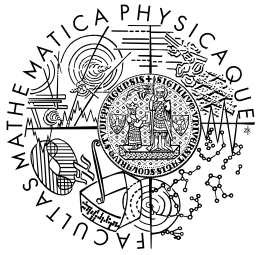

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Vlastimil Dort

String Analysis for Code Contracts

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Pavel Parízek, Ph.D.

Study programme: Computer Science

Study branch: Software Systems

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: String Analysis for Code Contracts

Author: Bc. Vlastimil Dort

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Using contracts such as method preconditions, postconditions, and class invariants in code is an approach to preventing bugs in object-oriented programs. In the .NET Framework, this is possible due to the framework of Code Contracts, which includes Clousot, a tool for static program analysis based on abstract interpretation. Although string is a fundamental type in .NET programs, Clousot does not have a usable support for analysis of string values. In this thesis, we explore the specifics of string manipulation in the C# language and in the .NET Framework, and show how they can be covered by static analysis. Our approach is to use the methods of the String class and a subset of regular expressions to specify string properties in code, and to use abstract interpretation with non-relational abstract domains to reason about those properties. We chose a small number of already published abstract domains for strings, which vary in their complexity and ability to represent different properties. We adapted those domains to our setting, including definitions of abstract semantics for the supported string methods. We implemented the abstract domains in Clousot in a way that cooperates with numerical analysis and allows adding more string abstract domains in the future.

Keywords: strings, static analysis, abstract interpretation, Code Contracts

I would like to thank RNDr. Pavel Parízek, Ph.D. for regular consultations and numerous comments and suggestions. I would also like to thank Francesco Logozzo for taking time to answer a few questions about Clousot. Finally, thanks to my parents for their support.

Contents

1 Introduction

1.1 Motivation

Presence of bugs in programs is a major problem of software engineering and programming. There are various approaches to reducing occurrences of bugs and making it easier to find them. One of those approaches is writing contracts in the code. Contracts are conditions associated with classes or methods. Method *preconditions* specify what conditions must hold when the method is called, *post-conditions* should be guaranteed by the method when the control returns from it, and *class invariants* should hold at certain points during the lifetime of an object. Listing 1.1 shows an example method with a precondition involving a parameter and a postcondition involving the return value.

Listing 1.1 Example of a method with informal contracts

```
int Abs(int x){
 Contract.Requires(x != int.MinValue); // precondition
 Contract.Ensures(Contract.Result<int>() >= 0); // postconsition
 return x < 0 ? -x : x;
}
```

This way the programmer reduces the set of possible valid program states, and a tool can be used to check that the specified contracts are not violated by the code. Reaching of an invalid program state is a programming bug, which could be caught using contracts before it propagates further. Contracts can also provide documentation about the code.

The language known for introducing contracts under the name "Design by Contract" is Eiffel. For mainstream object-oriented programming languages, approaches to specifying contracts have been proposed, such as for C++ [1] and C# [2]. Currently, solutions using libraries or external tools are possible.

Code Contracts [3] is an existing implementation of contracts for .NET languages, notably C#. It allows specifying contracts in code without any special support from the compiler, using normal method calls such as those shown in Listing 1.1. The Code Contracts project was released in January 2015 as open-source under the MIT license [4]. It implements both runtime checking and static analysis of contracts.

The runtime checking is conceptually straightforward. The contracts are checked by simply evaluating the conditions every time. The problem is a performance overhead (depending on the complexity of the contracts). Also, test inputs must be provided in order to reach the conditions, and the coverage is directly dependent on the quality of the test suite. Static analysis, on the other hand, does not impose any runtime overhead, but it can slow down the build process, and report false warnings. It is also not as easy to implement, because the analysis must consider multiple program states at once, and not just the specific ones provided.

The static checker of Code Contracts uses abstract interpretation to track possible values of variables. It has an advanced support for numeric values,

providing multiple abstract domains with varying precision and complexity [5]. However, the support for **string values** is currently very limited.

This is a significant shortcoming, because string is an important data type, present in most programming languages. Strings can be used for representing all kinds of information from short tokens to large and structured data. They are easy to understand by humans, so they have a major role in user interfaces, but they are also used in communication protocols and data storage. Bugs concerning manipulation of strings can easily be means of security threats. For example, the infamous SQL injection attacks are caused by an invalid assumption that the structure of a generated query, which is a result of string concatenation, is fixed. This works only if the part provied by a user does not contain certain special characters. That condition may not be true for strings that are a direct and unchecked input from a user.

Having a support for analysis of string values in a static checker could help prevent bugs that allow such attacks.

1.2 Goals

The goal of this thesis is to extend the Code Contracts static checker to also check properties of string values. This requires the following:

- to find interesting properties of string values that can be checked,
- to find a way to specify the properties about strings in Code Contracts,
- finding suitable string abstract domains for the static checker, which would be able to represent and verify those properties,
- to implement checking of the properties in the Code Contracts checker using the abstract domains.

To achieve the goals, it is important to understand the role of strings in .NET, the framework of abstract interpretation, and the usage and implementation of the Code Contracts static checker.

1.3 Structure of the Thesis

The second chapter contains formal definition of strings and the mathematical notation used in this thesis, introduces the basic concepts of abstract interpretation, and summarizes various aspects of strings specific to the .NET framework. The third chapter explores the possible useful string properties and the ability of current tools to check string properties. The abstract domains for strings that were considered and implemented are discussed in the fourth chapter. The fifth chapter describes the details of how the string analysis was implemented in the Code Contracts static checker. The sixth chapter shows results of experimental evaluation of the implemented features. A user guide is provided in the appendix.

2 Background

In this chapter, we introduce the necessary context, needed to implement the analysis of string values in Code Contracts. This includes a way to formally describe the static analysis using mathematical definitions and the framework of abstract interpretation. Then, the role of strings in the .NET framework is discussed, and the semantics of standard library operations is defined using the notation introduced in Section 2.1. Finally, the Code Contracts framework itself is described, focusing on its static checker, Clousot.

2.1 Mathematical Foundations

The analysis of string values can be described mathematically, using the abstract interpretation framework and mathematical definition of strings that we provide below.

2.1.1 Basic Definitions

First, we define a few basic notations used in mathematical expressions.

Definition 2.1.1. $\mathbb{N} = [0, \infty)$ is a set of *natural numbers* (non-negative integers).

Definition 2.1.2. Let X be a set, then $\mathcal{P}(X) = \{Y : Y \subseteq X\}$ is a *power set* (set of subsets) of X.

Definition 2.1.3. $\mathbb{B} = \{\mathsf{t}, \mathsf{f}\}$ is a set of *boolean* values.

To represent possible values of a boolean variable, we need a four-valued logic, which can be defined and interpreted in multiple ways: as subsets of $\mathbb{B}$, mapping from $\mathbb{B}$ to $\mathbb{B}$, or tuples of $\mathbb{B}$. When using the tuples, the first item tells whether the value can be false, and the second item whether the value can be true. To make that clear, we use the $\langle \mathsf{f}: \mathbb{B}, \mathsf{t}: \mathbb{B} \rangle$ notation.

Definition 2.1.4. The *four-valued logic* is a set $\{\bot, \mathsf{f}, \mathsf{t}, \top\}$ with the following isomorphic representations:

Meaning	Symbol	$\mathcal{P}(\mathbb{B})$	$\mathbb{B} \to \mathbb{B}$	$\mathbb{B}\mathbb{B}$	$\langle \mathsf{f}: \mathbb{B}, \mathsf{t}: \mathbb{B} \rangle$
Bottom	$\bot$	$\{\}$	$\lambda x.\mathsf{f}$	$\langle \mathsf{f}, \mathsf{f} \rangle$	$\langle \mathsf{f}: \mathsf{f}, \mathsf{t}: \mathsf{f} \rangle$
False	f	$\{\mathsf{f}\}$	$\lambda x.\neg x$	$\langle \mathsf{t}, \mathsf{f} \rangle$	$\langle \mathsf{f}: \mathsf{t}, \mathsf{t}: \mathsf{f} \rangle$
True	t	$\{\mathsf{t}\}$	$\lambda x.x$	$\langle \mathsf{f}, \mathsf{t} \rangle$	$\langle \mathsf{f}: \mathsf{f}, \mathsf{t}: \mathsf{t} \rangle$
Top	$\top$	$\{\mathsf{f}, \mathsf{t}\}$	$\lambda x.\mathsf{t}$	$\langle \mathsf{t}, \mathsf{t} \rangle$	$\langle \mathsf{f}: \mathsf{t}, \mathsf{t}: \mathsf{t} \rangle$

2.1.1.1 Strings

Now we define the notion of strings mathematically.

Definition 2.1.5. Let the *alphabet* Σ be a finite set of characters. Then a *string* $s = \langle s_0, s_1, \ldots, s_{l-1} \rangle$ is a finite sequence of characters from Σ. The *empty string* is $\epsilon = \langle\rangle$, i.e. a sequence of length 0. The *set of all strings* is Σ^*, and a *language* L is a set of strings $L \subseteq \Sigma^*$.

A basic operation on strings is concatenation.

Definition 2.1.6. Let $s = \langle s_0, s_1, \ldots, s_{l-1}\rangle$ and $t = \langle t_0, t_1, \ldots, t_{k-1}\rangle$ be strings, then $st = \langle s_0, s_1, \ldots, s_{l-1}, t_0, t_1, \ldots, t_{k-1}\rangle$ is a *concatenation* of s and t.

We define substring and indexing operators to support referring to a part of a string. The characters are indexed starting from zero.

Definition 2.1.7 (Indexing)**.** Let $s = \langle s_0, s_1, \ldots, s_{l-1}\rangle \in \Sigma^*$ be a string, then the *length* of s is $|s| = l$. The *prefix* of s of length i is $s[:i] = \langle s_0, s_1, \ldots, s_{i-1}\rangle$, the *suffix* of s starting at index i is $s[i:] = \langle s_i, s_{i+1}, \ldots, s_{l-1}\rangle$ and the *substring* of s of starting at index i, of length $j - i$, is $s[i:j] = \langle s_i, s_{i+1}, \ldots, s_{j-1}\rangle$. The *character of s at* index i is $s[i] = s_i$.

If there is a linear order $\leq$ on Σ, we define the lexicographical linear order $\leq$ on Σ^*. We assume $<$, $\geq$ and $>$ are defined in terms of $\leq$ as usual.

Definition 2.1.8. $c \leq d \Leftrightarrow \begin{cases} \mathsf{t} & c = \epsilon \vee c[0] < d[0] \\ \mathsf{f} & c \neq \epsilon \wedge (d = \epsilon \vee c[0] > d[0]) \\ c[1:] \leq d[1:] & c[0] = d[0] \end{cases}$

Now, we define a few operations on strings that will be used later in other definitions.

Definition 2.1.9 (Common parts)**.** Let $S \subseteq \Sigma^*$ be a set of strings.

Then the *length of the longest common prefix* of strings in a set S is $\mathrm{lcpl}(S) = \max\{i \colon \forall 0 \leq j < i \colon \forall s, t \in S \colon s[j] = t[j]\}$ and the *length of the longest common suffix* is $\mathrm{lcsl}(S) = \max\{i \colon \forall 0 \leq j < i \colon \forall s, t \in S \colon s[|s| - j - 1] = t[|t| - j - 1]\}$.

The *longest common prefix* of strings in S is $\mathrm{lcp}(S) = s[:\mathrm{lcpl}(S)]$ and the *longest common suffix* is $\mathrm{lcs}(S) = s[|s| - \mathrm{lcsl}(S):]$.

Definition 2.1.10 (Character substitution)**.** Let $c, f, t \in \Sigma$ be characters. Then $c[f := t] = \begin{cases} c & c \neq f \\ t & c = f \end{cases}$.

Definition 2.1.11. Let $s = \langle s_0, s_1, \ldots, s_{l-1}\rangle \in \Sigma^*$, then $s[f := t] = \langle s_0[f := t], s_2[f := t], \ldots, s_{l-1}[f := t]\rangle$ is a *substitution* of t for f in s.

Definition 2.1.12. Let $s = \langle s_0, s_1, \ldots, s_{l-1}\rangle \in \Sigma^*$, then $\mathrm{char}(s) = \{s_0, s_2, \ldots, s_{l-1}\}$ is the *set of characters in s*.

Definition 2.1.13 (Searching)**.** Let $s, t \in \Sigma^*$ be strings. Then a *set of indices* of occurrences of s in t is $\mathrm{indexset}(t, s) = \{i \colon t[i : i + |s|] = s\}$, the *index of the first occurrence* of s in t is $\mathrm{firstindex}(t, s) = \min \mathrm{indexset}(t, s)$, and the *index of the last occurrence* of t in s is $\mathrm{lastindex}(t, s) = \max \mathrm{indexset}(t, s)$.

We also define a few operators on languages.

Definition 2.1.14. Let $K, L \subseteq \Sigma^*$ be languages. Then the *concatenation of languages* K and L is $KL = \{st \colon s \in K, t \in L\}$, the *$i$-th power* of language L is $L^i = \begin{cases} \epsilon & i = 0 \\ L^{i-1}L & i > 0 \end{cases}$, and the *iteration* of L is $L^* = \bigcup_{i \in [0,\infty)} L^i$.

Definition 2.1.15. We will use $\bigcirc$ to mean a language of single characters (Σ) and $\circledast$ to mean the language of all strings (Σ^*).

Example 2.1.16. Writing $t \in \circledast s \circledast$ means that s is a substring of t. Writing $t \in p\circledast$ means that p is a prefix of t.

2.1.2 Abstract Interpretation

The approach of abstract interpretation was introduced by Cousot [6]. It can be used to analyze various properties of programs, but we are interested in a particular setting, where the concrete domain C describes the program state at some point in the program. The program state is represented by the mapping $\mathbb{V} \to V$ from variables to their values. The statements of the program determine transition functions between the program states, and can be used to construct a system of equations about the possible program states at respective program points. The exact solution of such a system is generally not computable, so we choose an abstraction of the possible program states. The abstraction is usually defined as a complete lattice of abstract elements.

Definition 2.1.17. A complete lattice $\langle D, \sqsubseteq_D, \sqcup_D, \sqcap_D, \bot_D, \top_D \rangle$ is a set D, with:

A *partial order* $\sqsubseteq_D$.

The *meet* operator $\sqcap_D$, which is the greatest lower bound.

The *join* operator $\sqcup_D$, which is the least upper bound.

The *bottom* element $\bot_D$, which is the lowest element $\bot_D = \bigsqcup_D \emptyset$, meaning *unreachable*.

The *top* element $\top_D$, which is the highest element $\top_D = \bigsqcap_D \emptyset$, meaning *all possible values*.

The *concrete domain* of possible program states is a lattice $\langle \mathcal{P}(C), \subseteq, \cup, \cap, \emptyset, C \rangle$.

The *abstract domain* is a lattice $\langle D, \sqsubseteq_D, \sqcup_D, \sqcap_D, \bot_D, \top_D \rangle$, which is connected to the concrete domain by an abstraction function $\alpha_D \colon \mathcal{P}(C) \to D$ and concretization function $\gamma_D \colon D \to \mathcal{P}(C)$. Ideally, they together form a Galois connection $\mathcal{P}(C) \underset{\alpha_D}{\overset{\gamma_D}{\leftrightarrows}} D$.

2.1.2.1 Non-relational Domains

An easy way to approximate sets of possible assignments of values variables ($\mathcal{P}(\mathbb{V} \to V)$), is considering the variables one by one (cartesian abstraction). This is represented by a mapping $\mathbb{V} \to \mathcal{P}(V)$ from variables to sets of values. That way we track possible values for individual variables, but we have no information about their relations. For example, in a program, it might hold that a string variable is always a prefix of another variable. In a non-relational domain, this information cannot be kept, so the result is that both variables can contain any value.

For variables of string type, the possible values are $V = \Sigma^*$, and the mapping $\mathbb{V} \to \mathcal{P}(\Sigma^*)$ tells us for each variable the language specifying the strings that the variable can have. To design a non-relational abstract domain for strings, we can consider Σ^* to be the concrete domain and the abstraction and concretization functions will be $\alpha_D \colon \mathcal{P}(\Sigma^*) \to D$ and $\gamma_D \colon D \to \mathcal{P}(\Sigma^*)$.

2.1.2.2 Reduced Product

An important operation on abstract domains is the reduced product. It allows us to combine the strengths of multiple abstract domains into a single abstract domain. It is similar to cartesian product but excludes unnecessary elements, which represent the same set of concrete values as some other elements.

2.1.2.3 Data-flow Analysis (fixpoint algorithm)

Data-flow analysis can be used to compute the abstraction of possible program states at all program points. It works on a control flow graph (CFG) of the program (possibly analysing one method at a time), and starts with assigning the bottom element to all program points, except the entry points. Then it iteratively increases the abstract values at the program points according to the transition relation on the edges of the CFG, until the values are stable (i.e., until the fixed point is reached).

2.1.2.4 Widening

To ensure that the fixpoint algorithm terminates, either the lattice of the abstract domain must have a finite height (or, more precisely, must not contain an infinite ascending chain). Alternatively, when joining a newly computed value with the previous value at a program point, instead of using the fully precise join operator $\sqcup_D$, we can use a less precise *widening* operator ∇_D instead. The widening operator overapproximates the result of the join operator to avoid infinite increasing sequences in the data-flow analysis. Using widening, termination of the algorithm is guaranteed even for lattices of infinite height.

2.1.2.5 Soundness, Completeness, Overapproximation

Soundness and completeness are desirable properties of the static analysis. In our setting, soundness of the static analysis means that the computed abstraction covers all possible program states and does not miss any state, so when the analysis says that something cannot happen, we can be sure about that. But a sound analysis can *overapproximate* the result, which means that the result can represent more states, including some that are not really reachable. This results in reporting *false positives*, warning about bugs that are not in the program. We want to avoid false positives, so the ultimate overapproximation of computing $\top$ for everything is sound but not useful.

On the other hand, completeness would mean that the analysis does not provide answer that cannot really occur during the program execution. The analysis cannot be generally complete and sound.

The abstraction function α_D alone introduces overapproximation by mapping multiple sets of concrete values to a single abstract value. Further overapproximation occurs when using the widening operator. Moreover, the soundess of the analysis is not violated by overapproximating at any other point. This is important for definitions of the abstract semantics of operations (used in the transition function), where the precise specification would be too complex. Instead, we can define the abstract semantics of the operation as an upper bound on the abstract result. We can imagine this as analysing a modified program that non-deterministically introduces additional program states at some transitions.

2.2 Strings in C# and .NET

Strings are supported by almost all programming languages and platforms. However, there can be significant differences in how they are implemented and what

Listing 2.1 Usage of various types of strings in C#

```
// Using string
string s = "a" + "b";
s = s.Replace("a", "aa");
// Using char[]
char c = new char[]{'s', 't', 'r'};
c[2] = 'd';
// Using StringBuilder
StringBuilder sb = new StringBuilder();
sb.AppendLine("OK");
string s = sb.ToString();
// Using IEnumerable<char>
IEnumerable<char> f = "user@domain.com";
IEnumerable<char> z = from c in f
 where c>='a' && c<='z' select char.ToUpper(c);
string value = string.Concat(z);
// Using char*
fixed char* src = "string"
char* dst = stackalloc char[16];
while(*src) *dst++ = *src++;
// Using SecureString
using (SecureString ss = new SecureString()){
 ss.AppendChar('A');
 ss.InsertAt(0, 'B');
 using (Process.Start(file, s, ss, domain)){
  //...
 }
}
```

relevant features are available in the language or standard library. The analysis tools must reflect that. Therefore, in this section, we focus on specific details of working with strings in the .NET Framework and C#.

2.2.1 Representations of Strings

The .NET Framework provides multiple types capable of representing strings, where each type defines different efficient or easy-to-use operations, provides different guarantees to the programmer, or allows interoperability with other systems. Examples of multiple ways of handling strings are shown in Listing 2.1.

The `string` type. The basic representation of strings in C# is the type `string` (`System.String`) [7]. It is an immutable sequence of `char` values, which are UTF-16 code units. It can even contain invalid UTF-16 sequences and the character `'\0'`, which is used as a terminator in some programming languages. Because `string` is a reference type, variable of type `string` can have the `null` value, which is a different value than an empty string (`""`).

Being an immutable reference type, `string` is very efficient for passing strings around. However, immutability also means that operations such as concatenation or truncation cannot mutate the string in place, and therefore a new instance must be created every time. If a lot of string modifications is performed in a program,

using a mutable representation of the strings might be more appropriate.

The `char[]` type. The simplest type of mutable strings is an array of characters, `char[]`. It can represent exactly the same sequences of `char`s as `string`. Once an array is allocated, its length cannot be changed. However, the contents (individual characters) can be changed.

The `StringBuilder` type. A fully mutable string representation is provided by the `System.Text.StringBuilder` class [8]. It is particularly useful for generating a string by multiple successive concatenations: first, we create an empty `StringBuilder`, then append the individual parts of the generated string, and at the end we convert to it to an immutable `string` value.

The `IEnumerable<char>` type. The `IEnumerable<char>` interface is a common interface implemented both by `string` and `char[]`. The only operation it allows is a forward enumeration of the characters. However, using extension methods and LINQ operators, this type can be used to do transformations of strings. The interface can also be implemented by user types. This allows `IEnumerable<char>` to also represent possibly infinite or non-deterministic sequences.

Other types. There are other types capable of representing strings, mainly used for interoperability. The `char*` type, known from C, also exists in C#. It is a pointer type, which can only be used in unsafe code. The `byte[]` array can be used when communicating through byte streams, using the `Encoding` class for conversion from UTF-16 to a specific encoding. `System.Security.SecureString` can be used if the string should be deleted from memory for security purposes, instead of leaving that to garbage collector.

2.2.2 Unicode Characters

Unicode is a character set used on modern computer systems. It supports multiple encodings, but in .NET, UTF-16 [9, Chapter 3] is used for all in-memory strings. Each `char` value is a single code unit of UTF-16.

The `string` is a sequence of `char`s (*code units*). However, two successive code units can form a surrogate pair, representing a single *code point* (which represents a Unicode character). Therefore, we must not confuse the Unicode characters and characters (`char`s) in C#.

In the .NET framework, indexing of characters in a string always works with `char`s (code units), not code points. There are some methods that do work with code points, for example the static methods of `char` have overloaded versions with a string and an index as parameters, so that the character or a surrogate pair at that index in the string can be examined. There is also an API for indexing *text elements* (`System.Globalization.StringInfo`). This is a higher-level concept which in addition considers sequences of combining characters to be a single element.

In this thesis, we will treat all strings as sequences of UTF-16 code units, implicitly assuming the alphabet Σ_{u16}.

Definition 2.2.1. The set of *UTF-16 code units* is $\Sigma_{\text{u16}} = \{\sharp 0000 \ldots \sharp ffff\}$.

The code units are written as four-digit hexadecimal numbers. When referring to characters from the ASCII range, we will use the character directly, written in a non-proportional font. For example, $\texttt{a} = \sharp 0061$.

2.2.2.1 Categories, Blocks and Scripts

The Unicode characters are divided according to various criteria [9]. Unicode *categories* roughly describe how the character is used, for example Upper-case Letter or Math Symbol. Each category has a two letter acronym. This information is available in .NET using the `UnicodeCategory` enumeration [8]. There is a fixed list of categories, but the assigned category for a character can change [10]. The assignment of characters is not absolutely stable and depends on the version of Unicode that is implemented in the used runtime [11]. In particular, in future versions, the unassigned characters will change to assigned.

Unicode *blocks* are based on ranges of code points. For example, the block containing ASCII characters (most of C# code is ASCII characters) is called "Basic Latin". Although the blocks often contain characters from the same writing system, more precise information is provided by the Script property [12].

2.2.3 Language Constructs

The `string` type plays a special role in several language constructs of the C# programming language.

Literals. C# supports writing literals of type `string` in code, using the common syntax in double quotes with backslash escape sequences. String constants can be stored in `const` fields. String values can be used as the parameters of attributes. However, it has been shown that the encoding used for attributes in .NET cannot represent all possible values that the `string` type can hold [13].

C# identifiers. The source code of C# programs is also a string (but not available in the program itself). However, identifiers used in the source code can appear in the program at runtime as string values using features such as `nameof`, reflection, dynamic calls, or caller attribute, as demonstrated in Listing 2.2.

Switch. The `switch` statement supports switching on integral types and `string`. While integral types have a direct support for the switch statement in the CIL (`switch` instruction), for strings it can be implemented by a sequence of `Equals` comparisons (as shown in Listing 2.3). This is efficient for a small number of cases. For a larger number of cases, the compiler will use a different implementation, for example the compiler for C# 5.0 converts the string to an integer index by a lookup in a `Dictionary`, and then uses the `switch` instruction. The Roslyn compiler uses a hash-based decision tree. Simplified implementations are shown in Listing 2.4. The generated code is much more complicated than a chain of comparisons. It is also dependent on the particular compiler used, so it would be difficult to recognize and analyze this pattern in tools.

Listing 2.2 Using identifier names as strings

```
void Identifiers(string parameter){
 string s1 = nameof(parameter); // s1 = "parameter"
 string s2 = typeof(Dynamic).Name; // s2 = "Dynamic"
 dynamic d = new Dynamic();
 string s3 = d.Property; // s3 = "Property"
 string s4 = Caller(); // s4 = "Identifiers"
}
string Caller([CallerMemberName]string callerName=null){
 return callerName; //The name of the calling method
}
class Dynamic : DynamicObject{
 public override bool TryGetMember
  (GetMemberBinder bnd, out object value){
  value = bnd.Name;
  return true;
 }
}
```

Listing 2.3 Switch statement and equivalent code

```
string Method(string arg){
 switch(arg){
  case "A":
   return A();
  case "B":
   return B();
  case "C":
   return C();
  case null:
   return Null();
  default:
   return Default();
 }
}
```

```
string Method(string arg){
 if(arg == null)
  return Null();
 else if(arg.Equals("A"))
  return A();
 else if(arg.Equals("B"))
  return B();
 else if(arg.Equals("C"))
  return C();
 else
  return Default();
}
```

Listing 2.4 Alternative implementations of switch

```
Dictionary<string, int> map =
 new Dictionary<string, int>{
  {"A", 0}, {"B", 1}, {"C", 2},
 };
int index;
if(arg == null)
 return Null();
else if(
 map.TryGetValue(arg, out index)
){
 switch(value){
  case 0:
   return A();
  case 1:
   return B();
  case 2:
   return C();
 }
}
return Default();
```

```
uint hash = ComputeHash(arg);
if (num <= 3289118412u){
 if (num != 0u){
  if (num == 3289118412u)
   if (arg == "A")
    return A;
 }
 else if (arg == null)
  return Null();
}
else{
 if (num != 3322673650u) {
  if (num == 3339451269u)
   if (arg == "B")
    return B;
 }
 else if (arg == "C")
  return C;
}
return Default();
```

Listing 2.5 String interpolation and equivalent code

String interpolation:

```
string x = "A", y = "B";
string z = $"From␣{x}␣to␣{y}";
```

Equivalent code:

```
string arg = "A", arg2 = "B";
string text = string.Format("From␣{0}␣to␣{1}", arg, arg2);
```

String interpolation. String interpolation is a new feature of C# 6 [14]. It allows to use a concise syntax instead of explicitly writing `string.Format`, as shown in Listing 2.5. The generated IL code for the two examples is the same.

