

ETH ZÜRICH

MASTER'S THESIS REPORT

**Static Checking of TouchDevelop
Programs against Web Service
Specifications**

Author:
Pascal Zimmermann

Supervisor:
Lucas Brutschy
Prof. Dr. Peter Müller

Chair of Programming Methodology
Department of Computer Science

July 16, 2014

Abstract

Mobile devices such as tablet computers and smartphones are becoming increasingly popular. With the TouchDevelop programming environment, Microsoft Research is exploring how to further benefit from the computing power of mobile devices. With an easy to use, touch-optimized interface, TouchDevelop is especially targeting lay programmers and students. The TouchBoost project aims to help TouchDevelop programmers write better code by employing various static analyses that identify and report possible programming errors.

Many real programs do not work stand-alone; they reuse code and access data. The diversity of today's computing devices fosters the development of software systems that run in their own environment and are remotely accessed over a network. Such a web service and a program typically communicate using HTTP.

We propose a static analysis for programs that send a request to a web service using an URL. The URL identifies the web service and conveys possible input parameters. In a program, an URL is just a string. The specifications of the called web service impose constraints on the value of the URL string. Hence, knowledge about the possible values of URL strings is crucial for our analysis.

We formalize a predicate logic on strings. A so-called “Split expression” provides a predicate in higher-order logic that divides a string into two substrings and applies a predicate to both substrings. We define the abstract semantics of the Split expression and implement an algorithm that divides the employed string domain at a given delimiter. The split operation is able to extract the string representation of parameters passed to the web service. The parameters however, need not necessarily be strings. We develop an extension to an existing string analysis with the relational information of objects and their string representation.

The web service specifications give guarantees about the returned strings. Often, the returned string is structured hierarchically. We design structural annotations for string values to collect information about the represented format (e.g., JSON, XML) and fields that are guaranteed to exist if the string is converted to the appropriate object.

Contents

Abstract	3
1 Introduction	7
2 Background	11
2.1 TouchDevelop	11
2.2 Static Program Analysis	12
2.3 Abstract Interpretation	13
2.4 TouchBoost	14
2.5 Web Services	14
2.6 Bricks	15
2.7 Maybe Contained Character Analysis	17
3 Analysis	19
3.1 Expressions	20
3.2 Relational Bricks	22
3.2.1 Symbolic Variables	24
3.2.2 Abstract String	25
3.2.3 Abstract Strings with Maybe Contained Character Domain	27
3.3 Split Expressions	29
3.3.1 Split Expression as a Predicate	31
3.3.2 Forall-Splits Quantifier	31
3.4 Split Expression in the Abstract Domain	32
3.4.1 Introducing the Abstract Evaluation of Split Expressions	33
3.4.2 Split Operation on Relational Bricks	35
3.4.3 Termination of Split Expressions	41
3.5 Web Service Specification	42
3.5.1 Intermediate Representation of Web Service Specification	43
3.5.2 Derived Pre- and Postconditions	46
3.6 Structural Annotations for String Values	49

4	Evaluation	53
4.1	Results	55
4.2	Discussion	58
5	Related Work	63
6	Conclusion	65
6.1	Future Work	66

Chapter 1

Introduction

TouchDevelop is a programming environment developed by Microsoft Research. TouchDevelop allows users to create applications for their mobile devices with a user interface that is optimized for development directly on the device using a touch screen; code can be built using predefined blocks which greatly reduces the need for keyboard inputs. TouchDevelop comes with an API that provides a lot of functionality, such as reading the devices sensor data, establishing Internet connections, and printing graphical and textual output. Furthermore, the only requirement to run TouchDevelop is a modern web browser, which means that TouchDevelop scripts can be executed on many different devices. All those factors make TouchDevelop attractive for programmers without much programming experience.

The barrier to publishing TouchDevelop scripts is very low, leading to potentially many publications by inexperienced programmers. Program analysis can help programmers writing correct code by pointing out possible mistakes. Currently, the Chair of Programming Methodology at the ETH Zürich is working on TouchBoost, a toolbox providing various static analyses that will help TouchDevelop programmers write correct scripts by finding and warning about possible erroneous behavior. Once deployed, TouchBoost should be able to improve the overall quality of published TouchDevelop scripts.

A web service is a software system that interacts with a client program remotely over a network. Since they are operating in their own environment, web services are not limited to the same resources as the calling application is. Often, web services are used to request specific information from a large set of data. Analyzing a program that communicates with a web service comes with different challenges than analyzing a program that uses a traditional library that is executed in the same environment as the program.

