which is the same as a regular expression except that a consecutive repetition is not allowed, i.e., $0^*1^* \Rightarrow \{0,1\}^*$;
2) Abstract interpretation of a program on the abstract domain of regular strings.

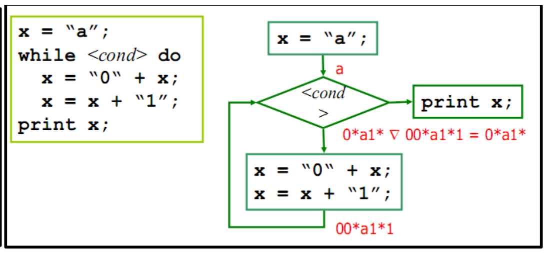
Considering again the small program P, we can depict this method like this:



```
x = “a”;
while <cond> do
   x = “0“ + x;
   x = x + “1”;
print x;
```

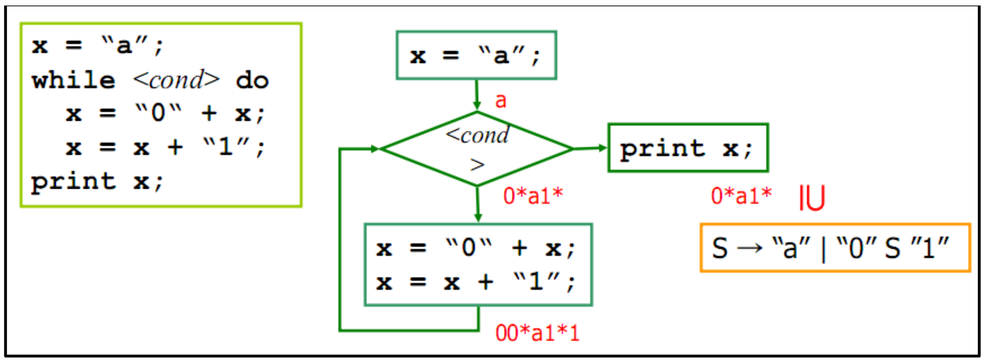


```
x = "a";
while <cond> do
   x = "0" + x;
   x = x + "1";
print x;
```



```
x = "a";
while <cond> do
   x = "0" + x;
   x = x + "1";
print x;
```

Finally, we obtain:



```
x = "a";
while <cond> do
   x = "0" + x;
   x = x + "1";
print x;
```

Our *Bricks* domain is very similar to the work of (Choi, Lee, Kim, & Doh, 2006), but our widening is less precise than theirs. An interesting future work could be trying to plug their widening in our domain. Our *String Graph* domain, instead, seems more precise than their domain. In fact, the preliminary test with string graphs on the code example above produced exactly the reference grammar, while their result (0^*a1^*) is less precise (the number of $0s$ and $1s$ is not constrained to be the same).

Yu, Bultan, Cova, and Ibarra (Yu, Bultan, Cova, & Ibarra, 2008) presented an automata-based approach for the verification of string operations in PHP programs based on symbolic string analysis. In their string analysis approach, they encode the set of string values that string variables can take as deterministic finite automaton (DFA). At each program point, each string variable is associated with a DFA. The language accepted by the DFA corresponds to the values that the corresponding string variable can take at that program point. They implemented all string functions using a symbolic automata representation (MBDD representation from the MONA automata package) and leveraged efficient manipulations on MBDDs, e.g., determinization and minimization. Particularly, they proposed a novel algorithm for language-based replacement. Their replacement function takes three DFAs as arguments and outputs a DFA. Finally, they applied a widening operator defined on automata to approximate fixpoint computations. If this conservative approximation does not include any bad patterns (specified as regular expressions), they conclude that the program does not contain any errors or vulnerabilities. Using the string analysis techniques proposed in this paper, it is possible to automatically verify that a string variable is properly sanitize data program point, showing that attacks are not possible.

Doh, Kim, and Schmidt (Doh, Kim, & Schmidt, 2009) reported a powerful technique called “abstract parsing” that statically analyzes the string values from programs. They combine LR(k)-parsing technology and data-flow analysis to analyze, in advance of execution, the documents generated dynamically by a program. Based on the document language’s context-free reference grammar and the program’s control structure, the analysis predicts how the documents will be generated and parses the predicted documents. Their strategy remembers context-free structure by computing abstract LR-parse stacks. The technique is implemented in Objective Caml and has statically validated a suite of PHP programs that dynamically generate HTML documents. With this paper, they have improved the precision of previous analyses of programs that dynamically generate documents. Note that, before Doh, Kim, and Schmidt’s abstract parsing technique, string analysis works were all “*analyze-then*-parse” techniques. Their technique employs, instead, simultaneous “*analysis-and-parsing*”, which means statically analyze a program that generates strings, and, at the same time, parse the generated strings with the LR(k) reference grammar. The abstracted parse-stack is the abstract denotations of strings. Their technique can finally be summarized like this. Given a string-generating program and a reference grammar for the string variable at a *hotspot*:

- generate data-flow equations from the program;
- solve the equations in the abstract domain of parse stacks of the reference grammar.