2.2.4 Operations

In this section, we will look at operations on string types provided by the C# language and the .NET Framework Class Library. The operations can be used to query information about strings and to create new string values.

2.2.4.1 Culture-specific Operation

Because strings are used to communicate with users, and users can come from different cultures, some of the operations may have different semantics in different cultures, for example string comparison. Other operations are independent of the user's culture, for example `Length`. A lot of operations by default perform culture specific behavior according to the user's current culture. This can sometimes lead to unexpected results [15].

We must distinguish between three kinds of operatins: ordinal operations, culture-invariant operations, and culture-specific operations. Ordinal operations

work directly with code unit values. This makes static analysis of them feasible. Culture invariant operations do not allow modification of their behavior by culture, but their exact definition can still be too complex. The culture-specific operations are out of scope of static analysis.

The methods in the .NET Framework Class Library handle this situation in different ways:

- The operation is only culture-invariant or ordinal (for example `Substring`, `Concat`).
- The method accepts an argument of enumeration type `StringComparison`. This argument specifies, whether the operation will be ordinal, invariant or use the current culture, and specifies case sensitivity. The argument can be `StringComparison.Ordinal`.
- The operation accepts a boolean flag saying whether it should be case sensitive, and an argument of type `CultureInfo`. This can be `CultureInfo.InvariantCulture`.
- The method is culture specific and there is a variant with a different name that is culture invariant or ordinal (for example `ToLower`, `ToLowerInvariant`).
- The method takes `CultureInfo` and `CompareOptions` as arguments. The second argument allows optionally ignoring more properties of characters for the purpose of sorting. It can be `CompareOptions.Ordinal`.

2.2.4.2 Handling of `null` Values

As mentioned in Section 2.2.1, `string` is a reference type, and therefore all variables of type `string` can have the `null` value. The `null` value is handled in these ways by operations:

- In the case of instance methods, if the value on the left side of the dot operator is `null`, `NullReferenceException` is always thrown even before the method is called.
- A `null` argument is treated as an empty string by concatenation operations.
- In comparison operations, a `null` argument is treated as a special value, which compares lower than all other values.
- A `null` argument causes an exception (`ArgumentNullException` is thrown) in most other operations.

Some operations are implemented as static methods to accept `null` as the first argument. String operations rarely return `null` value. One example is `IsInterned`.

2.2.4.3 Methods of the `string` Class

The `string` type provides a wide range of operations [8]. In this section, we describe them and provide formal definitions for some of the operations.

Concatenation. The `Concat` method is provided with multiple overloaded versions, based on the number of arguments. If there are too many arguments, an array is created to pass the strings to the method. If any of the strings is `null`, it is treated as an empty string. `Concat` is also used to implement the operator `+` on strings [7]. The compiler might optimize concatenations of string constants and `null` values [16].

Definition 2.2.2. We will model the `Concat` method by the function $\mathsf{Concat}_{\mathcal{S}}$, which is defined as $\mathsf{Concat}_{\mathcal{S}}(s,t) = st$.

In Definition 2.2.2, the $\mathcal{S}$ subscript of the Concat function indicates that the arguments of the function are concrete strings.

A string can be also inserted at a specific index into another string by `Insert`. In this case, a `null` argument causes an exception.

Definition 2.2.3. We model the `Insert` method by the expression $\mathsf{Insert}_{\mathcal{S}}(s,i,t) = s[:i]ts[i:]$.

Comparison. To compare strings and other types in .NET, we can use two distinct comparisons - equality comparison (supported by implementors of `IEquatable`) and ordering (supported by implementors of `IComparable`). The `string` class implements both of those interfaces. The equality comparison is implemented by the `Equals` method with various overloaded versions. There is a static two-parameter version, which allows treating `null` values as a distinct value, without throwing exceptions. The same logic is also used for the `==` operator [7]. Additionally, a comparison mode can be specified. It is `Ordinal` by default.

Definition 2.2.4. We model the `Equals` method with `Ordinal` comparison on non-`null` values by $\mathsf{Equals}_{S}(s,t) \Leftrightarrow s = t$.

Ordered comparison is done by the static method `Compare` and the interface method `CompareTo`, with various overloaded versions to specify the case, culture, and comparison mode. There are also overloaded versions accepting the index and length of the substrings that should be compared. There is also an ordinal variant called `CompareOrdinal`.

Definition 2.2.5. We model the `CompareOrdinal` method on non-`null` values by $\mathsf{Compare}_{S}(s,t) = \begin{cases} (-\infty,-1] & s < t \\ 0 & s = t \\ [1,\infty) & s > t \end{cases}$.

The exact value of the returned number is not specified, and might depend on the implementation. For example, it can be the difference between the first differing pair of characters, or the difference of the lengths of the strings.

Additionally, on reference types including `string`, programmers can also use a third type of comparison, which is reference equality. The reference equality always implies value equality. For interned strings, this holds also in the other direction.

Containment. The `Contains` methods uses ordinal comparison to check whether a string contains another string.

Definition 2.2.6. We model `Contains` by $\mathsf{Contains}_{\mathcal{S}}(t, a) \Leftrightarrow t \in \circledast a \circledast$.

The `StartsWith` and `EndsWith` methods limit the occurrence of the contained substring to the start (prefix) or end (suffix). They are by default culture sensitive.

Definition 2.2.7. We model `StartsWith` with ordinal comparison by $\mathsf{StartsWith}_{\mathcal{S}}(s, t) \Leftrightarrow s \in t\circledast$, and `EndsWith` by $\mathsf{EndsWith}_{\mathcal{S}}(s, t) \Leftrightarrow s \in \circledast t$.

Searching. Searching a string returns a position of a substring within the string. `IndexOf`(`string`) is by default a culture-sensitive operation, but `IndexOf`(`char`) uses ordinal comparison. It returns the index of the first occurrence, or -1 if there is no occurrence.

Definition 2.2.8. We model `IndexOf` with ordinal comparison by $\mathsf{IndexOf}_{\mathcal{S}}(s, t) = \begin{cases} \text{firstindex}(s, t) & s \in \circledast t \circledast \\ -1 & s \notin \circledast t \circledast \end{cases}$.

`LastIndexOf` is similar to `IndexOf`, but returns the index of the last occurrence. When searching an empty string in a non-empty string, it returns the index of the last character.

Definition 2.2.9. We model the `LastIndexOf` method with ordinal comparison by $\mathsf{LastIndexOf}_{\mathcal{S}}(s, t) = \begin{cases} \text{lastindex}(s, t) & s \in \circledast t \circledast \wedge t \neq \epsilon \\ -1 & s \notin \circledast t \circledast \\ \max(0, |s| - 1) & t = \epsilon \end{cases}$.

Formatting. The `Format` method is used to generate a new string from a format string and values of other objects. If the format string is constant, it can be modeled by concatenations of the constant parts with the variable parts.

Copying and Conversion. There are several methods which do not change the represented value. Both `ToString` and `Clone` return the same instance (`this`). `Copy` creates a new instance with the same value. `ToCharArray` converts a string to `char[]`, while `string` has a constructor that constructs a string from `char[]`.

Interning. Strings in .NET can be interned, to ensure that the same instance is used to represent the same string value. This influences reference equality, but has no effect when we are only interested int the string value. The `Intern` method returns an interned instance of the same value. The `IsInterned` returns `null` if the string is not interned.

Normalization. Unicode defines normalization forms in order to make a single representation for different character sequences that should be handled equivalently. Normalization can be achieved and checked using the `Normalize` and `IsNormalized` methods. Supporting this feature would require the analysis to know the exact definitions of normalization.

Emptiness. We can check whether a string is empty by comparing `Length == 0`, but the static `IsNullOrEmpty` combines this with a `null`-check.

Definition 2.2.10. We model the `IsNullOrEmpty` method on non-`null` values by $\mathsf{IsEmpty}_{\mathcal{S}}(s) \Leftrightarrow s = \epsilon$.

The `IsNullOrWhitespace` method also ignores white-space characters, but it is dependent on what characters are currently considered to be white-space (Unicode categories).

Array Delimiters. We can convert between an array and delimited strings by the `Join` and `Split` methods. Arrays of strings are out of scope of this thesis.

Length. The `Length` property returns the number of characters.

Definition 2.2.11. We model the `Length` property by $\mathsf{Length}_{\mathcal{S}}(s) = |s|$.

Padding. Padding extends the string to have at least the specified length, using a padding character. If character is not specified, a space character (♯0020) is used.

Definition 2.2.12. We model the `PadLeft` method by $\mathsf{PadLeft}_{\mathcal{S}}(s, n, c) = \begin{cases} c^{n-|s|}s & |s| < n \\ s & |s| \geq n \end{cases}$ and the `PadRight` method by $\mathsf{PadRight}_{\mathcal{S}}(s, n, c) = \begin{cases} sc^{n-|s|} & |s| < n \\ s & |s| \geq n \end{cases}$.

Replace. The `Replace` method has two overloaded versions: one replaces a single character by another character, the other version replaces occurrences of a string by another string. For the string replacement, it is significant that the string is searched from the beginning, and if an occurrence is found, the search continues after the end of the occurrence. It does not replace overlapping occurrences or new occurrences arising by the preceding replacements. The result string therefore can still contain occurrences of the old string. Replacing an empty string is not allowed.

Definition 2.2.13. We model the `Replace` methods by $\mathsf{ReplaceC}_{\mathcal{S}}(s, c, d) = s[c := d]$ and

$$\mathsf{ReplaceS}_{S}(s, t, u) = \begin{cases} s[: i]u\mathsf{ReplaceS}_{S}(s[i + |t| :], t, u) & i = \text{firstindex}(s, t) \\ s & s \notin \circledast t \circledast \\ \bot & t = \epsilon \end{cases}.$$

Substring. The `Substring` methods extract a part of the strings based on an index from the start. The length is optional.

Definition 2.2.14. We model `Substring` by $\mathsf{Substring}_{S}(s, i, l) = s[i : i + l]$ and $\mathsf{SubstringEnd}_{\mathcal{S}}(s, i) = s[i :]$.

The `Remove` method is similar, but removes the specified part and keeps the rest.

Definition 2.2.15. We model `Remove` by $\mathsf{Remove}_{\mathcal{S}}(s, i, l) = s[:i]s[i+l:]$ and $\mathsf{RemoveEnd}_{\mathcal{S}}(s, i) = s[:i]\ i < |s|$.

Case conversion. The `ToLower` and `ToUpper` methods replace characters by their counterparts of the selected case. This is a culture-dependent operation, however, there are methods `ToLowerInvariant` and `ToUpperInvariant` that are culture-invariant. However, the result is still dependent on the specific definition of character case pairs.

Trimming. The `TrimStart`, `TrimEnd`, and `Trim` methods remove characters belonging to a specific set from the start, the end, or both ends, of a string. If called with no argument or a `null` argument, or if an empty array is provided, then white-space characters are trimmed.

Definition 2.2.16. We model the `TrimStart` method with a non-empty second argument by

$$\mathsf{TrimStart}_{\mathcal{S}}(s, d) = \begin{cases} s[i:] & i = \min\{k \colon s[k] \notin \mathrm{char}(d)\} \\ \epsilon & \mathrm{char}(s) \subseteq \mathrm{char}(d) \end{cases},$$

the `TrimEnd` method by

$$\mathsf{TrimEnd}_{\mathcal{S}}(s, d) = \begin{cases} s[:j] & j = \max\{k \colon s[k] \notin \mathrm{char}(d)\} \\ \epsilon & \mathrm{char}(s) \subseteq \mathrm{char}(d) \end{cases},$$

and the `Trim` method by

$$\mathsf{Trim}_{\mathcal{S}}(s, d) = \begin{cases} s[i:j+1] & I = \{k \colon s[k] \notin \mathrm{char}(d)\} \wedge i = \min I \wedge j = \max I \\ \epsilon & \mathrm{char}(s) \subseteq \mathrm{char}(d) \end{cases}.$$

2.2.4.4 Methods of the `StringBuilder` Class

`StringBuilder` provides different operations than `string`. The methods of the `StringBuilder` class usually mutate the object-in place and return `this`.

Appending. The `Append` method is a counterpart of the `Concat` method on `string`, but changing the `StringBuilder` instance in place. It can therefore be modeled by the same function $\mathsf{Concat}_{\mathcal{S}}(s, t)$.

The `AppendLine` variant adds a new-line character which is `"\n"` $\langle \sharp 000a \rangle$ or `"\r\n"` $\langle \sharp 000d, \sharp 000a \rangle$ depending on the `Environment.NewLine` property.

Length and capacity. The `Length` property works the same as for `string` and can be modeled by $\mathsf{Length}_{\mathcal{S}}$. `StringBuilder` also has a `Capacity` property, which corresponds to the amount of allocated memory. Additionally, it allows to specify a maximum capacity, however, this limit is not reliable.

Conversion. The constructor can be used to initialize `StringBuilder` with an existing `string` value. The standard `ToString` method converts the value of `StringBuilder` to `string`.

2.2.4.5 Operations with the Character Arrays

Character arrays (`char[]`) support getting length (`Length` property), and getting or setting a character at an index using the `[]` operator.

2.3 Regular Expressions

Regular expressions are a flexible and popular way of manipulating strings in many programming languages. They can be used for searching and replacing a pattern in a string. Although they are called *regular expressions*, the class of languages they can recognize are not limited to the mathematical definition of regular languages[1].

2.3.1 Regex in .NET

Regular expressions are supported in the .NET Framework by the `Regex` class. The regular expression is specified by a string argument to a constructor or a static method. The syntax of the regular expression language is not standardized and the .NET Framework uses its own definition [17]. Furthermore, the semantics of the regular expression string can be altered by using a `RegexOptions` flags argument to the methods or constructors. Some of the options can be also changed inside the regular expression, so that they apply only to a part of it.

A notable diferrence from the current Unicode recommendation [18] is that regex matching in .NET operates on UTF-16 code units. Character sets might be named by Unicode category or a name corresponding to a `char.Is*` method.

2.3.2 Specifying a Language

In this thesis we will consider only one method of the `Regex` class – the `IsMatch` method. It determines, whether a specified string contains a match of the specified regular expression. The regular expression therefore represents a language, and can be used in contracts to check that a string value belongs to that language.

An essential feature of regular expressions are anchors. Without using anchors, `IsMatch` would return `true` if any substring matches the expression, so it would be only possible to express languages of the form $\circledast L \circledast$. To check a property of the whole string, the begin and end anchors must be used. `"^"` is the anchor for the beginning a string. The `"$"` anchor is for the end of a string, but if the last character is end-of-line (`'\n'`), it matches before that character too. To match only the end of the string, `"\\z"` must be used. The meaning of the anchors can be also changed by the `RegexOptions.Multiline` flag.

2.3.3 Formal Definitions

To allow formally representing regular expressions including anchors, we define a subset of regular expressions in terms of mappings from strings to sets of matches.

[1] For example, the context free language $\{\texttt{a}^n\texttt{ba}^n : n \in \mathbb{N}\}$ is recognized by regex `"^(?<a>a*)b\\k<a>$"`, and the context-sensitive language $\{\texttt{a}^n\texttt{b}^n\texttt{c}^n : n \in \mathbb{N}\}$ is recognized by regex `"^(?<a>a)*(?<b-a>b)*(?<-b>c)*(?(a)(?!))(?(b)(?!))$"`.

Each match is represented by two indices (begin and end of the interval of the match).

Definition 2.3.1. A *matcher* M is a function from a string to a set of intervals within the string, $M\colon \Sigma^* \to \mathcal{P}(\mathbb{N}^2)$, where $\forall \langle i, j\rangle \in M(s)\colon 0 \le i \le j \le |s|$.

We build a matcher for a regular expression using a few matcher constructors, which roughly correspond to the AST of the regex string. The first parenthesis contains constructor aguments, the second parenthesis means application of the matcher to a string value.

Definition 2.3.2. The *empty string matcher* is $\mathsf{empty}()(s) = \{\langle i, i\rangle : 0 \le i \le |s|\}$. The *begin* and *end anchor matchers* are $\mathsf{begin}()(s) = \{\langle 0, 0\rangle\}$ and $\mathsf{end}()(s) = \{\langle |s|, |s|\rangle\}$. The *matcher for a single character* from a set of characters $S \subseteq \Sigma$ is $\mathsf{single}(S)(s) = \{\langle i, i+1\rangle : s[i] \in S\}$.

Definition 2.3.3. Let $M_1, \ldots, M_n$ be matchers, then a *concatenation matcher* is $\mathsf{concat}(M_1, \ldots, M_n)(s) = \{\langle i_0, i_n\rangle : \exists i_1, \ldots, i_{n-1}\colon \forall 1 \le k \le n\colon \langle i_{k-1}, i_k\rangle \in M_k(s)\}$ and a *union matcher* is $\mathsf{union}(M_1, \ldots, M_n)(s) = \bigcup_{i=1}^{n} M_i(s)$.

Definition 2.3.4. Let M be a matcher, f be the lower bound on the number of repetitions, and t be the upper bound. Then a *bounded loop matcher* is $\mathsf{loop}(M, f, t)(s) = \{\langle i_0, j\rangle : \exists n \in [f, t] : \exists i_1, \ldots, i_n\colon j = i_n \wedge \langle i_k, i_{k+1}\rangle \in M(s)\}$, and an *unbounded loop matcher* is $\mathsf{loop}(M, f, \infty)(s) = \{\langle i_0, j\rangle : \exists n \in [f, \infty) : \exists i_1, \ldots, i_n\colon j = i_n \wedge \langle i_k, i_{k+1}\rangle \in M(s)\}$.

When using the bounded loop matcher constructor, we will only consider cases where $t \ge 2$. Otherwise, we will use the following equivalent representations: $\mathsf{loop}(M, 0, 0) = \mathsf{empty}()$, $\mathsf{loop}(M, 0, 1) = \mathsf{union}(M, \mathsf{empty}())$, and $\mathsf{loop}(M, 1, 1) = M$. The following matchers are also defined in terms of matchers defined before.

Definition 2.3.5. The *non-matcher* is $\mathsf{none}()(s) = \mathsf{single}(\emptyset) = \emptyset$, the *wildcard matcher* is $\mathsf{wildcard}() = \mathsf{single}(\Sigma)$, and a *maybe matcher* is $\mathsf{maybe}(M)(s) = \mathsf{union}(M, \mathsf{empty}()) = M(s) \cup \mathsf{empty}()(s)$

2.3.4 Parsing Regular Expression Strings

In order to construct the matcher, the regular expression string must be parsed. We are only concerned about regular expression strings that are compile-time constants. Furthermore, some features are too complex to be modeled by matchers, such as backreferences. We will not support expressions containing them. Since we do not need to distinguish multiple matches, lazy loops such as `"x*?"` can be treated the same as normal loops.

Definition 2.3.6. The matcher for a regular expression r is $\mathrm{regex}(r)$. The syntax elements are transformed as in Table 2.1. The expression in the left column might contain subexpressions `e` and `f`, which are recursively parsed to matchers M and N.

Regex	Matcher
`e|f`	$\mathsf{union}(M, N)$
`e{m,n}` or `e{m,n}?`	$\mathsf{loop}(M, m, n)$
`e*` or `e*?`	$\mathsf{loop}(M, 0, \infty)$
`e+` or `e+?`	$\mathsf{loop}(M, 1, \infty)$
`ef`	$\mathsf{concat}(M, N)$
`e?`	$\mathsf{maybe}(M)$
`a`	$\mathsf{single}(\{\mathtt{a}\})$
`[abc]`	$\mathsf{single}(\{\mathtt{a}, \mathtt{b}, \mathtt{c}\})$
`^`	$\mathsf{begin}()$
`\z`	$\mathsf{end}()$
`$`	$\mathsf{concat}(\mathsf{maybe}(\mathsf{single}(\sharp 000\mathtt{a})), \mathsf{end}())$
`.`	$\mathsf{wildcard}()$
`(?!)`	$\mathsf{none}()$
`(?:e)`	M

Table 2.1: Correspondence between regular expression strings and matcher constructors

Example 2.3.7. The matcher for the regular expression `"^(?:a|bc*)$"` is $\mathsf{regex}(\mathtt{\text{"}\hat{}\mathtt{(?:a|bc*)\$\text{"}}) = \mathsf{concat}(\mathsf{begin}(), \mathsf{union}(\mathsf{single}(\mathtt{a}), \mathsf{concat}(\mathsf{single}(\mathtt{b}), \mathsf{loop}(\mathsf{single}(\mathtt{c}), 0, \infty))), \mathsf{maybe}(\mathsf{single}(\sharp 000\mathtt{a})), \mathsf{end}())$.

Definition 2.3.8. We model the `Regex.IsMatch` method by $\mathsf{IsMatch}_{\mathcal{S}}(s, r) \Leftrightarrow \mathsf{regex}(r)(s) \neq \emptyset$.

2.4 Code Contracts

Code Contracts is a language-independent implementation of contracts for .NET [5, 19, 3], implemented in C#. Currently, the source code is hosted on Github [4]. This website is also used for issue tracking. The recent development focuses on adaptation for VisualStudio 2015 and the Roslyn Compiler.

2.4.1 Writing Contracts in Code

Using Code Contracts means writing assertions in code, using special classes and methods that can be recognized by the Code Contract tools. The contracts in the source look like a normal code, but do not work without the provided tools. The contract classes are part of the .NET Framework Class Library and do not require using external libraries [19].

The most important class is `System.Diagnostics.Contracts.Contract`. It provides static methods that are used to specify contract assertions. The `Assert` and `Assume` methods can be used anywhere in the code and their argument is a boolean condition that must hold at that point. The condition can be any expression that calls only methods marked by the `[Pure]` attribute.

The static checker tool, Clousot, tries to prove that the `Assert` conditions are guaranteed to hold, assuming the `Assume` conditions hold. The `Requires`

Listing 2.6 Example of class with contracts

```
using System.Diagnostics.Contracts;
class WithContracts{
 private int x;
 [ContractInvariantMethod]
 public void Invariants(){
  Contract.Invariant(x >= 0);
 }
 public int AddValue(int a){
  Contract.Requires(a > 0);
  Contract.Ensures(Contract.Result<int>() > 0);
  Contract.Ensures(x > Contract.OldValue(x));
  x += a;
  return x;
 }
 public void AddArray(int[] arr){
  Contract.Requires(arr != null);
  for(int i = 0; i < arr.Length && arr[i] > 0; ++i){
   x += arr[i];
  }
 }
}
```

and `Ensures` clauses can be used only at the beginning of a method. They are the preconditions and postconditions of the method. For the purpose of static analysis, they are transformed into asserts and assumes (depending whether the body of the method or a method calling it is analyzed – for example, in the method body, the preconditions are assumed and postconditions asserted [5]).

The Listing 2.6 shows an example of a class with postconditions, preconditions and an invariant method.

2.4.2 Runtime and Static Checking

The Code Contracts framework consists of multiple tools.

- Runtime checking is provided by Foxtrot (or `ccrewrite.exe`), which rewrites the compiled assembly to contain code checking the contracts at runtime.
- Static checking is provided by Clousot (or `cccheck.exe`). It implements static analysis based on abstract interpretation. It tries to prove that the specified contracts are not violated. The analysis is not complete, so it might report false positives (unproven assertions). Its strength depends on the selected abstract domains. However, the analysis is generally not sound either, due to an optimistic abstraction of heap [5]. That means false negatives might also occur.
- `ccdoc.exe` exports contracts to documentation.
- There is also an extension for Visual Studio, which allows configuring the checkers in a graphical user interface form and running them automatically on build.

2.4.3 Overview of Clousot Internals

The concepts used in Clousot were described by Fähndrich and Logozzo [5]. We will introduce specific parts of implementation found in the source code [4], that are relevant to implementing a new value analysis.

The executable project is `Executables/Clousot`[2], but the main code of the program is in the project `ParticularAnalysis/ClousotMain`[3].

The static checker works on a compiled assembly. It decodes the CIL, and constructs a control-flow graph. All invariants, preconditions and postconditions are changed to assert and assume instructions. Assume instructions are also implicit after conditionals. Clousot also tries to decode some expressions and makes an abstraction of the heap (`OptimisticHeapAbstraction`).

Clousot contains multiple analyzers, which are grouped as nested classes of the `Analyzers` class in `ParticularAnalysis/Analyzers`[4]. The analyzers are found by reflection. An analyzer may specify options, which are by reflection filled from arguments on the command line. The analyzer calls the corresponding analysis, which usually runs the data-flow algorithm (`ForwardAnalysisSolver` in `AnalysisInfrastructure/CodeAnalysis/DFA.cs`[5]) on the decoded program with symbolic variables, using a selected abstract domain, which implements `IAbstractDomain` (from `AnalysisInfrastructure/Abstract Domains`[6]). This interface includes the standard operators of abstract domains, such as `Bottom`, `Top`, `Join`, `Meet` and `Widening`. The analysis also handles the transfer function using the visitor pattern, where each instruction of the IL has an associated handler method. For example, the `Call` method handles transfer function for method calls and `Newobj` handles constructor calls and `Assume` handles assume statements. After the DFA reaches the fixpoint, the analysis is queried whether it could prove various facts using `IFactQuery`.

Some of the classes are nested inside the `TypeBindings` class, so that the generic parameters do not have to be repeated for each of them.

[2] `Executables/Clousot` is the path within the `CodeContracts.sln` Visual Studio solution. The path on the file system is `Microsoft.Research\Clousot`.