Recent sales statistics highlight the importance of mobile devices. Statistics published by Gartner [17], [18] show a worldwide shipment of 315 967 516 PCs and a worldwide sale of 195 435 004 tablets in 2013. The statistics also

```

1  var location := senses → current location
2  var base := "http://api.nytimes.com/svc/events/v2/listings.json
   ?"
3  var position := "ll=" // location → latitude // "," // location →
   longitude
4  var key := "&api-key=" // data → key
5  var url := base // position // key
6  var response := web → download json(url)
7  if (not response → is invalid)
8    var numEvents := response → number("num_results")
9    var events := response → json("result")
10   var i := 0
11   while (i < numEvents)
12     events → at(i) → string("event_name") → post to wall
13     i := i + 1

```

Figure 1.1: Motivating example.

show a significant rise in tablet sales (68%) and a decline in PC shipments (-10%) from 2012 to 2013. Even without adding Smartphones to the equation, those numbers emphasize the significant contribution of mobile devices to the number of today's computing devices. As such, the development of platform independent web services, and the research of new, mobile-friendly programming environments such as TouchDevelop is promising.

This work contributes to the TouchBoost toolbox by providing an analysis that reasons about the communication between TouchDevelop scripts and web services.

There are some important differences between web services and regular libraries that an analysis must be aware of. Firstly, the functionality implemented by the web service is not directly linked in memory. Instead, communication is done over a network connection and follows corresponding protocols such as HTTP. At any time, there is no guarantee that the web service is available, which means communication may potentially fail. Secondly, the sent messages are not objects that are checked by the type system, but rather string values conveying the semantic information. Hence, our proposed analysis is heavily dependent on statically knowing the values a string in the program can take.

Figure 1.1 shows code written in TouchDevelop with a call to a method provided by a web service. When executing, the code is supposed to retrieve and display a list of events that are held nearby. The event listing is provided by the web API of The New York Times [1]. The web service expects the latitude and longitude coordinates being passed to the web service as part of the URL string. The example program retrieves the coordinates by asking the device for its current location. The coordinates are passed to the web service as a key-value pair in the query part of the URL, i.e., after the

question mark.

The actual communication with the web service is done when the `download json` method is called in line 5. This method parses the textual response of the web service as a JSON object. The communication with a web service may fail, returning an invalid response, because the web service could be unavailable. Statically, there is no way to guarantee the validity of the response.

Inside the if-statement, in lines 7 to 12, the JSON-object `response` is valid. Nonetheless, the validity property by itself cannot guarantee that the fields `num_results` or `result` exist in the JSON-object. Conservatively, a static analysis can give warnings that the keys might not exist, but the specification of the web service could provide more precise information.

The specification of the web service provides contracts on how to communicate with the web service. In the example, an analysis can find out that the event-listings method of The New York Times web API is accessed. Suppose the contracts of this method require the caller to provide a location in New York as a tuple of numeric values that represent the latitude and the longitude. Suppose that, if those preconditions are met by the caller, the service guarantees that a string is returned that represents a JSON-object containing the integer field `num_results` and the field `result`. Given those contracts of the web service and the program code in the example it is possible to deduce that `numEvents` takes a valid integer value and that the value of `events` is a valid JSON object.

Our work makes the following contributions:

- We propose an augmentation of an existing string analysis to keep the relation between the value of (sub)strings and the value of the arbitrary typed object they represent.
- We define a format and an intermediate representation for specification of stateless web services.
- In the abstract interpretation framework, we propose a domain of structural annotations for string values.
- We design and implement a static analysis for the TouchBoost toolbox that checks TouchDevelop programs against web service specifications.
- We are the first to formally capture the semantics of web service specifications in regular programming languages.

Chapter 2

Background

The purpose of this chapter is to prepare the reader for the analysis that is described in Chapter 3. The analysis contributes to the TouchBoost project, which is introduced in Section 2.4. TouchBoost provides various static analyses (Section 2.2) that are designed in the abstract interpretation framework (Section 2.3) for programs written in TouchDevelop (Section 2.1). The goal of the proposed analysis is to receive reasonable precise information about values involved in communication with web services. Web services are introduced in Section 2.5. A program communicates with the web service over HTTP. Specifically, our analysis investigates the constraints the web service specifications put on URL strings and the response. The approach depends on statically knowing the values of strings. In Section 3.2, we propose an augmentation to the Bricks string analysis which is introduced in Section 2.6. The augmentation utilizes the Maybe Contained Character analysis introduced in Section 2.7 to improve the precision of the analysis.