Kong, Choi, and Yi (Kong, Choi, & Yi, 2009) presented an abstract interpretation for statically checking the syntax of generated code in two-staged programs. This technique is based on Doh, Kim, and Schmidt's "abstract parsing" (Doh, Kim, & Schmidt, 2009). Kong et al. adopted this technique for two-staged programming languages and formulated it in the abstract interpretation framework. They generalized the abstract-parsing abstract interpretation, as usual, by parameterizing the abstract domain of parse stacks. By choosing an appropriate abstract domain, one can control the analysis precision and cost, as long as the domain satisfies the condition they provide. They also presented a particular instance of the abstract domain, namely an abstract parse stack and its widening with k-cutting. Their contribution is to lay a basis to expose the power and, if any, limitation of abstract parsing as static analysis for multi-staged languages with concatenation. This formulation not only gives a more concise and elegant explanation of the original idea but also decouples the core idea of abstract parsing from its implementation. The parameterizing of their abstract domain lets them control the precision and cost of the analysis. In this thesis, we present various abstract domains which have very different balances of precision and cost. Thus, one can choose the domain more suited to the analysis she has to make. For example, a simple but very efficient analysis can be conducted with the prefix or suffix domain. A more complex (but also more costly) analysis can be made through the string graphs which are definitely more precise than the other domains we presented.

12 Conclusions and future work

String analysis is a static analysis technique that determines the string values that a variable can hold at specific points in a program. This information is often useful to help program understanding, to detect and fix programming errors and security vulnerabilities, and to solve certain program verification problems.

In this thesis we approached string analysis using the abstract interpretation framework. In particular, we focused on the construction of abstract domains. We created five domains:

1. In the first domain we kept trace of the characters surely contained and maybe contained in the string. Character inclusion was our main concern, while we did not consider order amongst characters.
2. In the second domain we represented strings as a known prefix followed by an unknown suffix. Here we considered both character inclusion and order; however, such information was limited to a little part of the string (the prefix).
3. In the third domain we represented strings as an unknown prefix followed by a known suffix. This domain is exactly the opposite of the second one.
4. The fourth domain takes inspiration from regular expressions. Here we represented a string as a sequence of bricks. In this domain we considered both character order and inclusion, along all the string (not only its beginning or its end).
5. In the fifth (and final) domain we adapted a known data structure (type graphs) to represent strings, thus calling them *string graphs*. This domain considered (just like the fourth one) both character inclusion and order on the entire length of the string.

The first three domains are quite simple and the information we can trace with them is limited. However, they are not computationally expensive (the prefix and suffix in particular) and they do not need to define a widening operator. The last two domains are certainly more complex, and they let us trace more interesting patterns. The lattices of such domains are infinite and do not satisfy ACC; thus, we are forced to define a widening operator. The domain based on string graphs, though, features a more tested and more precise widening operator (van Hentenryck, Cortesi, & Le Charlier, 1995). The fourth domain, instead, has a widening operator which can lose information very quickly.

As first future work, there is certainly the implementation of these domains. We can exploit an existing generic abstract analyzer. Such analyzer will take care of dealing with the programming language in general, heap structure and so on, whilst our domains cover the operations on strings.

Another future work is the improvement of the fourth and fifth domain. In fact, we could try to modify them in order to obtain complete orders. For the fourth domain, in particular, we could also define a better widening operator (maybe taking inspiration from (Choi, Lee, Kim, & Doh, 2006)). Moreover, we could try to combine the two domains.

Finally, strings analysis can be seen as analysis of characters arrays. Thus, we could generalize our analysis in order to consider arrays of *any* base type (not only characters), combining it with existing domains which abstract the properties of such base types.

13 Bibliography

Berard, B. (1999). *Systems and Software Verification.* Springer Verlag.

Choi, T., Lee, O., Kim, H., & Doh, K. (2006). A Practical String Analyzer by the Widening Approach. *APLAS 2006* (pp. 374-388). Springer.