[3] `Microsoft.Research\Clousot` on the file system.

[4] `Microsoft.Research\Analyzers` on the file system.

[5] `Microsoft.Research\CodeAnalysis` on the file system.

[6] `Microsoft.Research\AbstractInterpretation\Abstract Domains` on the file system.

2.4.3.1 Using Multiple Analyses

In the basic setting, the analyses are executed independently. However, this can mean unnecessary loss of information, because two analyses can be more precise if they have access to the information from the other one. The approach used in Clousot is to allow an analysis to accept "plug-ins", which will be called at appropriate places (for example from the IL visitor methods), and allow the plug-ins to increase precision of analysis results. The abstract domain is a reduced product of domains of the individual analyses.

The array analysis (`ArrayAnalysis` in `ParticularAnalysis/Analyzers/Array Analysis/ComposedAnalysisWithArrays.cs`) is implemented this way. When the array analysis is enabled on the command line, the other analyses are no longer executed independently, but work as plug-ins (`GenericPlugInAnalysisForComposedAnalysis`) to the array analysis. The plug-in can be just a wrapper calling the original independent analysis.

3 Analysis

The analysis of string values should ideally be able to capture interesting properties of strings. In this chapter, we list a few of such properties. Then we look at existing solutions for string analysis, focusing on their approach and the kinds of properties they can handle. Finally, we choose our approach to the problem, which will be suitable in the context of .NET and Code Contracts and will be able to reason about enough properties.

3.1 Examples of String Properties

All non-relational string properties can be defined as sets of strings (languages) that have the property. In this section, we describe a few selected properties of strings that may appear in .NET programs.

Usually, the properties concern strings that are intended to be processed by the computer (programming languages, serialized data, escape or control sequences), because properties related to strings viewed by users are not very easily defined, and they also typically depend on the runtime environment (using culture specific operations, Section 2.2.4.1), so they are not suitable for static checking. For example, the number of glyphs as seen by the user depends not only on the characters in the string, but also on the used font.

Limited length. String processing code might limit the length of the inputs, for example for buffer allocation. Passing a longer string can lead to buffer overflow, which used to be a common type of bug in C programs. In C#, this would be a problem only in unsafe code. Otherwise, unexpected `IndexOutOfRangeException` or `OutOfMemoryException` exceptions might be thrown.

Valid UTF-16 string. .NET strings may contain invalid UTF-16 code unit sequences. A valid UTF-16 string cannot contain surrogate code units that do not form surrogate pairs. The surrogate pair is formed by the high surrogate first, followed by the low surrogate. This property can be represented by the language $S_{\text{utf16}} = (\{\sharp 0000 \ldots \sharp d7ff\} \cup \{\sharp e000 \ldots \sharp ffff\} \cup (\{\sharp d800 \ldots \sharp dbff\}\ \{\sharp dc00 \ldots \sharp dfff\}))^*$.

ASCII. Strings allowing only ASCII characters [20] are frequently used for their simplicity and interoperability. The set of ASCII characters is $C_{\text{ascii}} = \{\sharp 0000 \ldots \sharp 007f\}$.

Escaped strings. Escaping is a common way to write characters with special meaning such as delimiters or non-printable characters. For example, string literals in C# are delimited by `""` and characters such as newline or double quote must be escaped (`"\n\""`). Escaping input is frequently needed when generating code of a formal language, where the program must ensure that the part generated from the input does not contain unescaped characters that would alter the structure of the result in an unwanted way. Enabling the input to contain unescaped delimiters is the basis of SQL or script injection.

Regex. The `Regex` class requires the expression (passed as a string argument to the constructor or to the matching methods) to be valid – to contain valid escape sequences and to have balanced parentheses. Regex expressions are often compile time constants (string literals in the C# code), so compile-time warning for invalid expressions is possible.

At runtime, a regex representing a constant string can be created by a call of `Regex.Escape`. Then we know it does not contain any special characters and inserting it into a regex will not change its structure.

Format Strings. For generating strings or for stream I/O, format methods (`string.Format`, `StringBuilder.AppendFormat` or methods of the streams) are used. The format is a string, where curly braces contain the index and format modifier (`"{0:d}"`) of the argument that should be inserted at that location. Literal brace characters must be doubled. Format strings are often compile time constants so they could also be checked at compile time. It is, however, not so much of an issue as in C, where the mismatch of the format string and argument type can lead to undefined behavior. If the format string is invalid, `FormatException` is thrown.

Primitive Types. The primitive types of C# can be converted to strings using `ToString` methods. For example, integers in decimal format ($C_{10} = \{\texttt{0} \ldots \texttt{9}\}$) or hexadecimal format ($C_{16} = \{\texttt{0} \ldots \texttt{9}\} \cup \{\texttt{a} \ldots \texttt{f}\} \cup \{\texttt{A} \ldots \texttt{F}\}$). The minus sign and decimal point can be altered by the local culture [15].

Date and Time. Dates and times are more complex common data types. The textual representation varies highly for different locales. The ISO standard specifies fixed format for date ($C_{10}^4\texttt{-}C_{10}^2\texttt{-}C_{10}^2$, for example `2013-01-08`), and time ($C_{10}^2\texttt{:}C_{10}^2$).

GUID. GUID (`Guid` struct) is a 128-bit number which has a hexadecimal text representation with delimiters at fixed places: $S_{\text{guid}} = C_{16}^8 \sharp 001\text{d}\,(C_{16}^4 \sharp 001\text{d})^3\, C_{16}^{12}$.

Base64. The Base64 encoding [21] is used to transfer binary data through ASCII channels. It contains only a safe subset of ASCII characters ($C_{\text{base64}} = \{\texttt{a} \ldots \texttt{z}\} \cup \{\texttt{A} \ldots \texttt{Z}\} \cup \{\texttt{0} \ldots \texttt{9}\} \cup \{\texttt{+}, \texttt{/}, \texttt{=}, \sharp 000\text{a}\}$). Base64 can be easily used from .NET by the `ToBase64String` or `FromBase64String` methods of the `Convert` class.

IP address. IP addresses are used in networking code. Version 4 addresses are usually presented in dotted-decimal format [22, 23], while version 6 addresses use colon-delimited hexadecimal format, and can be shortened [24, 25]. IP addresses can be used in conjunction with subnet prefix or port number [24].

URIs and URLs. URI can contain a limited set of ASCII characters, other characters must be escaped [22]. The language of a URI part without delimiters is $S_{\text{uri}} = (\{\texttt{0} \ldots \texttt{9}\} \cup \{\texttt{A} \ldots \texttt{Z}\} \cup \{\texttt{a} \ldots \texttt{z}\} \cup \{\sharp 002\text{d}, \sharp 002\text{e}, \sharp 005\text{f}, \sharp 007\text{e}\} \cup \texttt{\%}C_{16}C_{16})^*$. URLs often begin with common prefixes such as `http://`.

E-mail Addresses. E-mail addresses are used frequently in web page forms. The exact specification is quite complicated [26], so approximations such as `^(?("")("".+?""@)|(([0-9a-zA-Z]((\.(?!\.))|[-!#\$%&'\*\+/=\?\^`\{\}\|~\w])*)(?<=[0-9a-zA-Z])@))(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-zA-Z][-\w]*[0-9a-zA-Z]\.)+[a-zA-Z]{2,6}))$` [27] or `\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)` [28] are often used.

HTML and XML. XML formats can be used for data serialization, and HTML code is generated by ASP web pages. The grammar for XML [29] and especially HTML is quite complex. When generating HTML from user inputs, attention should be given to escaping (using entities).

3.2 Existing Solutions

Code Contracts. The current version of Code Contracts already supports some properties of strings. This is because the `Length` property of strings is analyzed as an integer value and the string operations have preconditions and postconditions that describe effects of the operation on this property. The contracts are specified in the file `Microsoft.Research/Contracts/MsCorlib/System.String.cs` (see Appendix A). There is also a very limited support for string operations in the code of the arithmetic analysis.

There was also an incomplete implementation of string abstract domains, only available in `DEBUG` builds. It had support for operations returning strings, but the implementation was not precise[1]. Results of operations returning integers or booleans were ignored, so using them in contracts was not possible. The domain was also not tested – for example, it was possible to trigger an infinite recursion in the checker.

BRICS Java String Analyser. Christensen et al. [30] use a constructed flow graph containing join nodes, string operation nodes, and nodes representing regular languages. The flow graph defines a grammar, which is then approximated by multilevel automata built using the Mohri-Nederhof algorithm. Multilevel automata allow transitions to refer to a string operation applied to a subautomaton at a lower level. The multilevel automaton is then converted to a deterministic finite automaton, which can be doubly exponential in size in the worst case.

Hampi. Hampi [31] is a solver for string constraints, which uses bit vector logic to represent strings and makes use of the STP bit-vector constraint solver. It is an useful tool for generating test inputs and counterexamples. Hampi supports regular expression or context free grammars as input languages. However, it restricts the languages to fixed size, claiming it does not pose a limitation.

[1]For example, on the `Top` element, the abstract version of `Concat` always returns `Top`, while a constant concatenated with `Top` has a known prefix. Similarly, `StringAbstraction.Contains` always returns `Top`, while if the argument is `""`, it could return `true`, because all strings contain it (except `null`, but then the operation would throw an exception).

Z3str. Z3str ("string theory plug-in to Z3") [32] adds support for string constraints to the SMT solver Z3. It combines string constraints with the strengths of Z3 for non-string constraints. Strings are not reduced to a uniform domain, but string operations are reduced to concatenation and boolean formulae. There is no support for regular expressions.

Canal. A LLVM-based static analyzer for C programs, Canal, supports a few abstract domains for strings [33]. The thesis presents domain of prefixes, suffixes and finite sets of prefixes.

A Suite of Abstract Domains for Static Analysis of String Values. Costantini [34] proposed several abstract domains for strings. It uses the framework of abstract interpretation, including the definition of domain operators with proofs and abstract semantics for a limited selection of operations.

None of the solutions described above is directly applicable to .NET code. In our solution, we want to support regular expressions, because they can represent most of the properties mentioned in Section 3.1. We want to avoid limiting the lengths of strings considered by the analysis to fixed number, because the string length itself is also an important property that can be used in contracts.

3.3 Chosen Approach

For our implementation of string analyis in Code Contracts, we use abstract interpretation of string values with custom abstract domains. This is because the approach of abstract interpretation is already implemented in Clousot. Design and implementation of suitable abstract domains for strings is the major remaining step.

We decided to implement the string abstract domains proposed by Costantini [34], because they are defined in terms of abstract interpretation, they are not platform-specific, and they provide increasing complexity.

For each abstract domain, we repeat its formal definition in this thesis. Then we define the abstract semantics for the operations specific to .NET. Most of them are our new definitions.

We decided to use the current implementation of string analysis in Clousot as a guide, but we created a different design of the abstract domains. We avoid using SMT solvers or any external software, which would introduce dependencies, non-determinism, etc. Furthermore, using an approach that does not fit into the abstract interpretation framework and the DFA algorithm would complicate reuse of existing features of Clousot and cooperation with other abstract domains.

We do not consider `null` values in our abstract domains. For most of the operations, they can be ignored. For the purpose of `Concat` and `IsNullOrEmpty`, we consider `null` to be an empty string. In other cases, `null` values can be analyzed by the existing "non-null" analysis in Clousot, and this can be utilized using the reduced product of our analysis with other analyses. That way we also make use of the numerical analysis to get information about integer values.

We use the `Regex.IsMatch` method and methods of `string` that return `bool` to specify properties, which can be checked in contracts. Belonging to a language specified by a regular expression covers many of the properties listed in Section 3.1.

By using only the methods of the standard libary, there is no need to extend the contract classes and to require adding references to our assemblies to the projects of the users. The runtime checker works seamlessly with properties written this way, and any existing code already using the supported methods in contracts for runtime checking would also get the benefits of static checking if using our implementation.

4 Abstract Domains for Strings

In this chapter, we describe in detail the abstract domains we chose to implement, and some variants of them, which we did not implement. We used all abstract domains defined in the article by Costantini [34] (i.e., Prefix, Suffix, Character Inclusion, Bricks and String Graphs), and we extended and modified them to suit our needs.

4.1 Design of String Abstract Domains

All abstract domains described here are non-relational, based on a set of elements together with a concretization function. The concretization function assigns a (possibly infinite) language to each element of the domain. For each domain D, apart from the concretization function γ_D, we need to define the bottom and top elements, and the join (least upper bound) and widening operators. Those are needed for the DFA algorithm. We may also define the meet (greatest lower bound) operator and an abstraction function, which forms a Galois connection with γ_D. However, this is not possible for all domains described here.

Operations. For each abstract domain D, we also need to define the abstract semantics, which means implementing the selected string operations on the abstract elements. We consider the methods described in Section 2.2.4, involving strings, integers and booleans. The best possible abstract semantics for the function F_S returning string value is F_D, such that $F_D(x_1, \ldots, x_n) = \alpha_D(\{F_S(s_1, \ldots, s_n) \colon \forall i \colon s_i \in \gamma_D(x_i)\})$. Sometimes, such precise abstract semantics would be too complex or not efficient to evaluate. In those cases, we overapproximate the result (in extreme cases returning $\top_D$). Such definitions can be improved in the future.

For some of the operations and abstract domains, the precision is much better and definition is much simpler, when we assume that some of the arguments are constants. In the case of domains that can exactly represent constants[1], there may be a special case for constant elements in the definition of the operation. For abstract domains that are not precise enough to represent constants (such as the Prefix domain), we add definitions for variants of the operation where some of the arguments are not abstract elements but concrete values.

For operations taking or returning integers, we consider an interval abstraction of integers. Because the integers represent string lengths or indices, we are only interested in non-negative integers. There are two exceptions: `IndexOf`, which can return -1, and `Compare`, which can return any integer value. For the Compare operation, the return value is not specified exactly, and the only interesting property of the result is the sign. We use the $\langle \mathsf{l}\colon, \mathsf{e}\colon, \mathsf{g}\colon \rangle$ notation to specify which signs can the result have (meaning negative, zero and positive). The corresponding set of integers can be obtained as a union of at most three intervals.

[1] An abstract domain can exactly represent constants if for each string, there is an abstract element which represents a language containing exactly the one string.

As method arguments are evaluated eagerly in .NET, when any of the arguments is bottom then the result of the operation is always bottom too. Therefore, in this chapter, we omit the cases of bottom arguments in the definitions of operations.

Assume and assert. To support string properties in contracts, we also need to handle the assert and assume statements. We are particularly interested in those that contain calls to boolean operations on strings. For the `Assert` statement, we simply need to evaluate the abstract operation to know whether the result can be `true` or `false`. This information can then be handed to the framework (through the `IFactQuery` interface) and possibly displayed to the user.

For `Assume` statements, we should use the assumed expression to improve our knowledge about the string variables used in the assumed expression. Because of that, we identify for each domain those boolean operations that help us reason about their arguments, provided that we know the result of the operation.

Regular expressions. To support asserts and assume statements containing `Regex.IsMatch`, we implement for each domain:

- checking whether the abstract element can or must match the regular expression,
- assuming that an element matches (or does not match) the regular expression.

The latter can be done by defining a conversion from a regular expression to an element of the abstract domain, and then combining it with the previous element by the meet operator.

4.2 Constant-based Abstract Domains

Abstract domains based on explicitly listing the possible values are very straightforward and can be defined for any type of values.

4.2.1 Flat Domain

The Flat Domain is the simplest abstract domain that tracks constant values (and constant values only). Using a flat domain is similar to constant propagation in compilers. The elements are string values and two special elements for top and bottom.

Definition 4.2.1 (Flat abstract domain)**.** $\mathcal{FL} = \langle \Sigma^* \cup \{\bot, \top\}, \sqcap_{\mathcal{FL}}, \sqcup_{\mathcal{FL}}, \bot_{\mathcal{FL}}, \top_{\mathcal{FL}} \rangle$. The domain operators are defined in Figure 4.1.

4.2.2 Constant Sets

The Constant Set domain tracks finite sets of strings. To avoid too much memory consumption, the size of the sets is limited.

$$x \sqcup_{\mathcal{FL}} y = \begin{cases} x & x = y \vee y = \bot \\ y & x = \bot \\ \top & \bot \neq x \neq y \neq \bot \end{cases}$$

$$x \sqcap_{\mathcal{FL}} y = \begin{cases} x & x = y \vee y = \top \\ y & x = \top \\ \bot & \text{otherwise} \end{cases}$$

$$\top_{\mathcal{FL}} = \top$$

$$\bot_{\mathcal{FL}} = \bot$$

$$x \sqsubseteq_{\mathcal{FL}} y \Leftrightarrow x = y \vee x = \bot \vee y = \top$$

$$\gamma_{\mathcal{FL}}(x) = x$$

$$\alpha_{\mathcal{FL}}(S) = \begin{cases} \top & |S| > 1 \\ c & |S| = 1 \wedge c \in S \\ \bot & |S| = 0 \end{cases}$$

Figure 4.1: $\mathcal{FL}$ domain

$$\alpha_{\mathcal{CS}_n}(S) = \begin{cases} S & |S| \leq n \\ \top & |S| > n \end{cases}$$

$$\gamma_{\mathcal{CS}_n}(x) = \begin{cases} x & x \neq \top \\ \Sigma^* & x = \top \end{cases}$$

Figure 4.2: $\mathcal{CS}_n$ domain

Definition 4.2.2 (Constant set abstract domain)**.** $\mathcal{CS}_n = \langle \{S \subseteq \Sigma^* \colon |S| \leq n\} \cup \{\top\}, \sqcap_{\mathcal{CS}_n}, \sqcup_{\mathcal{CS}_n}, \bot_{\mathcal{CS}_n}, \top_{\mathcal{CS}_n} \rangle$. The domain operators are defined in Figure 4.2.

Remark 4.2.3. $\mathcal{CS}_1$is equivalent to $\mathcal{FL}$.

The operations are implemented by evaluating them for each combination of the constants of the arguments.

4.2.3 Variants of Constant-Set Based Domains

The limit on the number of constants guarantees termination, but the memory consumption can be still large if the constants themselves are large. We may limit the length of the constants too. On the other hand, we might not limit the number of constants and instead use widening, which returns the top element if the number of constants exceeds the limit n.

$$
\begin{aligned}
\alpha_{\mathcal{L}}(S) &= \{|c| : c \in S\} \\
\gamma_{\mathcal{L}}(x) &= \left\{\bigcirc^i : i \in x\right\} \\
\bot_{\mathcal{L}} &= \emptyset \\
\top_{\mathcal{L}} &= \mathbb{N} \\
x \sqcup_{\mathcal{L}} y &= x \cup y \\
x \sqcap_{\mathcal{L}} y &= x \cap y \\
x \sqsubseteq_{\mathcal{L}} y &\Leftrightarrow x \subseteq y
\end{aligned}
$$

Figure 4.3: $\mathcal{L}$ domain

4.3 Length-based Abstract Domains

Every string value has an integer length. In .NET it is a non-negative integer of a 32-bit signed type (`int`).

4.3.1 Length

The Length abstract domain uses sets of natural numbers (including zero) to represent a language of strings of the specified lengths.

Definition 4.3.1 (Length abstract domain)**.** $\mathcal{L} = \langle \mathcal{P}(\mathbb{N}), \cup, \cap, \emptyset, \mathbb{N} \rangle$. The domain operators are defined in Figure 4.3.

The abstraction can be composed with any abstract domain for integers ($\mathcal{P}(\Sigma^*) \underset{\alpha_{\mathcal{L}}}{\overset{\gamma_{\mathcal{L}}}{\leftrightarrows}} \mathcal{P}(\mathbb{N}) \underset{\alpha_D}{\overset{\gamma_D}{\leftrightarrows}} D$). Suitable domains might be, for example, intervals or a union of a limited number of disjoint intervals.

4.3.1.1 Operations

The operations for the length domain are defined in Figure 4.4. The arguments and results of the string operations are sets of integers. For example, the PadLeft operation may return strings of the same lengths as the input (X), but only those that are at least as long as some of the target lengths (N). Otherwise, the string would be padded to some length in N. The PadRight operation is exactly the same.

The IndexOf operation returns a known index if the needle is known to be empty. If the needle is longer than the haystack, then it cannot be a substring, so the operation returns -1. Otherwise, the occurrence may be at any index, until the last on possible, which is the difference of the lengths. The LastIndexOf operation is similar, but handles an empty needle differently.

4.4 Substring-based Abstract Domains

Abstract domains based on whether the string contains some constant can be easily constructed. This includes restricting the occurence to prefixes or suffixes.

$$
\begin{aligned}
\mathsf{Concat}_{\mathcal{L}}(X,Y) &= \{x+y\colon x \in X \wedge y \in Y\} \\
\mathsf{Insert}_{\mathcal{L}}(X,I,Y) &= \{x+y\colon x \in X \cap [\min I, \infty) \wedge y \in Y\} \\
\mathsf{ReplaceC}_{\mathcal{L}}(X,C,D) &= X \\
\mathsf{ReplaceS}_{\mathcal{L}}(X,Y,Z) &= [\min(X,m), \max(X,n)] \\
&\qquad \text{where } m = \left\lfloor \frac{\min X}{\max Y} \right\rfloor \min Z \\
&\qquad \text{where } n = \left\lceil \frac{\max X}{\min Y} \right\rceil \max Z \\
\mathsf{Substring}_{\mathcal{L}}(X,I,L) &= L \cap [0, \max X - \min I] \\
\mathsf{SubstringEnd}_{\mathcal{L}}(X,I) &= \{x-i\colon x \in X \wedge i \in I\} \cap \mathbb{N} \\
\mathsf{Remove}_{\mathcal{L}}(X,I,L) &= \{x-l\colon x \in X \wedge l \in L\} \cap [\min I, \infty) \\
\mathsf{RemoveEnd}_{\mathcal{L}}(X,I) &= I \cap [0, \max X - 1] \\
\mathsf{PadLeft}_{\mathcal{L}}(X,N,C) &= (X \cap [\min N, \infty)) \cup (N \cap [\min X, \infty)) \\
\mathsf{PadRight}_{\mathcal{L}}(X,N,C) &= \mathsf{PadLeft}_{\mathcal{L}}(X,N,C) \\
\mathsf{Trim}_{\mathcal{L},\mathcal{S}}(X,s) &= [0, \max X] \\
\mathsf{TrimStart}_{\mathcal{L},\mathcal{S}}(X,s) &= [0, \max X] \\
\mathsf{TrimEnd}_{\mathcal{L},\mathcal{S}}(X,s) &= [0, \max X] \\
\mathsf{IsEmpty}_{\mathcal{L}}(X) &= \langle \mathsf{f}\colon X \not\subseteq \{0\}, \mathsf{t}\colon 0 \in X \rangle \\
\mathsf{Contains}_{\mathcal{L}}(X,Y) &= \langle \mathsf{f}\colon Y \not\subseteq \{0\}, \mathsf{t}\colon \min Y \leq \max X \rangle \\
\mathsf{StartsWith}_{\mathcal{L}}(X,Y) &= \mathsf{Contains}_{\mathcal{L}}(X,Y) \\
\mathsf{EndsWith}_{\mathcal{L}}(X,Y) &= \mathsf{Contains}_{\mathcal{L}}(X,Y) \\
\mathsf{Equals}_{\mathcal{L}}(X,Y) &= \langle \mathsf{f}\colon X \cup Y \not\subseteq \{0\}, \mathsf{t}\colon X \cap Y \neq \emptyset \rangle \\
\mathsf{Compare}_{\mathcal{L}}(X,Y) &= \langle \mathsf{l}\colon Y \not\subseteq \{0\}, \mathsf{e}\colon X \cap Y \neq \emptyset, \mathsf{g}\colon X \not\subseteq \{0\} \rangle \\
\mathsf{Length}_{\mathcal{L}}(X) &= X \\
\mathsf{IndexOf}_{\mathcal{L}}(X,Y) &= \begin{cases} \{0\} & Y = \{0\} \\ \{-1\} & \max X < \min Y \\ [-1, \max X - \min Y] & \text{otherwise} \end{cases} \\
\mathsf{LastIndexOf}_{\mathcal{L}}(X,Y) &= \begin{cases} \{\max(x-1,0)\colon x \in X\} & Y = \{0\} \\ \{-1\} & \max X < \min Y \\ [-1, \max X - \min Y] & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.4: $\mathcal{L}$ operations

$$
\begin{aligned}
x \sqcup_{\mathcal{PR}} y &= \mathrm{lcp}(x, y) \\
x \sqcap_{\mathcal{PR}} y &= \begin{cases} x & y \in x\circledast \\ y & x \in y\circledast \\ \bot & \text{otherwise} \end{cases} \\
\top_{\mathcal{PR}} &= \epsilon \\
\bot_{\mathcal{PR}} &= \bot \\
x \sqsubseteq_{\mathcal{PR}} y &\Leftrightarrow x \in y\circledast \\
\gamma_{\mathcal{PR}}(x) &= x\circledast \\
\alpha_{\mathcal{PR}}(S) &= \mathrm{lcp}(S)
\end{aligned}
$$

Figure 4.5: $\mathcal{PR}$ domain

4.4.1 Prefix

The Prefix domain [34] is a good starting example of an abstract domain designed specifically for strings, because it is very simple and allows easy and precise implementation of most operations. However, it is very limited in what properties it can represent.

Definition 4.4.1. The prefix domain is defined as $\mathcal{PR} = (\Sigma^* \cup \{\bot\}, \sqcap_{\mathcal{PR}}, \sqcup_{\mathcal{PR}}, \top_{\mathcal{PR}}, \bot_{\mathcal{PR}})$. The domain operators are defined in Figure 4.5.

There is no infinite increasing sequence in the lattice, so the widening operator is not necessary.

4.4.1.1 Operations

The operations for $\mathcal{PR}$ are defined in Figures 4.6 and 4.7. The function names have the $\mathcal{PR}$ subscript to indicate that their arguments are abstract elements from the prefix domain. The integer arguments are also abstract (sets of integers). When the subscript contains $\mathcal{S}$, it means one or more of the arguments (position indicated in the subscript) is a constant string. That version can be used in the analysis when the argument is a known constant string.