2.1 TouchDevelop

TouchDevelop [22] is an imperative object-oriented programming language developed by Microsoft research. The language is strongly and statically typed, while type inference is applied on local variables. TouchDevelop comes with a browser-based development environment [2] also provided by Microsoft. The development environment allows users to create programs, publish them, and execute them. Being run in the browser, TouchDevelop can be used on almost any device. Its interface is especially designed for development on touchscreens, with ease of use in mind. As such, TouchDevelop is not targeting professional programmers, but tries to appeal especially to hobby programmers and students.

TouchDevelop comes with a rich API that provides high-level methods for various tasks. Amongst other things, the TouchDevelop API provides drawing functions, functions to access to sensor data, functions to access to

media stored on the device, and functions to receive user input. In this work we are interested in the functions of the TouchDevelop API that communicate with software systems over a network.

Each published TouchDevelop script is publicly available via Microsoft's cloud service. TouchDevelop programmers also have the possibility to export their scripts and upload them in the Windows-, Android-, or iPhone/iPad app store. We evaluated the analysis that we describe in Chapter 3 against various real scripts.

2.2 Static Program Analysis

As an imperative programming language, TouchDevelop programs consist of a sequence of statements that each potentially alter the state. In return, the behavior of each statement is dependent on the state. For example, the statement $1 / x$ depends on the value of x at the time of the statements execution. Consequently, some states lead to erroneous execution of statements; if such a state is reached, the program has a bug. For example, if the state of a program sets the value of x to 0 just before the statement $1 / x$ is executed, the program will crash. The ultimate goal of program analysis is to point out bugs in the program code.

Opposed to testing or dynamic program analysis, which aims to find bugs in the program by executing it with concrete input values, static analysis computes the execution of the program without actual input values. Hence, static analysis can provide information about the behavior of a program for all possible executions of the program instead of only a limited number of executions that can be tested dynamically. However, in most programs, a state can take a very large set of values and static analysis usually needs to over approximate the state. Hence, a static analysis often has to sacrifice precision in favor of soundness.

An analysis is sound if it gives a warning about every bug in the program and precise if every warning reveals a genuine bug. Both properties are important for the usefulness of the analysis but are very hard to meet simultaneously. Note that an analysis that reports a lot of false errors, while revealing all genuine bugs, may not be useful to the user because the genuine bugs are hidden in the flood of false bug reports.

In this work we contribute to the TouchBoost project which depends on the static analyzer Sample. The intermediate representation of programs – in our case of TouchDevelop programs – is a control flow graph. Each node in a control flow graph contains either an assignment statement or a condition. Nodes containing a condition have two successors that depend on the execution of the condition. All other nodes, except for the last node in program execution, have one successor. Each node has an entry and an exit state. The control flow graph is not a tree; some statements in

a program may get looped over and executed multiple times. The static analyses compute a fixpoint of the states.

2.3 Abstract Interpretation

Abstract interpretation was developed by Cousot and Cousot [11]. It allows mathematical reasoning about programs without executing them, and provides a static analysis framework. Analyses that fit into the abstract interpretation framework are modifiable, because the abstract domains that represent values in the state can be added, removed or exchanged depending on the needs. The ASTRÉE analyzer, developed by Cousot et al. [12], is a famous static analyzer based on abstract interpretation for the C programming language.

For each statement, abstract interpretation makes an over approximation of all possible input and output states. The set of all possible abstract states form a lattice. In a lattice, every two elements have a unique smaller element and a unique larger element. Hence, in abstract interpretation two important operations on states S_1 and S_2 are defined: The least upper bound operator \sqcup and the greatest lower bound operator \sqcap .

Programs in the static analyzer Sample are represented as a control flow graph. Each node in the control flow graph has an entry state, that is, the state of the program before the statement in the node is executed, and an exit state. At each node, the abstract interpretation technique computes the transition from the entry state to the exit state by applying the abstract semantics imposed by the statement. If a node has multiple predecessors, e.g., the first statement after an if-statement, the input state is computed as the least upper bound of the exit states of all predecessors. This approach ensures that each abstract entry state is an over approximation to the concrete entry state.