Christensen, A., Møller, A., & Schwartzbach, M. (2003). Precise analysis of string expressions. *Proc. 10th International Static Analysis Symposium. 2694*, pp. 1-18. Springer-Verlag.

Christodorescu, M., Kidd, N., & Goh, W. (2005). String analysis for x86 binaries. *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (pp. 88 - 95). ACM.

Clarke, E. M. (2008). The Birth of Model Checking. In *Lecture Notes in Computer Science - 25 Years of Model Checking* (pp. 1-26). Springer Berlin / Heidelberg.

Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 238 - 252). ACM.

Cousot, P., & Cousot, R. (1992, August). Abstract interpretation frameworks. *Journal of Logic and Computation, 2*(4), 511—547.

Davey, B., & Priestley, H. (2002). *Introduction to Lattices and Orders.* Cambridge University Press.

Doh, K., Kim, H., & Schmidt, D. (2009). Abstract Parsing: Static Analysis of Dynamically Generated String Output Using LR-Parsing Technology. *Proceedings of the 16th International Symposium on Static Analysis* (pp. 256 - 272). Springer-Verlag.

Earley, J. (1970, February). An efficient context-free parsing algorithm. *Communications of the ACM, 13*(2), 94 - 102.

Ferrara, P. (2009). *Static analysis via abstract interpretation of multithreaded programs.*

Gould, C., Su, Z., & Devanbu, P. (2004). Static Checking of Dynamically Generated Queries in Database Applications. *Proceedings of the 26th International Conference on Software Engineering* (pp. 645 - 654). IEEE Computer Society.

Hecht, M. S. (1977). *Flow Analysis of Computer Programs.* Elsevier Science Inc.

Hosoya, H., & Pierce, B. (2003, May). XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol., 3*(2), 117-148.

http://en.wikipedia.org. (n.d.). Retrieved from Wikipedia.

Janssens, G., & Bruynooghe, M. (1990). *Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation: Definition and Proofs.* Report CW108, K. U. Leuven, Dept. of Computer Science.

Janssens, G., & Bruynooghe, M. (1992). Deriving Description of Possible Values of Program Variables by Means of Abstract Interpretation. *Journal of Logic Programming, 13*(2-3), 205-258.

Kildall, G. (1973). A unified approach to global program optimization. *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 194 - 206). ACM.

Kong, S., Choi, W., & Yi, K. (2009). Abstract parsing for two-staged languages with concatenation. *Proceedings of the eighth international conference on Generative programming and component engineering* (pp. 109-116). ACM.

Minamide, Y. (2005). Static approximation of dynamically generated Web pages. *Proceedings of the 14th international conference on World Wide Web* (pp. 432 - 441). ACM.

Mohri, M., & Nederhof, M.-J. (2001). Regular approximation of context-free grammars through transformation. (J.-C. Junqua, & G. van Noord, Eds.) *Robustness in Language and Speech Technology*, 153-163.

Monniaux, D. (2009, June). A minimalistic look at widening operators. *Higher-Order and Symbolic Computation, 22*(2), 145 - 154.

Nielson, F., Nielson, H., & Hankin, C. (2005). *Principles of Program Analysis.* Springer Verlag.

Peled, D., Pelliccione, P., & Spoletini, P. (2009). Model Checking. In *Wiley Encyclopedia of Computer Science and Engineering.*

Pierce, B. (2002). *Types and Programming Languages.* MIT Press.

Tabuchi, N., Sumii, E., & Yonezawa, A. (2002). Regular Expression Types for Strings in a Text Processing Language. *Electr. Notes Theor. Comput. Sci., 75*.

Thiemann, P. (2005). Grammar-based analysis of string expressions. *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation* (pp. 59 - 70). ACM.

van Hentenryck, P., Cortesi, A., & Le Charlier, B. (1995). Type Analysis of Prolog Using Type Graphs. *Journal of Logic Programming, 22*(3), 179-208.

Yu, F., Alkhalaf, M., & Bultan, T. (2009). *Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses.* UCSB Technical Report.

Yu, F., Bultan, T., & Ibarra, O. (2008). Symbolic Encoding of String Lengths. *Proceedings of the 3rd Graduate Student Workshop on Computing.* UCSB.

Yu, F., Bultan, T., & Ibarra, O. (2009). Symbolic String Verification: Combining String Analysis and Size Analysis. *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009).*

Yu, F., Bultan, T., Cova, M., & Ibarra, O. (2008). Symbolic String Verification: An Automata-based Approach. *Proceedings of the 15th International SPIN Workshop on Model Checking of Software (SPIN 2008).*