The result of boolean operations is a tuple of the form $\langle \mathsf{f}:, \mathsf{t}:\rangle$, where the f part represents a possibility of returning `false`, and t returning `true`, as shown in Definition 2.1.4. For example the $\mathsf{Contains}_{\mathcal{S},\mathcal{PR}}$ function, which evaluates the `Contains` method, where the first argument is a known constant and the second argument has a known prefix. It is always possible to find a string with such prefix that is not contained in the constant, so there is t in the f part, meaning that the method can always return `false`. However, it can only return `true`, if the known prefix of the second argument is contained in the first argument.

The prefix domain nicely supports operations such as concatenation of a constant string from the left, or inserting a constant into the known part. On the other hand, there is no upper bound on the length of a string, so for example padding is never guaranteed to occur.

$$\mathsf{Concat}_{\mathcal{PR}}(x, y) = x \qquad \mathsf{Concat}_{\mathcal{S},\mathcal{PR}}(s, y) = sy$$

$$\mathsf{Insert}_{\mathcal{PR}}(x, I, y) = \begin{cases} x & \min I > |x| \\ x[:i]y & I = \{i\} \wedge i \leq |x| \\ x[:\min I] & \text{otherwise} \end{cases}$$

$$\mathsf{Insert}_{\mathcal{PR},\mathcal{S}}(x, I, t) = \begin{cases} x & \min I > |x| \\ x[:i]tx[i:] & I = \{i\} \wedge i \leq |x| \\ x[:\min I] & \text{otherwise} \end{cases}$$

$$\mathsf{Insert}_{\mathcal{S},\mathcal{PR}}(s, I, y) = \begin{cases} \bot & \min I > |s| \\ s[:i]y & I = \{i\} \wedge i \leq |s| \\ s[:\min I] & \text{otherwise} \end{cases}$$

$$\mathsf{ReplaceC}_{\mathcal{PR}}(x, C, D) = \begin{cases} x[c := d] & C = \{c\} \wedge D = \{d\} \\ x[:i] & C \cap \operatorname{char}(x) \neq \emptyset \wedge \max(|C|, |D|) > 1 \\ \quad \text{where} & i = \min\{j : x[j] \in C\} \\ x & \text{otherwise} \end{cases}$$

$$\mathsf{ReplaceS}_{\mathcal{PR},\mathcal{S},\mathcal{S}}(x, s, t) = \begin{cases} x[:i]t\mathsf{ReplaceS}_{\mathcal{PR},\mathcal{S},S}(x[i + |s| :], s, t) & x \in \circledast s \circledast \\ \quad \text{where } i = \operatorname{firstindex}(x, s) & \\ x[:\min\{i : x[i:] = s[:|x| - i]\}] & x \notin \circledast s \circledast \end{cases}$$

$$\mathsf{ReplaceS}_{\mathcal{PR}}(x, y, z) = \begin{cases} x[:\operatorname{firstindex}(x, y)] & x \in \circledast y \circledast \\ x[:\min\{i : x[i:] = y[:|x| - i]\}] & x \notin \circledast y \circledast \end{cases}$$

$$\mathsf{Substring}_{\mathcal{PR}}(x, I, L) = \begin{cases} x[i:] & I = \{i\} \wedge i < |x| \leq i + \min L \\ x[i : \min L] & I = \{i\} \wedge i + \min L < |x| \\ \epsilon & \text{otherwise} \end{cases}$$

$$\mathsf{SubstringEnd}_{\mathcal{PR}}(x, I) = \begin{cases} x[i:] & I = \{i\} \wedge i < |x| \\ \epsilon & \text{otherwise} \end{cases}$$

$$\mathsf{Remove}_{\mathcal{PR}}(x, I, L) = \begin{cases} x & \min I \geq |x| \\ x[:i]x[i + l:] & I = \{i\} \wedge L = \{l\} \wedge i + l < |x| \\ x[:\min I] & \text{otherwise} \end{cases}$$

$$\mathsf{RemoveEnd}_{\mathcal{PR}}(x, I) = \begin{cases} x & \min I \geq |x| \\ x[:\min I] & \min I < |x| \end{cases}$$

$$\mathsf{PadLeft}_{\mathcal{PR}}(x, N, C) = \begin{cases} x & |x| \geq \max N \\ \operatorname{lcp}\left(x, c^{\max N}\right) & |x| < \max N \wedge C = \{c\} \\ \epsilon & \text{otherwise} \end{cases}$$

$$\mathsf{PadRight}_{\mathcal{PR}}(x, N, C) = x \qquad \mathsf{Trim}_{\mathcal{PR},\mathcal{S}}(x, s) = \mathsf{Trim}_{\mathcal{S}}(x, s)$$

$$\mathsf{TrimStart}_{\mathcal{PR},\mathcal{S}}(x, s) = \mathsf{TrimStart}_{\mathcal{S}}(x, s)$$

$$\mathsf{TrimEnd}_{\mathcal{PR},\mathcal{S}}(x, s) = \mathsf{TrimEnd}_{\mathcal{S}}(x, s)$$

Figure 4.6: $\mathcal{PR}$ operations

$$
\begin{aligned}
\mathsf{IsEmpty}_{\mathcal{PR}}(x) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon x = \epsilon \rangle \\
\mathsf{Contains}_{\mathcal{PR}}(x, y) &= \top \\
\mathsf{Contains}_{\mathcal{S},\mathcal{PR}}(s, y) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon s \in \circledast y \circledast \rangle \\
\mathsf{Contains}_{\mathcal{PR},\mathcal{S}}(x, d) &= \langle \mathsf{f}\colon x \notin \circledast d \circledast, \mathsf{t}\colon \mathsf{t} \rangle \\
\mathsf{StartsWith}_{\mathcal{PR}}(x, y) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon x \in y \circledast \vee y \in x \circledast \rangle \\
\mathsf{StartsWith}_{\mathcal{S},\mathcal{PR}}(s, y) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon s \in y \circledast \rangle \\
\mathsf{StartsWith}_{\mathcal{PR},\mathcal{S}}(x, d) &= \langle \mathsf{f}\colon x \notin d \circledast, \mathsf{t}\colon x \in d \circledast \vee d \in x \circledast \rangle \\
\mathsf{EndsWith}_{\mathcal{PR}}(x, y) &= \top \\
\mathsf{EndsWith}_{\mathcal{S},\mathcal{PR}}(s, y) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon s \in \circledast y \circledast \rangle \\
\mathsf{EndsWith}_{\mathcal{PR},\mathcal{S}}(x, d) &= \langle \mathsf{f}\colon d \neq \epsilon, \mathsf{t}\colon \mathsf{t} \rangle \\
\mathsf{Equals}_{\mathcal{PR}}(x, y) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon x \in y \circledast \vee y \in x \circledast \rangle \\
\mathsf{Equals}_{\mathcal{S},\mathcal{PR}}(s, y) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon s \in y \circledast \rangle \\
\mathsf{Compare}_{\mathcal{PR}}(x, y) &= \begin{cases} \langle \mathsf{l}\colon \mathsf{t}, \mathsf{e}\colon \mathsf{t}, \mathsf{g}\colon \mathsf{t} \rangle & x \in y \circledast \vee y \in x \circledast \\ \langle \mathsf{l}\colon \mathsf{t}, \mathsf{e}\colon \mathsf{f}, \mathsf{g}\colon \mathsf{f} \rangle & x < y \wedge y \notin x \circledast \\ \langle \mathsf{l}\colon \mathsf{f}, \mathsf{e}\colon \mathsf{f}, \mathsf{g}\colon \mathsf{t} \rangle & x > y \wedge x \notin y \circledast \end{cases} \\
\mathsf{Compare}_{\mathcal{S},\mathcal{PR}}(s, y) &= \begin{cases} \langle \mathsf{l}\colon \mathsf{t}, \mathsf{e}\colon \mathsf{t}, \mathsf{g}\colon \mathsf{t} \rangle & s \in y \circledast \wedge s \neq y \\ \langle \mathsf{l}\colon \mathsf{t}, \mathsf{e}\colon \mathsf{t}, \mathsf{g}\colon \mathsf{f} \rangle & s = y \\ \langle \mathsf{l}\colon \mathsf{t}, \mathsf{e}\colon \mathsf{f}, \mathsf{g}\colon \mathsf{f} \rangle & s < y \\ \langle \mathsf{l}\colon \mathsf{f}, \mathsf{e}\colon \mathsf{f}, \mathsf{g}\colon \mathsf{t} \rangle & s > y \wedge s \notin y \circledast \end{cases} \\
\mathsf{Length}_{\mathcal{PR}}(x) &= [|x|, \infty) \\
\mathrm{overlaps}(x, s) &= \{i\colon x[i:] = s[:|x| - i]\} \\
\mathrm{firstindexset}(s, y) &= \{\mathrm{firstindex}(s, s[i:])\colon i \in \mathrm{indexset}(s, y)\} \\
\mathsf{IndexOf}_{\mathcal{PR}}(x, y) &= \{-1\} \cup \mathrm{indexset}(x, y) \cup \mathrm{overlaps}(x, y) \cup [|x|, \infty) \\
\mathsf{IndexOf}_{\mathcal{S},\mathcal{PR}}(s, y) &= \begin{cases} \{-1, 0\} & s = y = \epsilon \\ \{-1\} \cup \mathrm{firstindexset}(s, y) & \text{otherwise} \end{cases} \\
\mathsf{IndexOf}_{\mathcal{PR},\mathcal{S}}(x, s) &= \begin{cases} \mathrm{firstindex}(x, s) & x \in \circledast s \circledast \\ \{-1\} \cup \mathrm{overlaps}(x, s) \cup [|x|, \infty) & x \notin \circledast s \circledast \end{cases} \\
\mathsf{LastIndexOf}_{\mathcal{PR}}(x, y) &= \{-1\} \cup \mathrm{indexset}(x, y) \cup \mathrm{overlaps}(x, y) \cup [|x|, \infty) \\
\mathsf{LastIndexOf}_{\mathcal{S},\mathcal{PR}}(s, y) &= \begin{cases} \{-1, 0\} & s = y = \epsilon \\ \{-1\} \cup \mathrm{indexset}(s, y) \setminus \{|s|\} & \text{otherwise} \end{cases} \\
\mathsf{LastIndexOf}_{\mathcal{PR},\mathcal{S}}(x, s) &= \begin{cases} [\mathrm{lastindex}(x, s), \infty) & x \in \circledast s \circledast \wedge s \neq \epsilon \\ [0, \infty) & x = s = \epsilon \\ [|x| - 1, \infty) & x \neq s = \epsilon \\ \{-1\} \cup [\min \mathrm{overlaps}(x, s), \infty) & x \notin \circledast s \circledast \end{cases}
\end{aligned}
$$

Figure 4.7: $\mathcal{PR}$ operations returning integers and booleans

Listing 4.1 Using `Contract.Assume` with $\mathcal{PR}$

```
// x -> Top
Contract.Assume(
 x.StartsWith("prefix", StringComparison.Ordinal));
// x -> Prefix("prefix")
x = "some" + x;
// x -> Prefix("someprefix")
Contract.Assert(
 x.StartsWith("someprefix", StringComparison.Ordinal));
// assertion proven
```

Some of the operations check whether two prefixes are comparable (using $\sqsubseteq_{\mathcal{PR}}$). For example, the StartsWith or Equals operations can never tell for sure that one argument starts or is equal to the other one, because there might be anything following the known prefix. If the prefixes are comparable, that means they are the same or one of them is longer, so the strings can be the same. If they are incomparable, then there is no chance of them being the same. On the other hand, the EndsWith method always returns unknown because we know nothing about the ends of the strings.

4.4.1.2 Regular Expressions

The regular expressions that correspond to prefixes are of the form `"^prefix"`.

Figure 4.8 shows definitions of regular expression matching and conversion from regular expressions to prefixes. In the first case, we have a known prefix and we want to know whether the strings starting with the prefix can or must match the regular expression. We interpret the regular expression, passing along a parameter z, which carries the offset from the begin() anchor. If z is $\bot$, it means we have not found the anchor yet, if it is $\top$, it means the offset from the anchor is not known.

The function fromRE interprets the regular expression to generate a prefix that overapproximates the set of string matching the regular expression. We again do a forward depth-first traversal of the tree, keeping the state in the parameter z. When z becomes $\top$, no more characters are added to the prefix.

4.4.1.3 Assume

The StartsWith predicate exactly represents the relationship between a string variable and its prefix abstraction. If we know that StartsWith returns t, we know that the first argument is soundly represented by the abstraction of the second argument, as shown in Figure 4.9. The same holds for Equals. We support IsMatch by converting the regular expression to a prefix using the fromRE function.

The assume statement can be used like in Listing 4.1.

4.4.2 Suffix

The Suffix domain is very similar to the prefix domain. The definition and domain operators are symmetric. The difference is in the operations that use indexing or look at the string in the forward direction (ReplaceS). Their implementation

$$
\begin{aligned}
\mathsf{IsMatch}_{\mathcal{PR}}(x, r) &= \mathrm{isMatch}(x, \mathrm{regex}(r), \bot) \\
\mathrm{isMatch}(x, \mathsf{begin}(), z) &= \begin{cases} \langle \mathsf{t}, 0 \rangle & z = \bot \vee z = 0 \\ \langle \mathsf{f}, \bot \rangle & z > 0 \end{cases} \\
\mathrm{isMatch}(x, \mathsf{end}(), z) &= \langle \top, \top \rangle \\
\mathrm{isMatch}(x, \mathsf{concat}(M_1, \ldots, M_n), z_0) &= \left\langle \bigwedge_{i=1}^{n} b_i, z_n \right\rangle \\
&\quad \text{where } \langle b_i, z_i \rangle = \mathrm{isMatch}(x, M_i, z_{i-1}) \\
\mathrm{isMatch}(x, \mathsf{union}(M_1, \ldots, M_n), z) &= \left\langle \bigvee_{i=1}^{n} b_i, \bigvee_{i=1}^{n} z_i \right\rangle \\
&\quad \text{where } \langle b_i, z_i \rangle = \mathrm{isMatch}(x, M_i, z) \\
\mathrm{isMatch}(x, \mathsf{single}(S), z) &= \begin{cases} \langle x[z] \in S, z + 1 \rangle & z \in [0, |x| - 1] \\ \langle \top, z + 1 \rangle & z \geq |x| \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{fromRE}(\mathsf{union}(M_1, \ldots, M_n), z) &= \langle \epsilon, \top \rangle \\
\mathrm{fromRE}(\mathsf{begin}(), z) &= \begin{cases} \langle \epsilon, 0 \rangle & z = \bot \vee z = 0 \\ \langle \bot, \top \rangle & z > 0 \\ \langle \epsilon, \top \rangle & z = \top \end{cases} \\
\mathrm{fromRE}(\mathsf{end}(), z) &= \langle \epsilon, \top \rangle \\
\mathrm{fromRE}(\mathsf{empty}(), z) &= \langle \epsilon, z \rangle \\
\mathrm{fromRE}(\mathsf{concat}(M_1, \ldots, M_n), z_0) &= \left\langle \prod_{i=1}^{n} x_i, z_n \right\rangle \\
&\quad \text{where } \langle x_i, z_i \rangle = \mathrm{fromRE}(M_i, z_{i-1}) \\
\mathrm{fromRE}(\mathsf{single}(S), z) &= \begin{cases} \langle s, z + 1 \rangle & S = \{s\} \wedge z \in \mathbb{N} \\ \langle \bot, \top \rangle & S = \emptyset \\ \langle \epsilon, \top \rangle & |S| > 1 \vee z \notin \mathbb{N} \end{cases} \\
\mathrm{fromRE}(M) &= x \\
&\quad \text{where } \langle x, z \rangle = \mathrm{fromRE}(M, \bot)
\end{aligned}
$$

Figure 4.8: $\mathcal{PR}$ support for regular expressions

$$
\begin{aligned}
\mathsf{StartsWith}_{\mathcal{PR}}(\alpha_{\mathcal{PR}}(S), y) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{PR}}(y) \\
\mathsf{Equals}_{\mathcal{PR}}(\alpha_{\mathcal{PR}}(S), y) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{PR}}(y) \\
\mathsf{IsMatch}_{\mathcal{PR}}(\alpha_{\mathcal{PR}}(S), r) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{PR}}(\mathrm{fromRE}(\mathrm{regex}(r)))
\end{aligned}
$$

Figure 4.9: $\mathcal{PR}$ assume operations

Listing 4.2 Example program for $\mathcal{SU}$

```
string FileName(string n){
 Contract.Ensures(
  Contract.Result<string>().
  EndsWith(".jpg", StringComparison.Ordinal)
 );

 string fn;
 if(n == null){
  fn = "default.jpg";
 }
 else if(n.EndsWith(".jpg")){
  fn = n;
 }
 else{
  fn = n + ".jpg";
 }
 return fn;
}
```

$$
\begin{aligned}
x \sqcup_{\mathcal{SU}} y &= \mathrm{lcs}(x, y) \\
x \sqcap_{\mathcal{SU}} y &= \begin{cases} x & y \in \circledast x \\ y & x \in \circledast y \\ \bot & \text{otherwise} \end{cases} \\
\top_{\mathcal{SU}} &= \epsilon \\
\bot_{\mathcal{SU}} &= \bot \\
x \sqsubseteq_{\mathcal{SU}} y &\Leftrightarrow x \in \circledast y \\
\gamma_{\mathcal{SU}}(x) &= \circledast x \\
\alpha_{\mathcal{SU}}(S) &= \mathrm{lcs}(S)
\end{aligned}
$$

Figure 4.10: $\mathcal{SU}$ domain

is more difficult and less precise with the Suffix domain, because the indexing is from the start, which is unknown.

Listing 4.2 shows an example program using strings where the file suffixes are always `.jpg`.

Definition 4.4.2 (Suffix domain)**.** $\mathcal{SU} = (\Sigma^* \cup \{\bot\}, \sqcap_{\mathcal{SU}}, \sqcup_{\mathcal{SU}}, \top_{\mathcal{SU}}, \bot_{\mathcal{SU}})$. The domain operators are defined in Figure 4.10.

4.4.2.1 Operations

The operations for $\mathcal{SU}$ are defined in Figure 4.11.

4.4.2.2 Regular Expressions

The regular expressions that correspond to prefixes are of the form `"abc\\z"`. Regular expressions of the form `"abc$"` are more common, but they do not rep-

$$\mathsf{Concat}_{\mathcal{SU}}(x, y) = y$$
$$\mathsf{Concat}_{\mathcal{SU},\mathcal{S}}(x, t) = xt$$
$$\mathsf{Insert}_{\mathcal{SU}}(x, I, y) = \begin{cases} x[i:] & y = \epsilon \\ \mathrm{lcs}(x[:i], y, y[:|y|-1])x[i:] & y \neq \epsilon \\ \quad \text{where } i = \min\{|x|, \sup I\} & \end{cases}$$
$$\mathsf{Insert}_{\mathcal{SU},\mathcal{S}}(x, I, t) = \begin{cases} x & t = \epsilon \vee i = 0 \\ \mathrm{lcs}(x[:i]t, x[:i-1]tx[i-1])x[i:] & t \neq \epsilon \\ \quad \text{where } i = \min\{|x|, \sup I\} & \end{cases}$$
$$\mathsf{Insert}_{\mathcal{S},\mathcal{SU}}(s, I, y) = \begin{cases} ys[i:] & I = \{i\} \\ \mathsf{Insert}_{\mathcal{SU}}(s, I, y) & \text{otherwise} \end{cases}$$
$$\mathsf{ReplaceC}_{\mathcal{SU}}(x, C, D) = \begin{cases} x[c := d] & C = \{c\} \wedge D = \{d\} \\ x[i+1:] & C \cap \mathrm{char}(x) \neq \emptyset \wedge \max(|C|, |D|) > 1 \\ \quad \text{where} & i = \max\{j : x[j] \in C\} \\ x & \text{otherwise} \end{cases}$$
$$\mathsf{ReplaceS}_{\mathcal{SU}}(x, y, z) = \epsilon$$
$$\mathsf{Substring}_{\mathcal{SU}}(x, I, L) = \epsilon$$
$$\mathsf{SubstringEnd}_{\mathcal{SU}}(x, I) = x[\min\{|x|, \sup I\}:]$$
$$\mathsf{Remove}_{\mathcal{SU}}(x, I, l) = \begin{cases} \mathrm{lcs}(x[m:], x[m+l:]) & \\ \quad \text{where } m = \min\{|x| - l, \sup I\} & \\ \epsilon & l \geq |x| \end{cases}$$
$$\mathsf{RemoveEnd}_{\mathcal{SU}}(x, I) = \epsilon$$
$$\mathsf{PadLeft}_{\mathcal{SU}}(x, N, C) = x$$
$$\mathsf{PadRight}_{\mathcal{SU}}(x, n, c) = \begin{cases} x & |x| \geq n \\ \mathrm{lcs}(x, c^n) & |x| < n \end{cases}$$
$$\mathsf{Trim}_{\mathcal{SU},\mathcal{S}}(x, c) = \mathsf{Trim}_{\mathcal{S}}(x, c)$$
$$\mathsf{TrimStart}_{\mathcal{SU},\mathcal{S}}(x, c) = \mathsf{TrimStart}_{\mathcal{S}}(x, c)$$
$$\mathsf{TrimEnd}_{\mathcal{SU},\mathcal{S}}(x, c) = \mathsf{TrimEnd}_{\mathcal{S}}(x, c)$$

Figure 4.11: $\mathcal{SU}$ operations

$$
\begin{aligned}
\mathsf{IsEmpty}_{\mathcal{SU}}(x) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon x = \epsilon \rangle \\
\mathsf{Contains}_{\mathcal{SU}}(x, y) &= \top \\
\mathsf{StartsWith}_{\mathcal{SU}}(x, y) &= \top \\
\mathsf{EndsWith}_{\mathcal{SU}}(x, y) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon x \in \circledast y \vee y \in \circledast x \rangle \\
\mathsf{EndsWith}_{\mathcal{SU},\mathcal{S}}(x, d) &= \langle \mathsf{f}\colon x \notin \circledast d, \mathsf{t}\colon x \in \circledast d \vee d \in \circledast x \rangle \\
\mathsf{Equals}_{\mathcal{SU}}(x, y) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon x \in \circledast y \vee y \in \circledast x \rangle \\
\mathsf{Compare}_{\mathcal{SU}}(x, y) &= \langle \mathsf{l}\colon \mathsf{t}, \mathsf{e}\colon x \in \circledast y \vee y \in \circledast x, \mathsf{g}\colon \mathsf{t} \rangle \\
\mathsf{Length}_{\mathcal{SU}}(x) &= [|x|, \infty) \\
\mathsf{IndexOf}_{\mathcal{SU},\mathcal{S}}(x, c) &= \begin{cases} 0 & c = \epsilon \\ [0, \infty) & x \in \circledast c \circledast \wedge c \neq \epsilon \\ [-1, \infty) & x \notin \circledast c \circledast \end{cases} \\
\mathsf{LastIndexOf}_{\mathcal{SU},\mathcal{S}}(x, c) &= \begin{cases} [0, \infty) & x = c = \epsilon \\ [|x| - 1, \infty) & x \neq c = \epsilon \\ [\mathrm{lastindex}(x, c), \infty) & x \in \circledast c \circledast \wedge c \neq \epsilon \\ [-1, \infty) & x \notin \circledast c \circledast \end{cases}
\end{aligned}
$$

Figure 4.12: $\mathcal{SU}$ operations

resent a language of strings with the same suffix, because the anchor can match the newline character at the end of the string. Otherwise, it is symmetric with prefixes, so we use the same algorithm as in Figure 4.8, but considering both the string value and the regular expression in reverse.

4.4.2.3 Assume

The EndsWith predicate exactly represents the relationship between a value and its suffix abstraction ($\mathsf{EndsWith}_{\mathcal{SU}}(\alpha_{\mathcal{SU}}(S), y) = \mathsf{t} \Rightarrow S \subseteq \gamma_{\mathcal{SU}}(y)$), similarly to $\mathcal{PR}$ and StartsWith. Equals and IsMatch work similarly to $\mathcal{PR}$ (see Figure 4.9).

4.4.3 Variants of Domains Based on Substrings

The ideas of the prefix and suffix domains can be used to create other similar domains.

String containment. The prefix and suffix domains require a specific position where the substring must occur. This could be relaxed to allow the string anywhere. The operation used in assume statements would be Contains. However, because the relative positions of the substrings is not fixed, there is not a unique lowest upper bound for all pairs of elements.

Sets of prefixes and suffixes. The definitions of $\mathcal{PR}$ and $\mathcal{SU}$ allow only one possible prefix or suffix. We could allow more of them, like in the $\mathcal{CS}_n$ domains.

$$\alpha_{\mathcal{CE}_f}(S) = \left\{ \prod_{i=0}^{|s|-1} f(s[i]) \colon s \in S \right\}$$
$$\gamma_{\mathcal{CE}_f}(X) = \left\{ \prod_{i=0}^{|s|-1} f^{-1}(s[i]) \colon s \in X \right\}$$

Figure 4.13: $\mathcal{CE}_f$ domain

Reduced products. The $\mathcal{PR}$, $\mathcal{SU}$ and similar domains could be combined using reduced product. However, the known prefix and suffix may or may not overlap.

Relational domains. The domains above track values of variables individually by relating a variable to a constant. We could also use another variable as the substring (prefix or suffix). In addition to assertions of the kind "variable has a constant prefix", we would have also "variable is a prefix of a constant" and "variable is prefix of another variable". Such domain would also track equality (when both strings are prefixes of each other).

4.5 Character-set-based Abstract Domains

Abstract domains based on sets of characters consider the characters in isolation, not looking at their relative or absolute positions within the string.

4.5.1 Character Set Abstraction

It might be unnecessarily precise to consider every single character of the alphabet, which is not small when using Unicode. We can partition the alphabet into equivalence classes and consider all equivalent characters to be the same. The useful classes may be ASCII characters or Unicode categories. What we must be aware of, is that if the abstractions of two characters are the same, it only means they are the same characters if the equivalence class has size 1.

Definition 4.5.1 (Character set abstraction)**.** Let Σ_1, Σ_2 be finite alphabets, f a function $f \colon \Sigma_1 \to \Sigma_2$, and $f^{-1} \colon \Sigma_2 \to \mathcal{P}(\Sigma_1)$, such that $f^{-1}(d) = \{c \in \Sigma_1 \colon f(c) = d\}$. Then the domain of strings with equivalent characters is $\mathcal{CE}_f = \langle \mathcal{P}(\Sigma_2^*), \cap, \cup, \emptyset, \Sigma_2^* \rangle$. The abstraction and concretization funtions are defined in Figure 4.13.