Statements might be executed more than once in a single program execution. Computing the abstract state transition only once for each statement does not guarantee us to cover at least all possible concrete states. Abstract interpretation computes a fixpoint using the least upper bound operator on the state after one iteration and the next iteration. However, such a fixpoint computation can take a long time or might in some cases not terminate at all. A widening operator ∇ speeds up the fixpoint computation. After a certain number of iterations, the widening operator over approximates the state after the last iteration. To speed up the fixpoint computation, the widening operator generally sacrifices precision.

2.4 TouchBoost

The analysis described in Section 3 is part of the TouchBoost project which provides various tools to analyze programs written in TouchDevelop. TouchBoost utilizes an analyzer called Sample (Static Analyzer for Multiple Languages) which is defined in the Abstract Interpretation framework [11]. Sample is being developed by the chair of programming methodology at the ETH Zürich.

Because TouchDevelop is especially targeting programmers without much programming experience, and because TouchDevelop scripts are so easily published, many of the published programs contain bugs. TouchBoost aims to help programmers write correct code by providing various static analysis tools that warn programmers about possible mistakes.

Currently, the TouchBoost tool suite contains various analyses for string values, for numerical values, and for the special `invalid` value which is similar to Java's `null`-value, albeit not exactly the same. Other Master's theses completed at the chair of programming methodology also contribute to TouchBoost. Raphael Fuchs [16] developed a backwards analysis in the abstract interpretation framework to decide whether a warning by the forward analyses is genuine. Daniel Schweizer [21] developed an analysis that overapproximates the cost of loops in a TouchDevelop program.

TouchBoost has been deployed as a plugin to the TouchDevelop programming environment.

2.5 Web Services

The term *web service architecture* is defined by the World Wide Web Consortium [6]. They define a web service in the following way:

Definition: A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

The definition is a bit strict as it states that a web service must provide its specification in a machine-processable format, which is often not provided. Especially more recent stateless RESTful web services – a term and architectural style introduced by Fielding and Taylor [14] – provide their specification in a human-readable but not machine-processable format. A key characteristic of the REST architecture is, that the client has to manage state – the response of a RESTful web service should not depend on variables that are not provided by the client. The two main benefits of stateless

web services over stateful web services are improved visibility, because single requests can be observed separately, and scalability, because the web service does not have to allocate resources on saving the state.

Web services have various benefits over traditional libraries:

- They are executed in their own environment and do not use the same resources as the client. This makes them especially useful for mobile devices (which are becoming the most commonly used computing device [22]), because their resources are often limited in favor of mobility and less power consumption.
- Being decoupled from the client code and running in their own environment, web services can actually be accessed independent on the platform and the language the client code is written in. Web services do not require the users to setup their system in a specific way. Hence, web services generally gain a greater audience than their traditional counterparts.
- They completely hide the implementation of their functionality. The executable binaries of a web service need not be available. If the binaries are not available, the program cannot be reverse engineered by disassembling. Hence, pirating a well designed web service is impossible.

Parameters of web service methods are often passed as a set of key-value pairs encoded in the query part of the URL. The response is often text in a machine readable format such as XML or JSON. Contracts on the values of the response and the request are defined by the web service specification. The contracts can restrict valid input parameters, specify the value of the response, and – for stateful web services – restrict the order in which various methods of the same service are to be executed.

Unfortunately, the client of a web service, i.e., a program that sends a request to a web service, does not statically have the web service specification. In a statically typed language such as TouchDevelop the contracts mostly state that the argument and the return value of of a method that communicates with a web service are of type string. Additionally, given the nature of the Internet, every call to a web service must acknowledge the possibility of a connection failure.

2.6 Bricks

The Bricks analysis is a string analysis based on abstract interpretation developed by Costatini et al. [10]. We already briefly discussed the Bricks analysis in Section 5 as it is the main string analysis we base our approach

on. In this section we will describe the Bricks analysis in more detail. Understanding the Bricks analysis is important for understanding the details of our analysis.

The bricks analysis approximates the values a string can take similar to regular expressions. A hands-on example of the functionality of the Bricks analysis is given in Figure 2.1.

In the abstract state, the Bricks analysis maintains a Bricks domain \mathcal{L} for each string value. The Bricks domain contains a list of Brick elements \mathcal{B} . Each Brick element is a set of string values and two integer numbers. The Brick element $\mathcal{B}[S]^{m,M}$ represents all concrete strings that can be built by concatenating between m and M elements from the set of strings S . For example, the Brick element $\mathcal{B}[\{hoch, haus\}]^{1,2}$ represents the string values *hoch*, *haus*, *hochhaus* and *haushoch*.