4.5.2 Allowed Characters

The domain of allowed characters restricts the set of available characters. This domain can track the property that a string does not contain certain characters.

Definition 4.5.2 (Domain of allowed characters)**.** $\mathcal{CA} = \langle \mathcal{P}(\Sigma) \cup \{\bot\}, \sqcap_{\mathcal{CA}}, \sqcup_{\mathcal{CA}}, \bot_{\mathcal{CA}}, \top_{\mathcal{CA}} \rangle$. The domain operators are defined in Figure 4.14.

$$
\begin{aligned}
\gamma_{\mathcal{CA}}(x) &= x^* \\
\alpha_{\mathcal{CA}}(S) &= \bigcup_{c \in S} \text{char}(c) \\
x \sqcup_{\mathcal{CA}} y &= x \cup y \\
x \sqcap_{\mathcal{CA}} y &= x \cap y \\
x \sqsubseteq_{\mathcal{CA}} y &= x \subseteq y \\
\top_{\mathcal{CA}} &= \Sigma \\
\bot_{\mathcal{CA}} &= \bot
\end{aligned}
$$

Figure 4.14: $\mathcal{CA}$ domain

$$
\begin{aligned}
\gamma_{\mathcal{CM}}(x) &= \{s \in \Sigma^* \colon x \subseteq \text{char}(s)\} \\
\alpha_{\mathcal{CM}}(S) &= \bigcap_{c \in S} \text{char}(c) \\
\top_{\mathcal{CM}} &= \emptyset \\
\bot_{\mathcal{CM}} &= \bot \\
x \sqcup_{\mathcal{CM}} y &= x \cap y
\end{aligned}
$$

Figure 4.15: $\mathcal{CM}$ domain

The char function represents the set of character in a string.

4.5.3 Mandatory Characters

The domain of mandatory characters is similar to $\mathcal{CA}$, but all characters are allowed and the specified characters must all occur at least once in the string.

Definition 4.5.3 (Domain of mandatory characters)**.** $\mathcal{CM} = \langle \mathcal{P}(\Sigma) \cup \{\bot\}, \sqcap_{\mathcal{CM}}, \sqcup_{\mathcal{CM}}, \bot_{\mathcal{CM}}, \top_{\mathcal{CM}} \rangle$. The domain operators are defined in Figure 4.15.

4.5.4 Character Inclusion

The reduced product of the previous two domains, $\mathcal{CA}$ and $\mathcal{CM}$, is defined by Costantini as "Character Inclusion" [30]. It tracks information about both characters that are allowed in a string and about characters that are mandatory in a string. We use a tuple syntax $\langle \mathsf{m}\colon, \mathsf{a}\colon \rangle$ to represent the abstract elements, where the first part is the set of mandatory and the second part is the set of allowed characters. The subscript notation is used to extract one of the sets from the tuple.

Definition 4.5.4 (Character inclusion abstract domain)**.** $\mathcal{CI} = \langle \{\langle \mathsf{m}\colon x_{\mathsf{m}}, \mathsf{a}\colon x_{\mathsf{a}} \rangle : x_{\mathsf{m}} \subseteq x_{\mathsf{a}} \subseteq \Sigma\} \cup \{\bot\}, \sqcap_{\mathcal{CI}}, \sqcup_{\mathcal{CI}}, \bot_{\mathcal{CI}}, \top_{\mathcal{CI}} \rangle$. The domain operators are defined in Figure 4.16.

$$
\begin{aligned}
\gamma_{\mathcal{CI}}(x) &= \{s \in \Sigma^* \colon x_{\mathsf{m}} \subseteq \mathrm{char}(s) \subseteq x_{\mathsf{a}}\} \\
\alpha_{\mathcal{CI}}(S) &= \left\langle \mathsf{m}\colon \bigcap_{c \in S} \mathrm{char}(c), \mathsf{a}\colon \bigcup_{c \in S} \mathrm{char}(c) \right\rangle \\
x \sqcup_{\mathcal{CI}} y &= \langle \mathsf{m}\colon x_{\mathsf{m}} \cap y_{\mathsf{m}}, \mathsf{a}\colon x_{\mathsf{a}} \cup y_{\mathsf{a}} \rangle \\
x \sqcap_{\mathcal{CI}} y &= \begin{cases} \langle \mathsf{m}\colon x_{\mathsf{m}} \cup y_{\mathsf{m}}, \mathsf{a}\colon x_{\mathsf{a}} \cap y_{\mathsf{a}} \rangle & x_{\mathsf{m}} \subseteq y_{\mathsf{a}} \wedge y_{\mathsf{m}} \subseteq x_{\mathsf{a}} \\ \bot & x_{\mathsf{m}} \not\subseteq y_{\mathsf{a}} \vee y_{\mathsf{m}} \not\subseteq x_{\mathsf{a}} \end{cases} \\
\bot_{\mathcal{CI}} &= \bot \\
\top_{\mathcal{CI}} &= \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \Sigma \rangle \\
x \sqsubseteq_{\mathcal{CI}} y &\Leftrightarrow x = \bot_{\mathcal{CI}} \vee (y_{\mathsf{m}} \subseteq x_{\mathsf{m}} \wedge x_{\mathsf{a}} \subseteq y_{\mathsf{a}})
\end{aligned}
$$

Figure 4.16: $\mathcal{CI}$ domain

4.5.4.1 Operations

The definitions of abstract operations for $\mathcal{CI}$ are in Figures 4.17 and 4.18. We take advantage of the fact that when there are any mandatory characters, then the string must not be empty. Some operations give more precise result if only a single character is allowed, or if the argument is a single-character constant. For example, Contains cannot return f if the second argument is a single-character constant that is mandatory in the first argument.

4.5.4.2 Regular Expressions

The Character Inclusion domain is less sensitive to the structure of the regular expression than in the case of Prefix and Suffix domains. To get a $\mathcal{CI}$ element from a regular expression, we interpret the expression bottom-up. The set of allowed characters can be only constrained if a part of the regular expression is closed between the begin() and end() anchors. This is signaled by the p parameter of the fromRE function. If it is c (closed), it means the regular expression must match the whole string end-to-end. Otherwise, if the parameter is o (open), all characters are allowed, because the match can occur on any substring and there can be any characters before or after the match.

For this abstract domain, the abstraction for a negative match (when `Regex.IsMatch` returns `false`) can also be computed. For example, if a string is known not to match a regular expression that is a set of characters, it means that none of those characters are allowed in the string.

The isMatch function also uses the open/closed parameter. In the concatenation and loop cases, it uses fromRE and fromNRE to get more precision than can be achieved by just recursively applying isMatch. The regular expression functions are defined in Figures 4.19 and 4.20.

4.5.4.3 Assume

The operation suited for use in assume statements for the $\mathcal{CI}$ domain is Contains. We can use both positive and negative assumptions (when the second argument is

$$\mathsf{Concat}_{\mathcal{CI}}(x, y) = \langle \mathsf{m}\colon x_\mathsf{m} \cup y_\mathsf{m}, \mathsf{a}\colon x_\mathsf{a} \cup y_\mathsf{a} \rangle$$

$$\mathsf{Insert}_{\mathcal{CI}}(x, I, y) = \begin{cases} \langle \mathsf{m}\colon x_\mathsf{m} \cup y_\mathsf{m}, \mathsf{a}\colon x_\mathsf{a} \cup y_\mathsf{a} \rangle & x_\mathsf{a} \neq \emptyset \vee 0 \in I \\ \bot & x_\mathsf{a} = \emptyset \wedge 0 \notin I \end{cases}$$

$$\mathsf{ReplaceC}_{\mathcal{CI}}(x, C, D) = \left\langle \begin{array}{l} \mathsf{m}\colon \begin{cases} x_\mathsf{m} \setminus C \cup D & C \subseteq x_\mathsf{m} \wedge |D| = 1 \\ x_\mathsf{m} \setminus C & \text{otherwise} \end{cases} \\ \mathsf{a}\colon \begin{cases} x_\mathsf{a} & C \cap x_\mathsf{a} = \emptyset \\ x_\mathsf{a} \setminus C \cup D & |C| = 1 \\ x_\mathsf{a} \cup D & \text{otherwise} \end{cases} \end{array} \right\rangle$$

$$\mathsf{ReplaceS}_{\mathcal{CI}}(x, y, z) = \begin{cases} \langle \mathsf{m}\colon x_\mathsf{m} \setminus y_\mathsf{a}, \mathsf{a}\colon x_\mathsf{a} \cup z_\mathsf{a} \rangle & y_\mathsf{m} \subseteq x_\mathsf{a} \\ x & y_\mathsf{m} \nsubseteq x_\mathsf{a} \end{cases}$$

$$\mathsf{Substring}_{\mathcal{CI}}(x, I, L) = \begin{cases} \bot & x_\mathsf{a} = \emptyset \wedge 0 \notin I \cap L \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \emptyset \rangle & L = \{0\} \\ \langle \mathsf{m}\colon x_\mathsf{a}, \mathsf{a}\colon x_\mathsf{a} \rangle & 0 \notin L \wedge |x_\mathsf{a}| = 1 \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon x_\mathsf{a} \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{SubstringEnd}_{\mathcal{CI}}(x, I) = \begin{cases} \bot & 0 \notin I \wedge x_\mathsf{a} = \emptyset \\ x & I = \{0\} \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon x_\mathsf{a} \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{Remove}_{\mathcal{CI}}(x, I, L) = \begin{cases} \bot & x_\mathsf{a} = \emptyset \wedge 0 \notin I \cap L \\ x & L = \{0\} \\ \langle \mathsf{m}\colon x_\mathsf{a}, \mathsf{a}\colon x_\mathsf{a} \rangle & 0 \notin I \wedge |x_\mathsf{a}| = 1 \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon x_\mathsf{a} \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{RemoveEnd}_{\mathcal{CI}}(x, I) = \begin{cases} \bot & x_\mathsf{a} = \emptyset \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \emptyset \rangle & I = \{0\} \wedge x_\mathsf{a} \neq \emptyset \\ \langle \mathsf{m}\colon x_\mathsf{a}, \mathsf{a}\colon x_\mathsf{a} \rangle & 0 \notin I \wedge |x_\mathsf{a}| = 1 \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon x_\mathsf{a} \rangle & \text{otherwise} \end{cases}$$

$$\mathsf{PadLeft}_{\mathcal{CI}}(x, N, C) = \left\langle \begin{array}{l} \mathsf{m}\colon \begin{cases} C & \min N > 0 \wedge x_\mathsf{a} \subseteq C \wedge |C| = 1 \\ x_\mathsf{m} & \min N = 0 \vee x_\mathsf{a} \nsubseteq C \vee |C| \geq 1 \end{cases} \\ \mathsf{a}\colon \begin{cases} x_\mathsf{a} & |x_\mathsf{m}| \geq \max N \\ x_\mathsf{a} \cup C & |x_\mathsf{m}| < \max N \end{cases} \end{array} \right\rangle$$

$$\mathsf{PadRight}_{\mathcal{CI}}(x, N, C) = \mathsf{PadLeft}_{\mathcal{CI}}(x, N, C)$$

$$\mathsf{Trim}_{\mathcal{CI}}(x, y) = \begin{cases} \langle \mathsf{m}\colon x_\mathsf{m} \setminus y_\mathsf{a}, \mathsf{a}\colon x_\mathsf{a} \rangle & x_\mathsf{a} \nsubseteq y_\mathsf{m} \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \emptyset \rangle & x_\mathsf{a} \subseteq y_\mathsf{m} \end{cases}$$

$$\mathsf{TrimStart}_{\mathcal{CI}}(x, y) = \mathsf{TrimEnd}_{\mathcal{CI}}(x, y) = \mathsf{Trim}_{\mathcal{CI}}(x, y)$$

Figure 4.17: $\mathcal{CI}$ operations returning strings

$$
\begin{aligned}
\mathsf{IsEmpty}_{\mathcal{CI}}(x) &= \langle \mathsf{f}\colon x_\mathsf{a} \neq \emptyset, \mathsf{t}\colon x_\mathsf{m} = \emptyset \rangle \\
\mathsf{Contains}_{\mathcal{CI}}(x, y) &= \langle \mathsf{f}\colon y_\mathsf{a} \neq \emptyset, \mathsf{t}\colon y_\mathsf{m} \subseteq x_\mathsf{a} \rangle \\
\mathsf{Contains}_{\mathcal{CI},\mathcal{S}}(x, s) &= \langle \mathsf{f}\colon s \neq \epsilon \wedge (|s| > 1 \vee s[0] \notin x_\mathsf{m}), \mathsf{t}\colon \mathrm{char}(s) \subseteq x_\mathsf{a} \rangle \\
\mathsf{StartsWith}_{\mathcal{CI}}(x, y) &= \langle \mathsf{f}\colon y_\mathsf{a} \neq \emptyset, \mathsf{t}\colon y_\mathsf{m} \subseteq x_\mathsf{a} \rangle \\
\mathsf{StartsWith}_{\mathcal{CI},\mathcal{S}}(x, s) &= \left\langle \begin{array}{l} \mathsf{f}\colon s \neq \epsilon \wedge (|s| > 1 \vee s[0] \notin x_\mathsf{m} \vee x_\mathsf{a} \neq \{s[0]\}) \\ \mathsf{t}\colon \mathrm{char}(s) \subseteq x_\mathsf{a} \end{array} \right\rangle \\
\mathsf{EndsWith}_{\mathcal{CI}}(x, y) &= \mathsf{StartsWith}_{\mathcal{CI}}(x, y) \\
\mathsf{EndsWith}_{\mathcal{CI},\mathcal{S}}(x, s) &= \mathsf{StartsWith}_{\mathcal{CI},\mathcal{S}}(x, s) \\
\mathsf{Equals}_{\mathcal{CI}}(x, y) &= \langle \mathsf{f}\colon x_\mathsf{a} \neq \emptyset \vee y_\mathsf{a} \neq \emptyset, \mathsf{t}\colon x_\mathsf{m} \subseteq y_\mathsf{a} \wedge y_\mathsf{m} \subseteq x_\mathsf{a} \rangle \\
\mathsf{Compare}_{\mathcal{CI}}(x, y) &= \langle \mathsf{l}\colon \mathrm{anyLess}(x, y), \mathsf{e}\colon x_\mathsf{m} \subseteq y_\mathsf{a} \wedge y_\mathsf{m} \subseteq x_\mathsf{a}, \mathsf{g}\colon \mathrm{anyLess}(y, x) \rangle \\
\mathrm{anyLess}(x, y) &\Leftrightarrow y_\mathsf{a} \neq \emptyset \wedge (x_\mathsf{m} = \emptyset \vee \min x_\mathsf{a} < \max y_\mathsf{a} \vee \max x_\mathsf{m} \leq \max y_\mathsf{a}) \\
\mathsf{Length}_{\mathcal{CI}}(x) &= \begin{cases} \{0\} & x_\mathsf{a} = \emptyset \\ [|x_\mathsf{m}|, \infty) & x_\mathsf{a} \neq \emptyset \end{cases} \\
\mathsf{IndexOf}_{\mathcal{CI}}(x, y) &= \begin{cases} \{0\} & y_\mathsf{a} = \emptyset \\ \{-1, 0\} & y_\mathsf{a} = x_\mathsf{a} \wedge |x_\mathsf{a}| = 1 \\ \{-1\} & y_\mathsf{m} \not\subseteq x_\mathsf{a} \\ [-1, \infty) & \text{otherwise} \end{cases} \\
\mathsf{IndexOf}_{\mathcal{CI},\mathcal{S}}(x, s) &= \begin{cases} \{0\} & |s| = 1 \wedge s[0] \in x_\mathsf{m} \wedge |x_\mathsf{a}| = 1 \\ [0, \infty) & |s| = 1 \wedge s[0] \in x_\mathsf{m} \wedge |x_\mathsf{a}| > 1 \\ \mathsf{IndexOf}_{\mathcal{CI}}(x, \alpha_{\mathcal{CI}}(s)) & \text{otherwise} \end{cases} \\
\mathsf{LastIndexOf}_{\mathcal{CI}}(x, y) &= \begin{cases} [0, \infty) & y_\mathsf{a} = \emptyset \\ \{-1\} & \mathrm{char}(s) \not\subseteq x_\mathsf{a} \\ [-1, \infty) & \text{otherwise} \end{cases} \\
\mathsf{LastIndexOf}_{\mathcal{CI},\mathcal{S}}(x, s) &= \begin{cases} [0, \infty) & |s| = 1 \wedge s[0] \in x_\mathsf{m} \\ \mathsf{LastIndexOf}_{\mathcal{CI}}(x, \alpha_{\mathcal{CI}}(s)) & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.18: $\mathcal{CI}$ operations

$$\mathrm{fromRE}(\mathsf{single}(S), p) = \left\langle \mathsf{m}\colon \begin{cases} \emptyset & |S| > 1 \\ S & |S| \leq 1 \end{cases}, \mathsf{a}\colon \begin{cases} \Sigma & p = \mathsf{o} \\ S & p = \mathsf{c} \end{cases} \right\rangle$$

$$\mathrm{fromRE}(\mathsf{begin}(), p) = \left\langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \begin{cases} \Sigma & p = \mathsf{o} \\ \emptyset & p = \mathsf{c} \end{cases} \right\rangle$$

$$\mathrm{fromRE}(\mathsf{end}(), p) = \mathrm{fromRE}(\mathsf{begin}(), p)$$

$$\mathrm{fromRE}(\mathsf{concat}(M_1, \ldots, M_n), p) = \left\langle \mathsf{m}\colon \bigcup_{i=1}^{n} x_{i\mathsf{m}}, \mathsf{a}\colon \bigcup_{i=1}^{n} x_{i\mathsf{a}} \right\rangle$$
$$\text{where } x_i = \mathrm{fromRE}(M_i, q)$$
$$\text{where } q = \begin{cases} \mathsf{c} & M_1 = \mathsf{begin}() \\ & \wedge M_n = \mathsf{end}() \\ p & \text{otherwise} \end{cases}$$

$$\mathrm{fromRE}(\mathsf{union}(M_1, \ldots, M_n), p) = \left\langle \mathsf{m}\colon \bigcap_{i=1}^{n} x_{i\mathsf{m}}, \mathsf{a}\colon \bigcup_{i=1}^{n} x_{i\mathsf{a}} \right\rangle$$
$$\text{where } x_i = \mathrm{fromRE}(M_i, p)$$

$$\mathrm{fromRE}(\mathsf{loop}(M, i, j), p) = \begin{cases} \mathrm{fromRE}(M, p) & i > 0 \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \mathrm{fromRE}(M, p)_{\mathsf{a}} \rangle & i = 0 \end{cases}$$

$$\mathrm{fromNRE}(\mathsf{single}(S)) = \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \Sigma \setminus S \rangle$$

$$\mathrm{fromNRE}(\mathsf{begin}()) = \bot$$

$$\mathrm{fromNRE}(\mathsf{end}()) = \bot$$

$$\mathrm{fromNRE}(\mathsf{empty}()) = \bot$$

$$\mathrm{fromNRE}(\mathsf{concat}(M_1, \ldots, M_n)) = \begin{cases} \bot & n = 0 \\ \mathrm{fromNRE}(M_1) & n = 1 \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \Sigma \rangle & n > 1 \end{cases}$$

$$\mathrm{fromNRE}(\mathsf{union}(M_1, \ldots, M_n)) = \left\langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \bigcap_{i=1}^{n} x_{i\mathsf{a}} \right\rangle$$
$$\text{where } x_i = \mathrm{fromNRE}(M_i)$$

$$\mathrm{fromNRE}(\mathsf{loop}(M, i, j)) = \begin{cases} \bot & i = 0 \\ \mathrm{fromNRE}(M) & i = 1 \\ \langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \Sigma \rangle & i > 1 \end{cases}$$

Figure 4.19: $\mathcal{CI}$ support for regular expressions, part 1

$$
\begin{aligned}
\mathrm{isMatch}(x, \mathsf{single}(S), \mathsf{o}) &= \langle \mathsf{f}\colon S = \emptyset \vee S \not\subseteq x_\mathsf{m}, \mathsf{t}\colon S \cap x_\mathsf{a} \neq \emptyset \rangle \\
\mathrm{isMatch}(x, \mathsf{single}(S), \mathsf{c}) &= \langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon S \cap x_\mathsf{a} \neq \emptyset \wedge x_\mathsf{m} \subseteq S \wedge |x_\mathsf{m}| \leq 1 \rangle \\
\mathrm{isMatch}(x, \mathsf{empty}(), \mathsf{o}) &= \mathsf{t} \\
\mathrm{isMatch}(x, \mathsf{empty}(), \mathsf{c}) &= \langle \mathsf{f}\colon x_\mathsf{a} \neq \emptyset, \mathsf{t}\colon x_\mathsf{m} = \emptyset \rangle \\
\mathrm{isMatch} \\ (x, \mathsf{concat}(M_1, \ldots, M_n), p) &= \begin{cases} \mathrm{isMatch}(\mathsf{concat}(M_2, \ldots, M_{n-1}), \mathsf{c}) \\ \qquad \text{where } M_1 = \mathsf{begin}() \wedge M_n = \mathsf{end}() \\ \left\langle \mathsf{f}\colon \mathsf{t}, \mathsf{t}\colon \bigwedge_{i=2}^{n-1} b_{i\mathsf{t}} \wedge x_\mathsf{m} \subseteq T \right\rangle \text{otherwise} \end{cases} \\
&\quad \text{where } T = \mathrm{fromRE}(\mathsf{concat}(M_1, \ldots, M_n), q)_\mathsf{a} \\
&\quad \text{where } q = \begin{cases} p & n = 1 \\ \mathsf{o} & n > 1 \end{cases} \\
\mathrm{isMatch} \\ (x, \mathsf{union}(M_1, \ldots, M_n), p) &= \left\langle \mathsf{f}\colon \bigwedge_{i=1}^{n} b_{i\mathsf{f}}, \mathsf{t}\colon \bigvee_{i=1}^{n} b_{i\mathsf{t}} \right\rangle \\
&\quad \text{where } b_i = \mathrm{isMatch}(x, M_i, p) \\
\mathrm{isMatch}(x, \mathsf{loop}(M, i, j), \mathsf{o})_\mathsf{t} &= d \vee i = 0 \\
\mathrm{isMatch}(x, \mathsf{loop}(M, i, j), \mathsf{c})_\mathsf{t} &= (d \wedge x_\mathsf{m} \subseteq \mathrm{fromRE}(M, \mathsf{c})_\mathsf{a}) \vee (i = 0 \wedge x_\mathsf{m} = \emptyset) \\
&\quad \text{where } d = \mathrm{isMatch}(x, M, \mathsf{o})_\mathsf{t} \\
\mathrm{isMatch}(x, \mathsf{loop}(M, i, j), \mathsf{c})_\mathsf{f} &= \begin{cases} x_\mathsf{a} \cap R \neq \emptyset & i = 0 \wedge j = \infty \\ x_\mathsf{a} \cap R \neq \emptyset \vee x_\mathsf{m} = \emptyset & i = 1 \wedge j = \infty \\ \mathsf{t} & \text{otherwise} \end{cases} \\
\mathrm{isMatch}(x, \mathsf{loop}(M, i, j), \mathsf{o})_\mathsf{f} &= \begin{cases} \mathsf{f} & i = 0 \\ x_\mathsf{m} \subseteq R & i = 1 \\ \mathsf{t} & i > 1 \end{cases} \\
&\quad \text{where } R = \mathrm{fromNRE}(M)_\mathsf{a}
\end{aligned}
$$

Figure 4.20: $\mathcal{CI}$ support for regular expressions, part 2

$$\begin{aligned}
\mathsf{Contains}_{\mathcal{CI}}(\alpha_{\mathcal{CI}}(S), x) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SU}}(\langle \mathsf{m}\colon x_m, \mathsf{a}\colon \Sigma\rangle)\\
\mathsf{Contains}_{\mathcal{CI},\mathcal{S}}(\alpha_{\mathcal{CI}}(S), s) = \mathsf{f} \wedge |s| = 1 &\Rightarrow S \subseteq \gamma_{\mathcal{SU}}(\langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \Sigma \setminus \{s[0]\}\rangle)\\
\mathsf{StartsWith}_{\mathcal{CI}}(\alpha_{\mathcal{CI}}(S), x) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SU}}(\langle \mathsf{m}\colon x_m, \mathsf{a}\colon \Sigma\rangle)\\
\mathsf{EndsWith}_{\mathcal{CI}}(\alpha_{\mathcal{CI}}(S), x) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SU}}(\langle \mathsf{m}\colon x_m, \mathsf{a}\colon \Sigma\rangle)\\
\mathsf{Equals}_{\mathcal{CI}}(\alpha_{\mathcal{CI}}(S), x) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SU}}(x)\\
\mathsf{IsEmpty}_{\mathcal{CI}}(\alpha_{\mathcal{CI}}(S)) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SU}}(\langle \mathsf{m}\colon \emptyset, \mathsf{a}\colon \emptyset\rangle)\\
\mathsf{IsMatch}_{\mathcal{CI}}(\alpha_{\mathcal{CI}}(S), r) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SU}}(\mathrm{fromRE}(\mathrm{regex}(r), \mathsf{o}))\\
\mathsf{IsMatch}_{\mathcal{CI}}(\alpha_{\mathcal{CI}}(S), r) = \mathsf{f} &\Rightarrow S \subseteq \gamma_{\mathcal{SU}}(\mathrm{fromNRE}(\mathrm{regex}(r)))
\end{aligned}$$

Figure 4.21: $\mathcal{CI}$ assume operations

a single character), as defined in Figure 4.21. The positive assumptions work the same also for StartsWith and EndsWith, but those operations do not provide any information if the result is negative. Assumptions about the IsMatch operation use the definitions in Figure 4.20. For IsEmpty, we know that no character is allowed, and for Equals, we can use the abstract element of the other argument.

4.5.5 Variants of Domains Based on Character Sets

Number of occurrences. Instead of requiring a character to occur one or more times, or to not occur at all, we might specify for each character how many times it should occur (using an integer abstraction such as intervals).

Multiple sets. If we allow multiple possible sets of characters, the analysis might be more precise, but we would have to figure out how to merge the sets to prevent the elements from growing too much.

4.6 Bricks

The Bricks domain is also defined by Costantini [34]. It represents a set of strings by a list of "bricks", where each brick specifies a set of allowed strings in a given part of the string and an interval saying how many times a string from that set might be used. The domain is parameterized by integers k_{L}, k_{I} and k_{S}, which limit the size of the elements in the widening operator (we use the set size limit also in the abstraction function).

4.6.1 Single Brick

First, we consider just a single brick. It is a set of string constants with the number of repetitions limited by an interval.

Definition 4.6.1 (Single brick domain)**.** $\mathcal{B} = \langle \{[S]^{m,n} : 0 \leq m \leq n \leq \infty, S \subseteq \Sigma^*\}, \sqsubseteq_{\mathcal{B}}, \sqcap_{\mathcal{B}}, \sqcup_{\mathcal{B}}, \bot_{\mathcal{B}}, \top_{\mathcal{B}}, \nabla_{\mathcal{B}}\rangle$. The domain operators are defined in Figure 4.22.

$$
\begin{aligned}
\gamma_{\mathcal{B}}\left([S]^{m,n}\right) &= \bigcup_{i=m}^{n} S^i \\
\alpha_{\mathcal{B}}(S) &= \begin{cases} [S]^{1,1} & |S| \leq k_{\mathsf{S}} \\ \top_{\mathcal{B}} & |S| > k_{\mathsf{S}} \end{cases} \\
[S]^{m,n} \sqcup_{\mathcal{B}} [T]^{k,l} &= [S \cup T]^{\min\{m,k\},\max\{n,l\}} \\
[S]^{m,n} \sqcap_{\mathcal{B}} [T]^{k,l} &= [S \cap T]^{\max\{m,k\},\min\{n,l\}} \\
[S]^{m,n} \nabla_{\mathcal{B}} [T]^{k,l} &= \begin{cases} \top_{\mathcal{B}} & |S \cup T| > k_{\mathsf{S}} \\ [S \cup T]^{0,\infty} & \max\{n,l\} - \min\{m,k\} > k_{\mathsf{I}} \\ [S]^{m,n} \sqcup_{\mathcal{B}} [T]^{k,l} & \text{otherwise} \end{cases} \\
\top_{\mathcal{B}} &= [\Sigma]^{0,\infty} \\
\bot_{\mathcal{B}} &= [\emptyset]^{1,1}
\end{aligned}
$$

Figure 4.22: $\mathcal{B}$ domain

A brick can precisely represent simple escaped strings. For example, $[\Sigma \setminus \{\sharp 0022, \sharp 005\mathrm{f}\} \cup \{\sharp 005\mathrm{f}\sharp 0022, \sharp 005\mathrm{f}\sharp 005\mathrm{f}\}]^{0,\infty}$ represents strings containing the characters only in escaped form.

4.6.2 Brick Lists

Brick list is simply a finite sequence of bricks.

Definition 4.6.2 (Bricks abstract domain)**.** $\mathcal{BR} = \langle \{\langle X_i \rangle_{i=1}^{n} : n \in \mathbb{N}, \forall i \in [1,n] : X_i \in \mathcal{B}\}, \sqsubseteq_{\mathcal{BR}}, \sqcap_{\mathcal{BR}}, \sqcup_{\mathcal{BR}}, \nabla_{\mathcal{BR}}, \top_{\mathcal{BR}} \bot_{\mathcal{BR}} \rangle$ The domain operators are defined in Figure .

The operators are defined in terms of the operators for single bricks, applying them to corresponding pairs of bricks from the lists. If one of the lists is shorter, it is extended by an algorithm that can be found in the original article [34]. The extension algorithm must add empty bricks to increase the length, but the algorithm also tries to insert them between other bricks so that equal bricks are aligned in the two lists if possible. There is also a set of normalization rules, which do not change the represented language, but simplify the list, including removing empty bricks. The rules are, informally:

- merge two bricks with the same sets of strings into one,
- split a brick with non-zero lower bound and a higher upper bound into a brick with constant bounds and a brick with a zero lower bound,
- expand the constants in a brick with constant repetitions,
- remove empty bricks.

$$
\begin{aligned}
\top_{\mathcal{BR}} &= \langle \top_{\mathcal{B}} \rangle \\
\bot_{\mathcal{BR}} &= \langle \bot_{\mathcal{B}} \rangle \\
X \sqcup_{\mathcal{BR}} Y &= \begin{cases} \text{extend}(X, Y) \sqcup_{\mathcal{BR}} Y & |X| < |Y| \\ X \sqcup_{\mathcal{BR}} \text{extend}(Y, X) & |X| > |Y| \\ \langle X_i \sqcup_{\mathcal{B}} Y_i : 1 \leq i \leq |X| \rangle & |X| = |Y| \end{cases} \\
X \nabla_{\mathcal{BR}} Y &= \begin{cases} \langle X_i \nabla_{\mathcal{B}} Y_i : 1 \leq i \leq |X| \rangle & |X| = |Y| \leq k_{\mathsf{L}} \\ \top_{\mathcal{BR}} & \max\{|X|, |Y|\} > k_{\mathsf{L}} \end{cases}
\end{aligned}
$$

Figure 4.23: $\mathcal{BR}$ domain

4.6.2.1 Drawbacks

Definition. The Bricks domain is not defined very precisely in the original article [34]. For example, it is not absolutely clear from the definition, whether infinity is a possible upper bound on the number of repetitions. This is obviously needed to construct the top brick, however, the definitions simply states that the bounds are integers. The definition also does not say that the sets of strings in the bricks must be finite.

Soundness. Secondly, the defined meet operation ($\sqcap_{\mathcal{BR}}$) is not sound. Consider these two lists of bricks:

$$
\begin{aligned}
X &= [\{\mathtt{ab}\}]^{0,1} [\{\mathtt{c}\}]^{1,1} \in \mathcal{BR} \\
Y &= [\{\mathtt{a}\}]^{1,1} [\{\mathtt{bc}\}]^{0,1} \in \mathcal{BR}
\end{aligned}
$$

There is no normalization rule that can be applied to X or Y, so these brick lists are normalized. According to the definition of $\gamma_{\mathcal{BR}}$, they represent these languages:

$$
\begin{aligned}
\gamma_{\mathcal{BR}}(X) &= \{\mathtt{abc}, \mathtt{c}\} \\
\gamma_{\mathcal{BR}}(Y) &= \{\mathtt{a}, \mathtt{abc}\} \\
\gamma_{\mathcal{BR}}(X) \cap \gamma_{\mathcal{BR}}(Y) &= \{\mathtt{abc}\}
\end{aligned}
$$

The intersection of the represented languages is non-empty. However, if we compute the meet of those brick lists, doing so by individual bricks results in bottom bricks as we show below, so the language of the meet is empty.

$$
\begin{aligned}
[\{\mathtt{ab}\}]^{0,1} \sqcap_{\mathcal{B}} [\{\mathtt{a}\}]^{1,1} &= [\emptyset]^{1,1} \\
[\{\mathtt{c}\}]^{1,1} \sqcap_{\mathcal{B}} [\{\mathtt{bc}\}]^{0,1} &= [\emptyset]^{1,1} \\
X \sqcap_{\mathcal{BR}} Y &= [\emptyset]^{1,1} [\emptyset]^{1,1} \\
\gamma_{\mathcal{BR}}(X \sqcap_{\mathcal{BR}} Y) &= \emptyset
\end{aligned}
$$

This means that the meet operator is not sound, $\gamma_{\mathcal{BR}}$ is not a complete meet morphism and it does not form a Galois connection.

The meet operation works if the bricks are aligned. That can happen when the two lists are the same or similar, or when the list extension algorithm succeeds in aligning the bricks in the right way.

We asked the authors of the article about this problem, however, they did not respond. The meet operation is not needed for the DFA, and in places where we would use the meet operator, we can use an overapproximation of the greatest lower bound (a trivial overapproximation is to return one of the operands).

Normalization. The normalization rules do not account for the empty string in S, which is equivalent to having the lower bound 0.

Theorem 4.6.3. *There is an equivalence of bricks* $\forall S \subseteq \Sigma^*, 0 \leq m \leq n\colon \gamma_{\mathcal{B}}([S \cup \{\epsilon\}]^{m,n}) = \gamma_{\mathcal{B}}\left([S \cup \{\epsilon\}]^{0,n}\right) = \gamma_{\mathcal{B}}\left([S]^{0,n}\right)$.

Proof. We prove a cycle of subset relations:

$$\gamma_{\mathcal{B}}([S \cup \{\epsilon\}]^{m,n}) = \bigcup_{i=m}^{n} S^i \subseteq \bigcup_{i=0}^{n} S^i = \gamma_{\mathcal{B}}\left([S \cup \{\epsilon\}]^{0,n}\right)$$

$$c \in \gamma_{\mathcal{B}}\left([S \cup \{\epsilon\}]^{0,n}\right) \Leftrightarrow \exists k \leq n, c_1, \ldots, c_k \in S \cup \{\epsilon\} : c = \prod_{i \in [1,k]} c_i$$

$$\Rightarrow \exists k \leq n, c_1, \ldots, c_k \in S \cup \{\epsilon\} : c = \prod_{i \in [1,k] \wedge c_i \neq \epsilon} c_i$$

$$\text{putting } l = k - |\{c_i : c_i = \epsilon\}| \wedge \forall i \leq l : d_i = c_{i + \left|\left\{c_j : j \leq i \wedge c_j = \epsilon\right\}\right|}$$

$$\Rightarrow \exists l \leq n, d_1, \ldots, d_l \in S\colon c = \prod_{i \in [1,l]} d_i$$

$$\Leftrightarrow c \in \gamma_{\mathcal{B}}\left([S]^{0,n}\right)$$

$$c \in \gamma_{\mathcal{B}}\left([S]^{0,n}\right) \Leftrightarrow \exists l \leq n, d_1, \ldots, d_l \in S\colon c = \prod_{i \in [1,l]} d_i$$

$$\Leftrightarrow \exists l \leq n, d_1, \ldots, d_l \in S\colon c = \prod_{i \in [1,l]} d_i \prod_{i \in [l+1,m]} \epsilon$$

$$\text{putting } k = \max\{m,l\} \wedge \forall i \leq l : c_i = d_i \wedge \forall l < i \leq k : c_i = \epsilon$$

$$\Rightarrow \exists m \leq k \leq n, c_1, \ldots, c_k \in S\colon c = \prod_{i \in [1,k]} c_i$$

$$\Leftrightarrow c \in \gamma_{\mathcal{B}}([S \cup \{\epsilon\}]^{m,n})$$

The first subset relation is trivial. In the second case, we remove all occurences of empty strings in the sequence $c_1, \ldots, c_k$ and shift the indices (by subtracting the number of preceding empty strings) to create a sequence $d_1, \ldots, d_l$ of non-empty strings, which has the same or shorter length and concatenates to the same string. In the third case, we add empty strings to the end of the sequence to satisfy the lower bound. □

The zero lower bound affects applicability of normalization rules: while $[\{\mathtt{a}, \epsilon\}]^{1,1}\,[\{\mathtt{b}, \epsilon\}]^{1,1}$ would be normalized to $[\{\mathtt{ab}, \mathtt{a}, \mathtt{b}, \epsilon\}]^{1,1}$, we cannot apply any rule to $[\{\mathtt{a}\}]^{0,1}\,[\{\mathtt{b}\}]^{0,1}$, which represents exactly the same language.

The normalization rules that expand repetitions and merge bricks can lead to exponential growth of the string sets. Consider for example the GUID language, represented precisely by a brick list:

$$X_{\text{guid}} = [C_{16}]^{8,8}\,[\text{-}]^{1,1}\,[C_{16}]^{4,4}\,[\text{-}]^{1,1}\,[C_{16}]^{4,4}\,[\text{-}]^{1,1}\,[C_{16}]^{4,4}\,[\text{-}]^{1,1}\,[C_{16}]^{12,12}$$

As all bricks of this list have constant number of repetitions, the normalization rules would concatenate all bricks into one brick containing all possible GUIDs as constants. This set would obviously not fit into memory. It seems reasonable not to apply those normalization rules in some cases.

It should also be noted that the normalization rule about merging bricks with the same sets should not be used if it would produce a brick that would be broken by the splitting rule (such as $[\{\mathsf{a}\}]^{0,1}\,[\{\mathsf{a}\}]^{1,1}$), otherwise the normalization algorithm would enter an infinite loop merging and splitting the bricks over and over again.

4.6.2.2 Operations

We define a few helper functions first, and then use them to define operations on bricks and brick lists in Figures 4.24 and 4.25. The minLen and maxLen compute the minimum and maximum length of strings represented by the brick lists. The char function returns the set of allowed charcters. The isConst and const functions determine whether the brick list represent a single string constant, and return it. The prefix and suffix try to extract the common prefix or suffix of the represented language.

To trim the list of bricks, trimStart uses a parameter, indicating, whether the current position is before (`b`) or after (`a`) the position where trimming ends. If the parameter is `u`, the characters may or may be not trimmed. The definition of trimEnd would be symmetric. The before function reutrns a list of bricks overapproximating a substring from the start up to an index from a specified interval. The definition of after would be similar, but keeping the parts of the string that are removed in before.

Several operations use the meet function with is the overapproximating version of the meet operator. We do not provide a formal definition here.

The operations defined in 4.26 are sound, but not necessarily the most precise. We had to choose some overapproximation that is efficiently computable. In some cases there are multiple possible ways how to do that.

4.6.2.3 Regular Expressions

Brick lists can naturally represent regular expressions containing non-nested concatenations of loops of unions. The formal definitions are shown in Figure 4.27.

Conversion from a regular expression to a list of bricks is defined recursively. There are three aditional paramters: l and r indicate, whether the left and right ends of the expression are tied to the start or end of the string by anchors. By default, the parameters are `o` (open), which means they are not tied. If an anchor is encountered in a cocatenation, the corresponding parameter is set to `c` (closed). The last parameter selects, whether the resulting list of brick should overapproximate (`t`) or underapproximate (`f`) the set of matching strings.

$$[S]^{m,n}\,[c := d] = [\{s[c := d]; s \in S\}]^{m,n}$$

$$\begin{aligned}
\operatorname{minLen}\left(\prod_{i=1}^{n} X_i\right) &= \sum_{i=1}^{n} \operatorname{minLen}(X_i) \\
\operatorname{minLen}([S]^{m,n}) &= \min\{|c| : c \in S\}\, m \\
\operatorname{maxLen}\left(\prod_{i=1}^{n} X_i\right) &= \sum_{i=1}^{n} \operatorname{maxLen}(X_i) \\
\operatorname{maxLen}([S]^{m,n}) &= \max\{|c| : c \in S\}\, n \\
\operatorname{char}\left(\prod_{i=1}^{n} X_i\right) &= \bigcup_{i=1}^{n} \operatorname{char}(X_i) \\
\operatorname{char}([S]^{m,n}) &= \bigcup_{c \in S} \operatorname{char}(c)
\end{aligned}$$

$$\operatorname{const}\left(\prod_{i=1}^{n} X_i\right) = \begin{cases} \bot & \exists i \colon \operatorname{const}(X_i) = \bot \\ \prod_{i=1}^{n} \operatorname{const}(X_i) & \forall i \colon \operatorname{isConst}(X_i) \\ \top & \text{otherwise} \end{cases}$$

$$\operatorname{const}([S]^{m,n}) = \begin{cases} \bot & m > n \vee (S = \emptyset \wedge m > 0) \\ s^m & S = \{s\} \wedge m = n \\ \epsilon & S = \{\epsilon\} \wedge m \leq n \\ \epsilon & m = n = 0 \\ \top & \text{otherwise} \end{cases}$$

$$\operatorname{isConst}\left(\prod_{i=1}^{n} X_i\right) \Leftrightarrow \forall i \colon \operatorname{isConst}(X_i)$$

$$\operatorname{isConst}([S]^{m,n}) \Leftrightarrow \operatorname{const}([S]^{m,n}) \in \Sigma^*$$

$$\operatorname{prefix}\left(\prod_{i=1}^{n} X_i\right) = \begin{cases} \epsilon & n = 0 \\ \operatorname{const}(X_1)\operatorname{prefix}(\prod_{i=2}^{n} X_i) & \operatorname{isConst}(X_1) \\ \operatorname{prefix}(X_1) & \text{otherwise} \end{cases}$$

$$\operatorname{prefix}([S]^{m,n}) = \begin{cases} \operatorname{lcp}(S) & m > 0 \\ \epsilon & m = 0 \end{cases}$$

$$\operatorname{suffix}\left(\prod_{i=1}^{n} X_i\right) = \begin{cases} \epsilon & n = 0 \\ \operatorname{suffix}\left(\prod_{i=1}^{n-1} X_i\right)\operatorname{const}(X_n) & \operatorname{isConst}(X_n) \\ \operatorname{suffix}([S]^{m,n}) & \text{otherwise} \end{cases}$$

$$\operatorname{suffix}([S]^{m,n}) = \begin{cases} \operatorname{lcs}(S) & m > 0 \\ \epsilon & m = 0 \end{cases}$$

Figure 4.24: Helper functions for $\mathcal{B}$ and $\mathcal{BR}$, part 1

$$\operatorname{trimStart}\left(\prod_{i=1}^{n} X_i, C\right) = \prod_{i=1}^{n} \operatorname{trimStart}(X_i, C, p_i)$$

$$\text{where } p_i = \begin{cases} \mathsf{b} & p_{i-1} = \mathsf{b} \wedge \operatorname{char}(X_i) \subseteq C \\ \mathsf{u} & p_{i-1} \neq \mathsf{a} \wedge \exists d \in S\colon \operatorname{char}(d) \subseteq C \\ \mathsf{a} & \text{otherwise} \end{cases}$$

$$\text{where } p_0 = \mathsf{b}$$

$$\operatorname{trimStart}([S]^{m,n}, C, \mathsf{b}) = \begin{cases} [\{\epsilon\}]^{0,0} & \operatorname{char}([S]^{m,n}) \subseteq C \\ [T]^{1,1}\,[S \cup T]^{m-1,n-1} & \exists d \in S\colon \operatorname{char}(d) \subseteq C \\ [T]^{1,1}\,[S]^{m-1,n-1} & \forall d \in S\colon \operatorname{char}(d) \nsubseteq C \end{cases}$$

$$\operatorname{trimStart}([S]^{m,n}, C, \mathsf{u}) = \begin{cases} [S \cup T]^{m,n} & \exists d \in S\colon \operatorname{char}(d) \subseteq C \\ [S \cup T]^{1,1}\,[S]^{m-1,n-1} & \forall d \in S\colon \operatorname{char}(d) \nsubseteq C \end{cases}$$

$$\text{where } T = \operatorname{trimStart}(S, C)$$

$$\operatorname{trimStart}([S]^{m,n}, C, \mathsf{a}) = [S]^{m,n}$$

$$\operatorname{trimStart}(S, C) = \{\mathsf{TrimStart}_{\mathcal{S}}(s, C)\colon s \in S\}$$

$$\operatorname{before}\left(\prod_{i=1}^{n} X_i, I\right) = \prod_{i=1}^{n} \operatorname{before}(X_i, [p_i, q_i])$$

$$\text{where } p_i = p_{i-1} - \operatorname{maxLen}(X_i)$$

$$\text{where } q_i = q_{i-1} - \operatorname{minLen}(X_i)$$

$$\text{where } p_0 = \min I, q_0 = \max I$$

$$\operatorname{ps}(S, i, j) = \{s[:k]\colon s \in S \wedge k \in [i, \min\{|s|, j\}]\}$$

$$\operatorname{before}([S]^{m,n}, [p, q]) = \begin{cases} [S]^{m,n} & p \geq nh \\ [S]^{k,l}\operatorname{before}\left([S]^{m-l,n-k}, [p - lh, q - kh]\right) & nh > p > g \\ [\operatorname{ps}(T, p, q)]^{1,1} & p < g \geq q \\ [\operatorname{ps}(T, p, q) \cup \operatorname{ps}(R, p, h)]^{1,1}\,[\operatorname{ps}(R, 0, h)]^{0,r} & \text{otherwise} \end{cases}$$

$$\text{where } k = \left\lfloor \frac{p}{h} \right\rfloor, l = \left\lfloor \frac{p}{g} \right\rfloor, r = \min\left\{n, \left\lceil \frac{q}{g} \right\rceil\right\} - 1$$

$$\text{where } R = \{s\colon s \in S \wedge |s| < q\}$$

$$\text{where } T = \{s\colon s \in S \wedge |s| \geq q\}$$

$$\text{where } g = \min\{|c| : c \in S\}, h = \max\{|c| : c \in S\}$$

Figure 4.25: Helper functions for $\mathcal{B}$ and $\mathcal{BR}$, part 2

$$
\begin{aligned}
\mathsf{Concat}_{\mathcal{BR}}(X,Y) &= XY \\
\mathsf{Insert}_{\mathcal{BR}}(X,I,Y) &= \mathrm{before}(X,I)Y\mathrm{after}(X,I) \\
\mathsf{ReplaceC}_{\mathcal{BR}}(K,f,t) &= \langle \mathsf{ReplaceC}_{\mathcal{B}}(K_i,f,t) : i \in [0,|K|] \rangle \\
\mathsf{ReplaceS}_{\mathcal{BR}}(X,Y,Z) &= \top_{\mathcal{B}} \\
\mathsf{Substring}_{\mathcal{BR}}(X,I,L) &= \mathrm{before}(\mathrm{after}(X,I),L) \\
\mathsf{SubstringEnd}_{\mathcal{BR}}(X,I) &= \mathrm{after}(X,I) \\
\mathsf{Remove}_{\mathcal{BR}}(X,I,L) &= \mathrm{before}(X,I)\mathrm{after}(\mathrm{after}(X,I),L) \\
\mathsf{RemoveEnd}_{\mathcal{BR}}(X,I) &= \mathrm{before}(X,I) \\
\mathrm{padding}(X,N,C) &= [C]^{\max(\min N - \mathrm{maxLen}(X),0),\max N - \mathrm{minLen}(X)} \\
\mathsf{PadLeft}_{\mathcal{BR}}(X,N,C) &= \mathrm{padding}(X,N,C)X \\
\mathsf{PadRight}_{\mathcal{BR}}(X,N,C) &= X\mathrm{padding}(X,N,C) \\
\mathsf{Trim}_{\mathcal{BR},\mathcal{S}}(X,s) &= \mathrm{trimEnd}(\mathrm{trimStart}(X,s),s) \\
\mathsf{TrimStart}_{\mathcal{BR},\mathcal{S}}(X,s) &= \mathrm{trimStart}(X,s) \\
\mathsf{TrimEnd}_{\mathcal{BR},\mathcal{S}}(X,s) &= \mathrm{trimEnd}(X,s) \\
\mathsf{IsEmpty}_{\mathcal{BR}}(X) &= \langle \mathsf{f}\colon \mathrm{maxLen}(X) \neq 0, \mathsf{t}\colon \mathrm{minLen}(X) = 0 \rangle \\
\mathsf{Contains}_{\mathcal{BR}}(X,Y) &= \begin{cases} \mathsf{t} & \mathrm{isConst}(Y) \wedge \\ & \quad \exists\, [S]^{m,n} \in X : S \subseteq \circledast\mathrm{const}(Y)\circledast \wedge m > 0 \\ \mathsf{f} & \mathrm{meet}(X, \top_{\mathcal{B}} Y \top_{\mathcal{B}}) = \bot_{\mathcal{BR}} \\ \top & \mathrm{otherwise} \end{cases} \\
\mathsf{StartsWith}_{\mathcal{BR}}(X,Y) &= \begin{cases} \mathsf{t} & \mathrm{isConst}(Y) \wedge \mathrm{prefix}(X) \in \mathrm{const}(Y)\circledast \\ \mathsf{f} & \mathrm{meet}(X, Y\top_{\mathcal{B}}) = \bot_{\mathcal{BR}} \\ \top & \mathrm{otherwise} \end{cases} \\
\mathsf{EndsWith}_{\mathcal{BR}}(X,Y) &= \begin{cases} \mathsf{t} & \mathrm{isConst}(Y) \wedge \mathrm{suffix}(X) \in \circledast\mathrm{const}(Y) \\ \mathsf{f} & \mathrm{meet}(X, \top_{\mathcal{B}} Y) = \bot_{\mathcal{BR}} \\ \top & \mathrm{otherwise} \end{cases} \\
\mathsf{Equals}_{\mathcal{BR}}(X,Y) &= \begin{cases} \mathsf{t} & \mathrm{isConst}(X) \wedge \mathrm{const}(X) = \mathrm{const}(Y) \\ \mathsf{f} & \mathrm{meet}(X,Y) = \bot_{\mathcal{BR}} \\ \top & \mathrm{otherwise} \end{cases} \\
\mathsf{Compare}_{\mathcal{BR}}(X,Y) &= \mathsf{Compare}_{\mathcal{PR}}(\mathrm{prefix}(X),\mathrm{prefix}(Y)) \\
\mathsf{Length}_{\mathcal{BR}}(X) &= [\mathrm{minLen}(X), \mathrm{maxLen}(X)] \\
\mathsf{IndexOf}_{\mathcal{BR}}(X,Y) &= \begin{cases} \{0\} & \mathrm{const}(Y) = \epsilon \\ \{-1\} & d < 0 \\ [-1,d] & \mathrm{otherwise} \\ & \quad \text{where } d = \mathrm{maxLen}(X) - \mathrm{minLen}(Y) \end{cases} \\
\mathsf{LastIndexOf}_{\mathcal{BR}}(X,Y) &= \begin{cases} [\max\{m-1,0\}, \max\{n-1,0\}] & \mathrm{const}(Y) = \epsilon \\ \quad \text{where } m = \mathrm{minLen}(X) \wedge n = \mathrm{maxLen}(X) & \\ [-1, \mathrm{maxLen}(X) - \mathrm{minLen}(Y)] & \mathrm{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.26: $\mathcal{BR}$ operations

For a set of characters, we create a single brick with a single repetition. If the expression is not tied to the end of the string, a top brick is inserted before and/or after the brick, to accomodate for the rest of the string. Concatenation is implemented by concatenating lists of bricks, but if all the bricks are constants, we simply create a new constant brick from them. For loops, we multiply the repeat counts of a brick if the inner expression converts to a single brick. Otherwise, the whole loop is overapproximated by the top brick.

To evaluate a match of a list of bricks against a regular expression, the regular expression is converted to a list of bricks. If the two lists do not represent any common string (which we find out by checking, whether an overapproximation of a meet of the two lists is bottom), there cannot be a match. On the other hand, the result cannot be false if the underapproximation of the regular expression contains all strings represented by the list of bricks. (lessEqual is an underapproximation of $\sqsubseteq_{\mathcal{BR}}$).

4.6.2.4 Assume

Support for assume statements using the $\mathcal{BR}$ domain is possible on all boolean operations. For the IsEmpty operation, both positive and negative assumptions are possible. If the value is assumed to be empty, it can be represented by an empty brick. If it is assumed not to be empty, the representing brick allows all characters, but requires the repeat count to be at least 1. For Contains, StartsWith, and EndsWith, we extract a constant part from the second argument (that part must be contained in the first argument) and add top bricks to represent the unknown part. Figure 4.28 shows the formal definitions.

4.6.3 Variants of the Bricks domain

The parameters for widening allow the Bricks domain to be tuned for speed or precision. Furthermore, the list extension algorithm and the normalization rules provide room for modifications. For example, we might define normalization rules that merge bricks that might overlap (overapproximating), to try to simplify the implemenation of the operations.

4.7 Graph-based Abstract Domains

It is common to represent languages by graph structures such as finite automata and grammars. We consider a single String Graph domain of that nature.

4.7.1 String Graphs

The String Graph domain ($\mathcal{SG}$) is defined by Costantini [34]. It is an adaptation of type graphs, which were used to analyze Prolog types. To represent strings, single characters and concatenation are considered to be functors.

In this thesis, we use a term notation with variables to describe string graphs, using $\mathsf{back}(x)$ to mean a backward edge to node x. For example, a string graph containing a cycle, that represents the language c^*, can be written as $x = \mathsf{or}(\mathsf{empty}(), \mathsf{concat}(\mathsf{char}(c), \mathsf{back}(x)))$.

$$\text{fromRE}(\mathsf{single}(S), l, r, a) = \text{wrap}(l)\,[S]^{1,1}\,\text{wrap}(r)$$

$$\text{wrap}(l) = \begin{cases} \top_{\mathcal{B}} & l = \mathsf{o} \\ \langle\rangle & l = \mathsf{c} \end{cases}$$

$$\text{fromRE}(\mathsf{loop}(M, i, j), l, r, a) = \begin{cases} \langle\rangle & \text{fromRE}(M) = \langle\rangle \\ [S]^{mi,nj} & \text{fromRE}(M) = [S]^{m,n} \\ \top_{\mathcal{B}} & |\text{fromRE}(M)| > 0 \wedge a \\ \bot_{\mathcal{B}} & |\text{fromRE}(M)| > 0 \wedge \neg a \end{cases}$$

$$\begin{aligned} &\text{fromRE} \\ &(\mathsf{concat}(M_1, \ldots, M_n), l, r, a) \end{aligned} = \text{fromConcat}(M_i, \ldots, M_j, l_0, r_0, a)$$

$$\text{where } l_0, i = \begin{cases} \mathsf{c}, 2 & M_1 = \mathsf{begin}() \\ l, 1 & \text{otherwise} \end{cases}$$

$$\text{where } r_0, i = \begin{cases} \mathsf{c}, n-1 & M_n = \mathsf{end}() \\ r, n & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\text{fromConcat} \\ &(M_1, \ldots, M_n, l, r, a) \end{aligned} = \begin{cases} [\{\prod_{i=1}^{n} s_i\}]^{1,1} & \forall i\colon Y_i = [\{s_i\}]^{1,1} \\ \prod_{i=1}^{n} Y_i & \text{otherwise} \end{cases}$$

$$\text{where } \forall i\colon Y_i = \text{fromRE}(M_i, l_i, r_i, a)$$

$$\text{where } l_1 = l \wedge \forall i > 1\colon l_i = \mathsf{c}$$

$$\text{where } r_n = r \wedge \forall i < n\colon r_i = \mathsf{c}$$

$$\begin{aligned} &\text{fromRE} \\ &(\mathsf{union}(M_1, \ldots, M_i), l, r, a) \end{aligned} = \begin{cases} \bot_{\mathcal{B}} & i > 1 \wedge \neg a \\ \bigsqcup_{i=0}^{n}{}_{\mathcal{BR}} \text{fromRE}(M_i, l, r, a) & \text{otherwise} \end{cases}$$

$$\mathsf{IsMatch}_{\mathcal{BR}}(X, r) = \left\langle \begin{array}{l} \mathsf{f}\colon \neg\text{lessEqual}(X, \text{fromRE}(\text{regex}(r), \mathsf{o}, \mathsf{o}, \mathsf{f})) \\ \mathsf{t}\colon \text{meet}(X, \text{fromRE}(\text{regex}(r), \mathsf{o}, \mathsf{o}, \mathsf{t})) \neq \bot_{\mathcal{BR}} \end{array} \right\rangle$$

Figure 4.27: $\mathcal{BR}$ support for regular expressions

$$\mathsf{IsEmpty}_{\mathcal{BR}}(\alpha_{\mathcal{BR}}(S)) = \mathsf{t} \Rightarrow S \subseteq \gamma_{\mathcal{BR}}\left([\{\epsilon\}]^{0,0}\right)$$

$$\mathsf{IsEmpty}_{\mathcal{BR}}(\alpha_{\mathcal{BR}}(S)) = \mathsf{f} \Rightarrow S \subseteq \gamma_{\mathcal{BR}}\left([\Sigma]^{1,\infty}\right)$$

$$\mathsf{Contains}_{\mathcal{BR}}(\alpha_{\mathcal{BR}}(S), X) = \mathsf{t} \Rightarrow S \subseteq \gamma_{\mathcal{BR}}\left(\top_{\mathcal{B}}\,[\{\text{const}(X)\}]^{1,1}\,\top_{\mathcal{B}}\right)$$

$$\mathsf{StartsWith}_{\mathcal{BR}}(\alpha_{\mathcal{BR}}(S), X) = \mathsf{t} \Rightarrow S \subseteq \gamma_{\mathcal{BR}}\left([\{\text{prefix}(X)\}]^{1,1}\,\top_{\mathcal{B}}\right)$$

$$\mathsf{EndsWith}_{\mathcal{BR}}(\alpha_{\mathcal{BR}}(S), X) = \mathsf{t} \Rightarrow S \subseteq \gamma_{\mathcal{BR}}\left(\top_{\mathcal{B}}\,[\{\text{suffix}(X)\}]^{1,1}\right)$$

$$\mathsf{Equals}_{\mathcal{BR}}(\alpha_{\mathcal{BR}}(S), X) = \mathsf{t} \Rightarrow S \subseteq \gamma_{\mathcal{BR}}(X)$$

$$\mathsf{IsMatch}_{\mathcal{BR}}(\alpha_{\mathcal{BR}}(S), r) = \mathsf{t} \Rightarrow S \subseteq \gamma_{\mathcal{BR}}(\text{fromRE}(\text{regex}(r), \mathsf{o}, \mathsf{o}, \mathsf{t}))$$

Figure 4.28: $\mathcal{BR}$ assume operations

$$
\begin{aligned}
x \sqcup_{\mathcal{SG}} y &= \mathsf{or}(x, y) \\
\gamma_{\mathcal{SG}}(x) &= \bigcup_{n=0}^{\infty} \mathrm{L}_{\mathcal{SG}}(x, n) \\
\alpha_{\mathcal{SG}}(S) &= \mathsf{or}(\mathsf{concat}(\mathsf{char}(c_i) \colon i \in [0, n-1]) \colon \langle c_0, c_1, \ldots, c_{n-1} \rangle \in S) \\
\top_{\mathcal{SG}} &= \mathsf{max}() \\
\bot_{\mathcal{SG}} &= \mathsf{bot}()
\end{aligned}
$$

$$
\mathrm{L}_{\mathcal{SG}}(x, n) = \begin{cases}
\{c\} & x = \mathsf{char}(c) \\
\prod_{i=1}^{k} \mathrm{L}_{\mathcal{SG}}(y_i, l) & x = \mathsf{concat}(y_1, \ldots, y_k) \\
\bigcup_{i=1}^{k} \mathrm{L}_{\mathcal{SG}}(y_i, l) & x = \mathsf{or}(y_1, \ldots, y_k) \\
\emptyset & x = \mathsf{bot}() \\
\Sigma^* & x = \mathsf{max}() \\
\emptyset & x = \mathsf{empty}() \\
\mathrm{L}_{\mathcal{SG}}(y, n-1) & x = \mathsf{back}(y) \wedge n > 0 \\
\emptyset & x = \mathsf{back}(y) \wedge n = 0
\end{cases}
$$

Figure 4.29: $\mathcal{SG}$ domain operators

Domain operators are defined in Figure 4.29. The concretization function [34] is defined using a system of recursive equations. Such definition can have multiple solutions, but the smallest one should be used. We present the definition using limited number of transitions over a backward edge. The language $\mathrm{L}_{\mathcal{SG}}(x, n)$ is the language of a string graph with root node x, limited to strings which are generated by using at most n backward edges on every path from the root node.

4.7.1.1 Drawbacks

The problem with the string graph domain is that, unlike in prolog types, where `concat/2` and `concat/3` would be separate types, results of concatenation of different numbers of strings are not distinguishable. This means that not all properties of type graphs are also applicable to string graphs. This problem is partially eliminated by normalization, but not fully. For example, we cannot use the meet operator to over-approximate operations such as Contains. Consider the following three string graphs: $x = \mathsf{concat}(\mathsf{max}(), \mathsf{char}(\mathsf{a}), \mathsf{max}())$, $y = \mathsf{concat}(\mathsf{char}(\mathsf{a}), \mathsf{max}())$ and $z = \mathsf{concat}(\mathsf{max}(), \mathsf{char}(\mathsf{b}), \mathsf{max}())$. No normalization rules apply. They all have non-empty intersection, for example $\mathsf{ab} \in x \cap y \cap z$. However, computing $x \sqcap_{\mathcal{SG}} y$ fails at the root node, because the labels of the concat nodes are different. For $x \sqcap_{\mathcal{SG}} y$, it fails on the character node.

4.7.1.2 Operations

Similarly to the bricks domain, we define a few helper functions on the nodes of the graphs in Figures 4.30 and 4.31. Then the operations can be defined as in Figure 4.32.

$$\mathrm{char}(x) = \begin{cases} c & x = \mathsf{char}(c) \\ \bigcup_{i=1}^{n} \mathrm{char}(y_i) & x = \mathsf{concat}(y_1, \ldots, y_n) \\ \bigcup_{i=1}^{n} \mathrm{char}(y_i) & x = \mathsf{or}(y_1, \ldots, y_n) \\ \emptyset & x = \mathsf{bot}() \vee x = \mathsf{back}(y) \\ \Sigma & x = \mathsf{max}() \\ \emptyset & x = \mathsf{empty}() \end{cases}$$

$$\mathrm{minLen}(x) = \begin{cases} 1 & x = \mathsf{char}(c) \\ \sum_{i=1}^{n} \mathrm{minLen}(y_i) & x = \mathsf{concat}(y_1, \ldots, y_n) \\ \min_{i=1}^{n} \mathrm{minLen}(y_i) & x = \mathsf{or}(y_1, \ldots, y_n) \\ \emptyset & x = \mathsf{bot}() \vee x = \mathsf{back}(y) \\ 0 & x = \mathsf{max}() \\ 0 & x = \mathsf{empty}() \end{cases}$$

$$\mathrm{maxLen}(x) = \begin{cases} 1 & x = \mathsf{char}(c) \\ \sum_{i=1}^{n} \mathrm{maxLen}(y_i) & x = \mathsf{concat}(y_1, \ldots, y_n) \\ \max_{i=1}^{n} \mathrm{maxLen}(y_i) & x = \mathsf{or}(y_1, \ldots, y_n) \\ \emptyset & x = \mathsf{bot}() \\ \infty & x = \mathsf{max}() \vee x = \mathsf{back}(y) \\ 0 & x = \mathsf{empty}() \end{cases}$$

Figure 4.30: Helper functions for $\mathcal{SG}$, part 1

$$\mathrm{const}(x) = \begin{cases} c & x = \mathsf{char}(c) \\ \prod_{i=1}^{n} \mathrm{const}(y_i) & x = \mathsf{concat}(y_1, \ldots, y_n) \\ \top & x = \mathsf{max}() \vee x = \mathsf{or}(y_1, \ldots, y_n) \vee x = \mathsf{back}(y) \\ \epsilon & x = \mathsf{empty}() \\ \bot & x = \mathsf{bot}() \end{cases}$$

$$\mathrm{prefix}(x) = \begin{cases} c & x = \mathsf{char}(c) \\ c_1 c_2 .. & x = \mathsf{concat}(y_1, \ldots, y_n) \\ \top & x = \mathsf{max}() \vee x = \mathsf{or}(y_1, \ldots, y_n) \vee x = \mathsf{back}(y) \\ \epsilon & x = \mathsf{empty}() \\ \bot & x = \mathsf{bot}() \end{cases}$$

$$\mathrm{replaceC}(x, f, t) = \begin{cases} \mathsf{char}(t) & x = \mathsf{char}(f) \\ \mathsf{concat}(\mathrm{replaceC}(y_i, f, t) \colon i \in [1, n]) & x = \mathsf{concat}(y_1, \ldots, y_n) \\ \mathsf{or}(\mathrm{replaceC}(y_i, f, t) \colon i \in [1, n]) & x = \mathsf{or}(y_1, \ldots, y_n) \\ x & \text{otherwise} \end{cases}$$

$$\mathrm{repeat}(x) = y$$
$$\text{where } y = \mathsf{or}(\mathsf{empty}(), \mathsf{concat}(x, \mathsf{back}(y)))$$

Figure 4.31: Helper functions for $\mathcal{SG}$, part 2

$$
\begin{aligned}
\mathsf{Concat}_{\mathcal{SG}}(x, y) &= \mathsf{concat}(x, y) \\
\mathsf{Insert}_{\mathcal{SG}}(x, I, y) &= \mathsf{concat}(\mathrm{before}(x, I), y, \mathrm{after}(x, I)) \\
\mathsf{ReplaceC}_{\mathcal{SG}}(x, C, D) &= \mathrm{replaceC}(x, C, \mathsf{or}(\mathsf{char}(d\colon\ d \in D))) \\
\mathsf{ReplaceS}_{\mathcal{SG}}(x, y, z) &= \mathsf{concat}(\mathrm{beforeC}(\mathrm{char}(y)), \mathsf{max}(), \mathrm{afterC}(\mathrm{char}(y))) \\
\mathsf{Substring}_{\mathcal{SG}}(x, I, L) &= \mathrm{before}(\mathrm{after}(x, I), L) \\
\mathsf{SubstringEnd}_{\mathcal{SG}}(x, I) &= \mathrm{after}(x, I) \\
\mathsf{Remove}_{\mathcal{SG}}(x, I, L) &= \mathsf{concat}(\mathrm{before}(x, I), \mathrm{after}(\mathrm{after}(x, I), L)) \\
\mathsf{RemoveEnd}_{\mathcal{SG}}(x, I) &= \mathrm{before}(x, I) \\
\mathsf{PadLeft}_{\mathcal{SG}}(x, n, c) &= \mathsf{concat}(\mathrm{repeat}(\mathsf{char}(c)), x) \\
\mathsf{PadRight}_{\mathcal{SG}}(x, n, c) &= \mathsf{concat}(x, \mathrm{repeat}(\mathsf{char}(c))) \\
\mathsf{Trim}_{\mathcal{SG},\mathcal{S}}(x, s) &= \mathrm{trimEnd}(\mathrm{trimStart}(x, \mathrm{char}(s)), \mathrm{char}(s)) \\
\mathsf{TrimStart}_{\mathcal{SG},\mathcal{S}}(x, s) &= \mathrm{trimStart}(x, \mathrm{char}(s)) \\
\mathsf{TrimEnd}_{\mathcal{SG},\mathcal{S}}(x, s) &= \mathrm{trimEnd}(x, \mathrm{char}(s)) \\
\mathsf{IsEmpty}_{\mathcal{SG}}(x) &= \langle \mathsf{f}\colon \mathrm{maxLen}(x) > 0, \mathsf{t}\colon \mathrm{minLen}(x) = 0 \rangle \\
\mathsf{Contains}_{\mathcal{SG}}(x, y) &= \begin{cases} \mathsf{t} & \mathrm{isConst}(y) \wedge \mathrm{containsS}(x, \mathrm{const}(y)) \\ \mathsf{f} & \mathrm{meet}(x, \mathsf{concat}(\mathsf{max}(), y, \mathsf{max}())) = \mathsf{bot}() \\ \top & \text{otherwise} \end{cases} \\
\mathsf{StartsWith}_{\mathcal{SG}}(x, y) &= \begin{cases} \mathsf{t} & \mathrm{isConst}(y) \wedge \mathrm{prefix}(x) \in \mathrm{const}(y)\circledast \\ \mathsf{f} & \mathrm{meet}(x, \mathsf{concat}(y, \mathsf{max}())) = \mathsf{bot}() \\ \top & \text{otherwise} \end{cases} \\
\mathsf{EndsWith}_{\mathcal{SG}}(x, y) &= \begin{cases} \mathsf{t} & \mathrm{isConst}(y) \wedge \mathrm{suffix}(x) \in \circledast\mathrm{const}(y) \\ \mathsf{f} & \mathrm{meet}(x, \mathsf{concat}(\mathsf{max}(), y)) = \mathsf{bot}() \\ \top & \text{otherwise} \end{cases} \\
\mathsf{Equals}_{\mathcal{SG}}(x, y) &= \begin{cases} \mathsf{t} & \mathrm{isConst}(x) \wedge \mathrm{const}(x) = \mathrm{const}(y) \\ \mathsf{f} & \mathrm{meet}(x, y) = \mathsf{bot}() \\ \top & \text{otherwise} \end{cases} \\
\mathsf{Compare}_{\mathcal{SG}}(x, y) &= \mathsf{Compare}_{\mathcal{PR}}(\mathrm{prefix}(x), \mathrm{prefix}(y)) \\
\mathsf{Length}_{\mathcal{SG}}(x) &= [\mathrm{minLen}(x), \mathrm{maxLen}(x)] \\
\mathsf{IndexOf}_{\mathcal{SG}}(x, y) &= \begin{cases} \{0\} & \mathrm{const}(y) = \epsilon \\ [-1, \mathrm{maxLen}(x) - \mathrm{minLen}(y)] & \text{otherwise} \end{cases} \\
\mathsf{LastIndexOf}_{\mathcal{SG}}(x, y) &= \begin{cases} [\max\{m-1, 0\}, \max\{n-1, 0\}] & \mathrm{const}(y) = \epsilon \\ \quad \text{where } m = \mathrm{minLen}(x) & \\ \quad \text{where } n = \mathrm{maxLen}(x) & \\ [-1, \mathrm{maxLen}(x) - \mathrm{minLen}(y)] & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.32: $\mathcal{SG}$ operations

$$\mathrm{wrap}(x, l, r) = \begin{cases} x & l = r = \mathsf{c} \\ \mathsf{concat}(x, \mathsf{max}()) & l = \mathsf{c} \wedge r = \mathsf{o} \\ \mathsf{concat}(\mathsf{max}(), x) & l = \mathsf{o} \wedge r = \mathsf{c} \\ \mathsf{concat}(\mathsf{max}(), x, \mathsf{max}()) & l = r = \mathsf{o} \end{cases}$$

$$\mathrm{fromRE}(\mathsf{single}(S), l, r, a) = \begin{cases} \mathsf{bot}() & S = \emptyset \\ \mathrm{wrap}(\mathsf{char}(c), l, r) & S = \{c\} \\ \mathrm{wrap}(\mathsf{or}(\mathsf{char}(c) \colon c \in S)) & \text{otherwise} \end{cases}$$

$$\mathrm{fromRE}(\mathsf{empty}(), l, r, a) = \mathrm{wrap}(\mathsf{empty}(), l, r)$$

$$\mathrm{fromRE}(\mathsf{loop}(M, i, j), l, r, a) = \begin{cases} \mathsf{bot}() & \neg a \\ \mathrm{wrap}(\mathrm{repeat}(\mathrm{fromRE}(M, \mathsf{c}, \mathsf{c}, \mathsf{t}))) & a \end{cases}$$

$$\begin{aligned} &\mathrm{fromRE} \\ &(\mathsf{union}(M_1, \ldots, M_n), l, r, a) \end{aligned} = \mathsf{or}(\mathrm{fromRE}(M_i, l, r) \colon i \in [1, n])$$

$$\begin{aligned} &\mathrm{fromConcat} \\ &(M_1, \ldots, M_i, l, r, a) \end{aligned} = \mathsf{concat}(\mathrm{fromRE}(M_i, l_i, r_i) \colon i \in [1, n])$$

$$\text{where } l_1 = l \wedge \forall i > 1 \colon l_i = \mathsf{c}$$

$$\text{where } r_n = r \wedge \forall i < n \colon r_i = \mathsf{c}$$

$$\mathrm{isMatch}(x, M) = \left\langle \begin{array}{l} \mathsf{f} \colon \neg \mathrm{lessEqual}(x, \mathrm{fromRE}(M, \mathsf{o}, \mathsf{o}, \mathsf{f})) \\ \mathsf{t} \colon \mathrm{meet}(x, \mathrm{fromRE}(M, \mathsf{o}, \mathsf{o}, \mathsf{t})) \neq \bot_{\mathcal{SG}} \end{array} \right\rangle$$

Figure 4.33: $\mathcal{SG}$ support for regular expressions

4.7.1.3 Regular Expressions

We convert regexes to string graphs. The mapping between string graphs and regex is defined very similarly as in the case of bricks, using the same three parameters l, r, a with the same meaning. not so straightforward as in the case of bricks. Concatenation and union translate directly to the concat() and or() nodes. Loops are overapproximated by loop in the graph, similar to those used in the padding operations. If the ends of the match are not tied to the end of the strings, max() nodes are inserted. The full definition is in Figure 4.33.

4.7.1.4 Assume

Assumptions are supported for the first argument of all boolean operations, by using the string graph for the second argument, possibly with adding concatenation to max() nodes to account for the unknown part. For IsEmpty, a graph with empty() root node is used. The formal definitions are in Figure 4.34.

$$
\begin{aligned}
\mathsf{IsEmpty}_{\mathcal{SG}}(\alpha_{\mathcal{SG}}(S)) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SG}}(\mathsf{empty}()) \\
\mathsf{Contains}_{\mathcal{SG}}(\alpha_{\mathcal{SG}}(S), x) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SG}}(\mathsf{concat}(\mathsf{max}(), x, \mathsf{max}())) \\
\mathsf{StartsWith}_{\mathcal{SG}}(\alpha_{\mathcal{SG}}(S), x) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SG}}(\mathsf{concat}(x, \mathsf{max}())) \\
\mathsf{EndsWith}_{\mathcal{SG}}(\alpha_{\mathcal{SG}}(S), x) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SG}}(\mathsf{concat}(\mathsf{max}(), x)) \\
\mathsf{Equals}_{\mathcal{SG}}(\alpha_{\mathcal{SG}}(S), x) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SG}}(x) \\
\mathsf{IsMatch}_{\mathcal{SG}}(\alpha_{\mathcal{SG}}(S), r) = \mathsf{t} &\Rightarrow S \subseteq \gamma_{\mathcal{SG}}(\mathrm{fromRE}(\mathrm{regex}(r), \mathsf{o}, \mathsf{o}, \mathsf{t}))
\end{aligned}
$$

Figure 4.34: $\mathcal{SG}$ assume operations

5 Implementation

We implemented the analysis of strings using the abstract domains described in Chapter 4 in Clousot.

Implementation of the analysis consisted of the following tasks:

- Adding a new analyzer that can be enabled from the command line.
- Implementing the abstract domains. The abstract domain to be used can be selected from the command line.
- Usage of the DFA algorithm and with the abstract semantics of string operations corresponding to the selected abstract domain.
- Parsing of regex strings in a way that can be used to extract the relevant string properties.
- Integrating the string analysis with other abstract domains already implemented in Clousot, so that they can seamlessly cooperate

This required changes at several locations in the original Code Contracts project. The upstream source code is available at `https://github.com/Microsoft/CodeContracts`. The modified code is attached (see Appendix A).

Apart from published research papers, there is no documentation of the internal APIs or detailed comments which would help navigate the source code of Clousot. Therefore, we used the existing string analysis implementation as a starting point and used other more developed analyses as a guide on how to implement the new features.

5.1 New Features in Clousot

The whole codebase of Code Contracts is very large, and only a small part was added or modified for the purposes of this thesis. We marked our changes by those comment lines:

```
// Modified by Vlastimil Dort (2015–2016)
```

or

```
// Created by Vlastimil Dort (2015–2016)
// Master thesis String Analysis for Code Contracts
```

5.1.1 String Abstraction

All the string abstract domains are represented by classes that implement a common interface, `IStringAbstraction` (see Figure 5.1). This interface defines the domain operators in a type-safe way and also inherits from `IAbstractDomain`, which is used by the algorithms in Clousot, but this interface is not type-safe and needs type casts.

The string operations are defined in `IStringOperations` and implemented in an inner class of each string abstraction class. The abstract domains implement many of the string operations listed in Section 2.2.4. The method signatures are

similar to the signatures of the concrete methods, but instead of string parameters, they take the abstract values. To allow more precise implementation for constant arguments (when the abstraction cannot precisely represent constants), we use a struct type which can contain either abstract values or constant strings. Integers are passed as intervals (`IndexInterval`) of possible values. Because all the integers are string indices, we do not distinguish between negative values, and there is also a special value for infinity (used for example to represent open intervals). The `CharInterval` class is used for parameters of type `char`.

For example, the $\mathsf{PadLeft}_{\mathcal{PR}}$ operation is implemented in method `Prefix PadLeft(Prefix,IndexInterval,CharInterval)`.

Operations that return `bool` in the concrete, have for each string parameter an additional parameter, which contains the variable corresponding to the argument. This variable is needed to construct the predicate that can be later used in assume statements.

Prefix and Suffix domains. The $\mathcal{PR}$ and $\mathcal{SU}$ domains are implemented as wrappers around a string value, which is the prefix or suffix.

Character Inclusion domain. The $\mathcal{CI}$ domain is implemented by the class `CharacterInclusion`. There is an bit array of allowed and mandatory characters. The `ICharacterClassification` interface represents the $\mathcal{CE}_f$ domain by returning the index to the bit array for each character.

Bricks. The `Bricks` class is a wrapper around a list of `Brick` objects. The implementation of widening, normalizaiton and list extension is delegated to the field of type `IBricksPolicy`.

String Graphs. String graphs are constructed from nodes. Each type of node has its class in a class hierarchy, except $\mathsf{empty}()$, which we represent as a concatenation of zero child nodes. Most of the operations on the string graphs are implemented using the visitor pattern. The visitor base class is generic, with a result type parameter and data type parameter. The visitor caches the results for each node to allow handling backward edges.

The source codes of the abstractions are in individual files in the folder `AnalysisInfrastructure/Abstract Domains/String` (see Section 2.4.3 and Appendix A).

5.1.2 The Main Abstract Domain

The abstract domain used for the string analysis is `StringAbstractDomain`. The selected type of the string abstraction is a generic parameter of this class. The implementation of operations for that abstraction is passed as an argument to the constructor.

For each supported string operation, there is a method, which takes as arguments the expressions that represents the arguments of the called

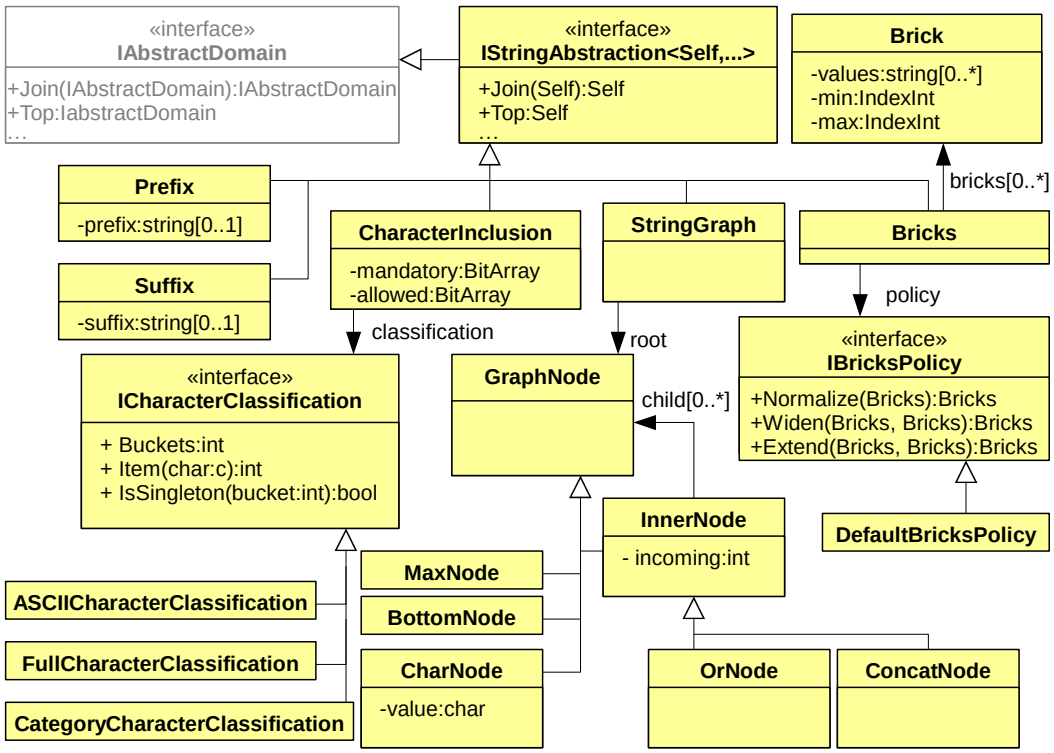

Figure 5.1: Class diagram of string abstractions

method and also a target expression, where the return value is assigned. For example, the PadLeft operation corresponds to the method `void PadLeft(Expression,Expression,Expression,Expression)`.

Clousot does not decode expressions containing string operations, and we chose not to extend it with this feature. Therefore for string expressions, we only recognize expressions that are string constants or variables that were previously assigned to and we know the corresponding abstract value. Then we call the implementation of the operation for the selected abstract domain using the abstract values, and assign the result abstract value to the target variable.

The abstract domain represents a mapping from variables to elements of the string domain. This is handled by the class `FunctionalAbstractDomain`, which is one of the generic abstract domains provided by Clousot.

Because we do not decode string expressions, in order to support assert and assume statements, we also keep a mapping from variables to boolean abstract values. The main abstract domain is actually a reduced product of functional domains from variables to string abstract values and to abstract predicates. This is achieved using the generic class `ReducedCartesianAbstractDomain`. The predicates can either be boolean constants (`true` or `false`) implemented by `FlatPredicate`, or it can be a `StringAbstractionPredicate`, which contains a variable together with two string abstract values, `trueAbstraction` and `falseAbstrations`. The meaning of this construct can be explained as follows: if the predicate is `true`, then the variable must contain only values specified by `trueAbstraction`. If the predicate is `false`, then the variable can only contain values represented by `falseAbstraction`. When an assume statement is reached, we now know that the assumed predicate must be `true` or `false`, so we assign

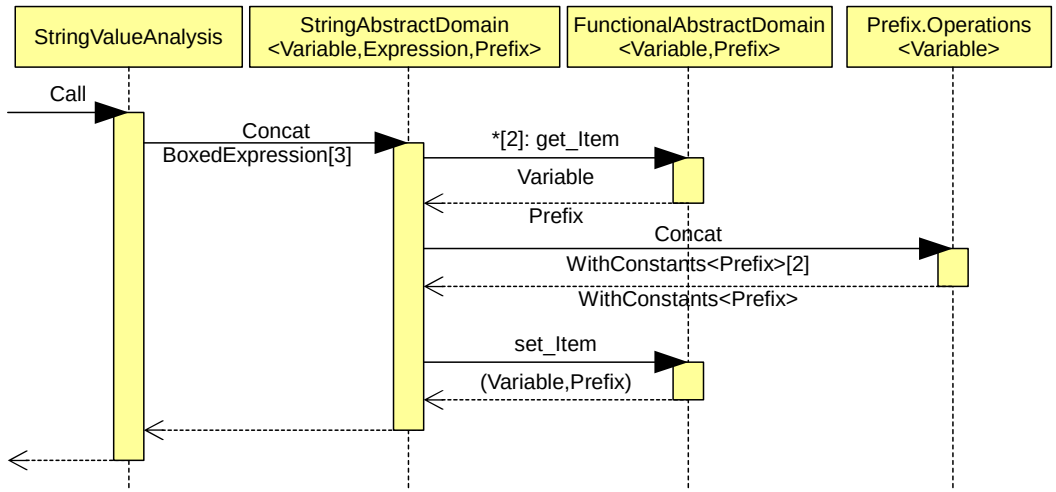

Figure 5.2: Operation call sequence diagram

the corresponding abstract value to the variable.

The assumptions are handled by the `TestTrue` and `TestFalse` methods. We use an expression visitor to decode boolean expressions. The `SimpleTestVisitor` class uses subclasses of `TestTrueVisitor` and `TestFalseVisitor` that are provided by Clousot.

The source code is in the file `AnalysisInfrastructure/AbstractDomains/String/StringAbstractDomain.cs`, loosely based on the previous implementation.

The diagram in Figure 5.2 shows a sequence of calls that handles a operation transfer function. The framework calls the `Call` method of the analysis, which recognizes the operation and calls the corresponding method of the abstract domain. There, the current values of the expressions are evaluated and the corresponding abstract implementation of the operation is applied on those values. Then the new value is stored to the target variable in the domain.

5.1.3 String Analysis

The string analysis `StringValueAnalysis` inherits from `GenericValueAnalysis`. It is handed a factory object that is used to construct the top value of `StringAbstractDomain`, which is then used to construct all the other elements.

The analysis contains visitor methods for IL instructions, which are called by the DFA algorithm according to the CFG. The `Call` method handles the transition for method calls. If a call to method of class `System.String` is detected, then the method signature is compared to the known signatures, and if the method is supported, the arguments are converted to `BoxedExpression`s and the corresponding method is called on the current `StringAbstractDomain`. Similar approach is used to handle methods of `Regex` and `StringBuilder`.

The `FactQuery` method is used to check whether the assertions have been proven. The `FactBase` provides the basic answers by querying the `StringAbstractDomain` for the predicate assigned to the boolean variable at the fixed point.

The code of this class is in the file `Particular Analysis/Analyzers/String`

`Analysis/StringAnalysis.cs` and is heavily based on the previous implementation.

5.1.4 String Analyzer

The `Analyzers.Strings` class acts as an entry point to the string analysis. Clousot finds the analyzer by reflection and enables it if an option of the same name is specified on the command line. The class also defines the options of the analysis as a class that is filled from the command line options by reflection.

We implemented a single analyzer, which has an option selecting the string abstract domain. The code is in the file `Analyzers\String Analysis\StringAnalysis.cs` and is heavily based on the previous implementation.

5.1.5 Combining with Other Abstract Domains

The cooperation with other abstract domains had to be done by making a plug-in to the analysis of arrays. We added a new plug-in, `StringWrapperPlugin`, based on the code of `EnumAnalysisPlugin` in the folder `Particular Analysis/Analyzers/Additional Analyses`. The plug-in handles the transition function by taking the current `ArrayState`, containing abstract values of all the running analyses. That may include array, non-null, numerical, and string, or other analyses. The plugin selects the string element from the array state in the IL transition handlers, then evaluates the transition on the string part using `StringValueAnalysis` and updates the array state with the new value. For the `Call` transition, it also queries and updates the numerical and non-null domains, because some string operations take or return integers, or handle `null` values in a special way. The `StringAbstractDomain` handlers for operations taking or returning integers have a parameter of type `NumericalAbstractDomain`, which is used to get or set the intervals for integer arguments or integer return values of the methods. If the string analysis runs in isolation, this parameter is `null`. The non-null information is wrapped using the `INullQuery` interface.

We modified the array analysis so that the string analysis is instantiated and propagated to where the plug-ins are set up (the `SetUpAdditionalAnalyses` method). We added a new index to the `ArrayState` to keep the string abstract domain element along with the other elements.

The last thing we needed to do, was to add the string analysis as a dependency of the array analysis in `ClousotMain.InitAndCheckArrayAnalysisDependences` (`ParticularAnalysis/ClousotMain/ClousotMain.cs`).

5.1.6 Regex

To support regular expressions, we implemented a recursive descent parser, which converts the regex string to an AST. The parser is located in a new project `Regex` in `AnalyisInfastrucutre`. The AST can be traversed using the visitor pattern (`Visitor`). Because we support only a subset of possible regular expressions, the `SimpleRegexVisitor` checks that the regular expressions consists of supported operator and provides a simpler interface to the implemeters.

5.2 Tests

The Code Contracts codebase does not contain unit tests. However, we used unit tests to test the string abstract domains, the implementation of operators, and that the operations have the expected output. The tests can be found in the `Regressions/Clousot/StringDomainUnitTests` and `Regressions/Clousot/RegexUnitTests` projects)

5.3 System Requirements

The code implemented in this thesis do not use any platform specific features. The running platform is limited by the implementation of the Code Contracts itself. The following system configuration or higher versions are supported:

- Windows 7
- Visual Studio 2013
- .NET Framework 4.5

6 Evaluation

We ran a few experiments on small benchmarks to evaluate the performance and precision of the implemented analysis with each string abstract domain. We do not know about any project that would use string properties in contracts, so we could not run our analysis on existing production code. Instead, we put together several example programs and run the analysis on them.

Test code. The code of the test programs and libraries is attached (Appendix A). Each test is in a separate project in the `src\Demos\Strings` directory.

The `PrefixTests`, `SuffixTests` and `CharacterInclusionTests` projects are designed specifically to test the code patterns that can be analyzed by the respective abstract domains. The class `Proven` contains methods with contracts or assert/assume statements, that should be proven by running the analysis with the abstract domain. On the other hand, the class `Unproven` contains assertions that should not be proven – either because they do not hold, or because the abstract domain is not precise enough to prove that property. The classes `IntegerProven` and `IntegerUnproven` contain tests involving integer properties. Therefore, the assertions in `IntegerProven` should only be proven when the array analysis is enabled.

The test `Properties` is an example application containing methods that require the arguments to have certain properties, similar to the ones described in Section 3.1.

The test `QueryGeneration` is a program that generates a SQL query using provided values and demonstrates the expected use of preconditions, postconditions and invariants regarding string values. The `StringManipulation` program performs string manipulation operations to transform an input into a derived output.

Running the tests. The domains are tested by running the whole analysis on the compiled binary file (`.exe` or `.dll`). We launched Clousot separately for each abstract domain, and also for a combination of the string domain with array analysis. We used the built-in time measurement feature of Clousot to measure the overall time of the analysis and the time spent per method.

To run the experiments, the `run.bat` batch file was used, with command line arguments specifying the selected abstract domain, the name of the test project, and a flag determining whether the array analysis should be run.

Results. The results of the experiments are in Tables 6.1 and 6.2. The table 6.1 shows the number of validated assertions[1] in the tests for the individual abstract domains. The domains were for comparison also run on tests designed for other domains.

Table 6.2 shows results for all the example programs. The Validated columns contains the number of assertions that were proven by the analysis, while the Uproven column shows the number of unproven assertions. The sum of the two numbers are higher for the analysis combined with arrays, because the other

[1] Unreachable assertions are counted as validated.

Domain	Test		
	Prefix	**Suffix**	**CharacterInclusion**
Prefix	46	8	0
Prefix+arrays	56	8	2
Suffix	8	34	1
Suffix+arrays	13	34	3
Character Inclusion	10	10	16
CI+arrays	15	11	26
Bricks	39	26	7
BR+arrays	43	26	7
String graphs	33	13	7
SG+arrays	39	13	9

Table 6.1: Evaluation of abstract domains on `PrefixTests`, `SuffixTests` and `CharacterInclusionTests`

analyses add implicit assertions about array and integer operations. The last two columns are the times reported by Clousot.

The results show that the Prefix and Suffix domains are similarly fast. The Character Inclusion domain working on the full range of characters is significantly slower. However, this can be improved by only considering ASCII characters (using a character set abstraction that treats all non-ascii characters as a single character). This domain has exactly the same results on the test programs.

The Bricks domain has reasonable results on all tests. The String Graph domains is a bit slower, but does not prove more assertions than the Bricks domain.

Combining the string analysis with array analysis makes it approximately 1 to 1.5 times slower and improves the result in almost all cases.

Overall, the tests do not show massive differences between the strength of the implemented abstract domains on the test programs. However, if we had to recommend the best abstract domain for a practical use, the Character Inclusion domain on ASCII characters and the Bricks domain seem to reasonable.

Domain	Validated	Unproven	Time	Time / method
Prefix	47	22	7.786 s	337 ms
Prefix+arrays	55	27	13.571 s	590 ms
Suffix	47	22	7.821 s	340 ms
Suffix+arrays	55	27	13.286 s	577 ms
Character Inclusion	56	13	1:07 min	2944 ms
CI+arrays	64	18	1:34 min	4260 ms
CI-ASCII	56	13	28.809 s	1252 ms
CI-ASCII+arrays	64	18	31.817 s	1383 ms
Bricks	53	16	34.932 s	1518 ms
BR+arrays	61	21	36.119 s	1570 ms
String graphs	50	19	45.190 s	1964 ms
SG+arrays	58	24	45.258 s	1967 ms

Table 6.2: Evaluation of abstract domains on `Properties`, `StringManupulation` and `QueryGeneration`

7 Conclusion

We achieved the goals stated in Section 1.2. First, we summarized the string data types and operations in .NET that can be analyzed, and proposed a means of specifying string properties through usage of the `string` class methods and the `Regex.IsMatch` method to achieve a limited support for regular expressions. We took abstract domains for strings that have been already published in a research paper, and defined the abstract semantics for a wide range of string operations available in .NET. For each domain, we also identified the operations that can be used to specify the properties expressed by that domain, and proposed a way to extract the properties from parsed regular expressions. We discovered problems with the definitions of the more complex abstractions, and worked around them.

We added support for string analysis to Code Contracts by implementing the string abstract domains in Clousot, allowing it to do automated checking of string properties. The task was complicated by the verbosity of the C# source code and by the lack of a documented API. The implemenation is general and can be easily extended by adding new abstract domains.

7.1 Future Work

Adding more string abstract domains to select from is one of the possible extensions of this work. The domains may be based on the ones described here with some modifications or others based on completely different principles. If necessary, support for more string operations may be added, particularly for methods working with arrays or more complex features of regular expressions, such as replacing or group matching. Interaction with arrays and `IEnumerable` might be improved. The operations that are overapproximated may be defined with a better precision.

This work enabled static checking of new properties, which could be used by adding contracts to APIs of the Framework Class Library and most importantly by adding contracts to projects already using Code Contracts. For easy use of the new analysis, the corresponding new options should be integrated into the Visual Studio extension.

If static analysis of contracts involving strings is adopted by users, the abstract domains should be evaluated on real programs.

Bibliography

[1] Reis, Gabriel Dos et al. *Simple Contracts for C++* [online]. April 2015. [Accessed 22. 3. 2016]. Available from: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/n4415.pdf.

[2] stephentoub [Toub, Stephen] et. al. *Proposal: Method Contracts* [online]. [Accessed 22. 3. 2016]. Available from: https://github.com/dotnet/roslyn/issues/119.

[3] *Code Contracts* [online]. Microsoft. [Accessed 24. 7. 2016]. Available from: http://research.microsoft.com/en-us/projects/contracts/.

[4] *Code Contracts* [online]. [Accessed 24. 7. 2016]. Available from: http://github.com/Microsoft/CodeContracts.

[5] Fähndrich, Manuel and Logozzo, Francesco. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software*. Springer, 2010. p. 10–30.

[6] Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[7] *C# Language Specification Version 5.0* [online]. [Accessed 9. 7. 2014]. Available from: http://www.microsoft.com/en-us/download/confirmation.aspx?id=7029.

[8] *.NET Framework Class Library* [online]. [Accessed 9. 7. 2014]. Available from: http://msdn.microsoft.com/en-us/library/gg145045%28v=vs.110%29.aspx.

[9] *Unicode 8.0.0* [online]. Unicode, Inc., June 2015. [Accessed 24. 7. 2016]. Available from: http://unicode.org/versions/Unicode8.0.0/.

[10] *Unicode Character Encoding Stability Policies* [online]. Unicode, Inc., April 2015. [Accessed 10. 4. 2016]. Available from: http://www.unicode.org/policies/stability_policy.html.

[11] Skeet, Jon. *When is an identifier not an identifier? ...* [online]. December 2014. [Accessed 22. 3. 2016]. Available from: http://codeblog.jonskeet.uk/2014/12/01/when-is-an-identifier-not-an-identifier-attack-of-the-mongolian-vowel-separator/.

[12] Davis, Mark and Whistler, Ken. Unicode Script Property. UAX #24, Unicode, Inc., June 2015. Available from: http://unicode.org/reports/tr24/tr24-24.html.

[13] Skeet, Jon. *When is a string not a string?* [online]. November 2014. [Accessed 22. 3. 2016]. Available from: http://codeblog.jonskeet.uk/2014/11/07/when-is-a-string-not-a-string/.

[14] *.NET Compiler Platform ("Roslyn")* [online]. Microsoft. [Accessed 24. 7. 2016]. Available from: https://roslyn.codeplex.com/wikipage?title=Language%20Feature%20Status&referringTitle=Documentation.

[15] Skeet, Jon. *The BobbyTables culture* [online]. August 2014. [Accessed 22. 3. 2016]. Available from: http://codeblog.jonskeet.uk/2014/08/08/the-bobbytables-culture/.

[16] Lippert, Eric. *String concatenation behind the scenes* [online]. [Accessed 20. 3. 2016]. Available from: http://ericlippert.com/2013/06/17/string-concatenation-behind-the-scenes-part-one/.

[17] *Regular Expression Language - Quick Reference* [online]. Microsoft. [Accessed 1. 6. 2016]. Available from: https://msdn.microsoft.com/en-us/library/az24scfc%28v=vs.110%29.aspx.

[18] Davis, Mark and Heninger, Andy. Unicode Regular Expressions. UTS #18, Unicode Consortium, November 2013. Available from: http://www.unicode.org/reports/tr18/tr18-17.html.

[19] *Code Contracts User Manual* [online]. Microsoft Corporation. [Accessed 24. 7. 2016]. Available from: http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf.

[20] Cerf, V.G. ASCII format for network interchange. STD 80, RFC Editor, October 1969. Available from: http://www.rfc-editor.org/rfc/rfc20.txt. doi: 10.17487/RFC0020.

[21] Josefsson, S. The Base16, Base32, and Base64 Data Encodings. RFC 4648, RFC Editor, October 2006. Available from: http://www.rfc-editor.org/rfc/rfc4648.txt. doi: 10.17487/RFC4648.

[22] Berners-Lee, Tim; Fielding, Roy T. and Masinter, Larry. Uniform Resource Identifier (URI): Generic Syntax. STD 66, RFC Editor, January 2005. Available from: http://www.rfc-editor.org/rfc/rfc3986.txt. doi: 10.17487/RFC3986.

[23] Braden, R. Requirements for Internet Hosts - Application and Support. STD 3, RFC Editor, October 1989. Available from: http://www.rfc-editor.org/rfc/rfc1123.txt. doi: 10.17487/RFC1123.

[24] Hinden, R. and Deering, S. IP Version 6 Addressing Architecture. RFC 4291, RFC Editor, February 2006. Available from: http://www.rfc-editor.org/rfc/rfc4291.txt. doi: 10.17487/RFC4291.

[25] Kawamura, S. and Kawashima, M. A Recommendation for IPv6 Address Text Representation. RFC 5952, RFC Editor, August 2010. Available from: http://www.rfc-editor.org/rfc/rfc5952.txt. doi: 10.17487/RFC5952.

[26] Klensin, J. Application Techniques for Checking and Transformation of Names. RFC 3696, RFC Editor, February 2004. Available from: http://www.rfc-editor.org/rfc/rfc3696.txt. doi: 10.17487/RFC3696.

[27] *How To: Use Regular Expressions to Constrain Input in ASP.NET* [online]. Microsoft. [Accessed 11. 5. 2016]. Available from: https://msdn.microsoft.com/en-us/library/ff650303.aspx.

[28] *RegularExpressionValidator.ValidationExpression Property* [online]. Microsoft. [Accessed 11. 5. 2016]. Available from: https://msdn.microsoft.com/en-us/library/system.web.ui.mobilecontrols.regularexpressionvalidator.validationexpression.aspx.

[29] Bray, Tim et al. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C recommendation, W3C, November 2008. Available from: http://www.w3.org/TR/2008/REC-xml-20081126/.

[30] Christensen, Aske Simon; Møller, Anders and Schwartzbach, Michael I. Precise Analysis of String Expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, p. 1–18, Berlin, Heidelberg, 2003. Springer-Verlag. Available from: http://dl.acm.org/citation.cfm?id=1760267.1760269. ISBN 3-540-40325-6.

[31] Kieżun, Adam et al. HAMPI: A solver for string constraints. In *ISSTA 2009, Proceedings of the 2009 International Symposium on Software Testing and Analysis*, Chicago, IL, USA, July 21–23, 2009. Available from: https://people.csail.mit.edu/akiezun/issta54-kiezun.pdf.

[32] Zheng, Yunhui; Zhang, Xiangyu and Ganesh, Vijay. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, p. 114–124, New York, NY, USA, 2013. ACM. Available from: http://doi.acm.org/10.1145/2491411.2491456. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491456.

[33] Šuta, Matej. String abstract domains. Master's thesis, supervisor: Karel Klíč, Masarykova univerzita, Fakulta informatiky, Brno, 2013. Available from: http://is.muni.cz/th/389959/fi_m/.

[34] Costantini, Giulia; Ferrara, Pietro and Cortesi, Agostino. A suite of abstract domains for static analysis of string values. *Software: Practice and Experience*. 2015, 45, 2, p. 245–287. Available from: http://onlinelibrary.wiley.com/doi/10.1002/spe.2218/abstract. ISSN 1097-024X. doi: 10.1002/spe.2218.

List of Figures

A Contents of the CD

- `README.txt` Instructions on how to build and run the program
- `src\CodeContracts\Microsoft.Research\` Source code of Code Contracts
  - `AbstractInterpretation\Abstract Domains\` The `Abstract Domains` project
    - `Strings\` String abstract domains, operations and related utility classes
  - `Analyzers\` The `Analyzers` project
    - `Additional Analyses\` Plug-ins for the array analysis
    - `Array Analysis` Analysis of arrays allowing plug-ins
    - `String Analysis` Analysis of strings
  - `Clousot\` The executable project
  - `ClousotMain\` Implementation of Clousot
  - `Regex\` Parsing regular expressions
  - `RegressionTest\StringDomainUnitTests` Unit tests for abstract domains
  - `RegressionTest\RegexUnitTests\` Unit tests for regular expressions
- `src\CodeContracts\Demo\Strings` Test and example program source codes
- `bin\Clousot` Executable static checker
- `bin\Examples` Compiled example programs and libraries
- `doc\generated.chm` Generated documentation from the C# source code
- `doc\text.pdf` Text of the thesis

B User Guide

B.1 Writing String Contracts

You can write string expressions in Contracts, namely in the methods of the `Contract` class such as `Assert`, `Assume`, `Requires`, `Ensures` and `Invariant`. If you enable runtime contract checking, they will be checked as expected. When using the static checker, they might not be proven, also depending on the string abstract domain you choose.

Make sure you use `StringComparison.Ordinal` or invariant culture where appropriate. If you use culture-specific features, the results may vary depending on the culture, and cannot be reliably statically checked.

B.2 Running from the Command Line

To run the string analysis from the command line, add `-strings:domain=`*`name`* to the command arguments of `Clousot.exe`, where *`name`* is one of the names listed in Table B.1. By default, only failed assertions are printed to the output. To see the results for all assertions including the proven, add `-show:validations`. The help for other command line switches can be displayed by `-help` option.

To run the string analysis together with array and artihmetic analysis, also add those two options: `-array -bounds`.

To show the trace of the DFA algorithm including the abstract elements, add `-trace:dfa` to the list of options.

Name	Abstract domain
`prefix`	Prefix
`suffix`	Suffix
`characterinclusionfull`	Character Inclusion with individual characters
`characterinclusionascii`	Character Inclusion for ASCII characters
`bricks`	Bricks
`stringgraphs`	String Graphs

Table B.1: Abstract domain command line names