The Brick element that represents the set of all possible string values is denoted $\top_{\mathcal{B}}$. Furthermore, the Brick element representing an invalid string value, i.e., a value that can never be achieved by a program, is denoted by $\perp_{\mathcal{B}}$.

Multiple Bricks domains can represent the same set of strings. Costatini et al. define five normalization rules that map semantically equivalent Bricks domains to the same Bricks domain. The following equations determine the normalization rules. U , and V are sets of strings, m , M , a , b , x , and y are positive integers, and U^x is a set of strings containing the x -fold concatenation of all strings in U .

$$\mathcal{L} \left[\mathcal{B}[\emptyset]^{0,0} \mathcal{B}[\emptyset]^{0,0} \right] \mapsto \mathcal{L} \left[\mathcal{B}[\emptyset]^{0,0} \right] \quad (2.1)$$

$$\mathcal{L} \left[\mathcal{B}[U]^{1,1} \mathcal{B}[V]^{1,1} \right] \mapsto \mathcal{L} \left[\mathcal{B}[U \cup V]^{1,1} \right] \quad (2.2)$$

$$\mathcal{L} \left[\mathcal{B}[U]^{M,M} \right] \mapsto \mathcal{L} \left[\mathcal{B}[U^M]^{M,M} \right] \quad (2.3)$$

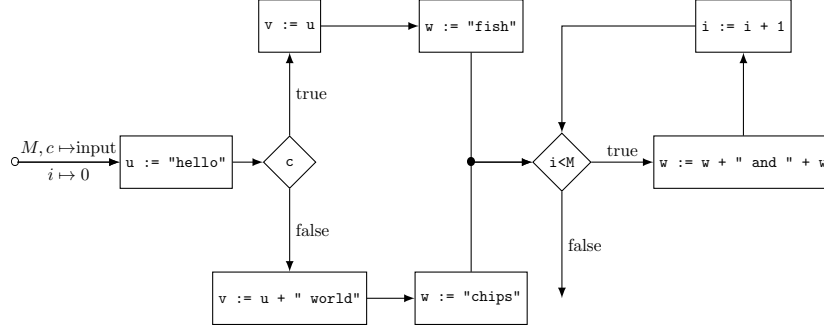
$$\mathcal{L} \left[\mathcal{B}[U]^{x,a} \mathcal{B}[U]^{y,b} \right] \mapsto \mathcal{L} \left[\mathcal{B}[U]^{x+y,a+b} \right] \quad (2.4)$$

$$\forall m > 1, M > m : \mathcal{L} \left[\mathcal{B}[U]^{m,M} \right] \mapsto \mathcal{L} \left[\mathcal{B}[U]^{m,m} \mathcal{B}[U]^{0,M-m} \right] \quad (2.5)$$

In TouchDevelop and many other object-oriented programming languages such as Java, a string representation of arbitrary-typed objects is defined. For example, a program can concatenate a string and a number. A shortcoming of the Bricks analysis is, that it cannot relate substrings created by non-string objects to an appropriate analysis for their actual type. For example, if a program computes the string "value=" + x , where x is not of type string and the plus operator means string concatenation, the Bricks analysis loses all information we may have about x . In Section 3.2 we develop an augmentation to the Bricks analysis to deal with these shortcomings.

2.7 Maybe Contained Character Analysis

Another string analysis defined in the abstract interpretation framework and proposed by Costatini et al. [10] is the Maybe Contained Character analysis. The analysis itself is strictly less expressive than the Bricks analysis, but is cheaper to compute. The analysis does just that what its name promises: It represents a string value as a set of characters that may be contained in the string. To conform the abstract interpretation framework by composing a complete lattice, the Maybe Contained Characters analysis has a top element – representing an infinitely large set of characters – and a bottom element. The least upper bound operator is defined as the union operator on the sets while the greatest lower bound operator is defined as the intersection of the sets.



(a) Control flow graph representation of a program that performs various operations on string values. $:=$ is the assignment operator, $+$ is the string concatenation operator. Assignment nodes are marked by boxes while conditional nodes are marked by diamonds. The values for M and c are already given as input; i is set to zero.

$$\begin{aligned}
 u &\mapsto \mathcal{L} \left[\mathcal{B} [\{\underline{hello}\}]^{1,1} \right] \\
 v &\mapsto \mathcal{L} \left[\mathcal{B} [\{\underline{hello}\}]^{1,1} \mathcal{B} [\{\underline{world}\}]^{0,1} \right] \\
 w &\mapsto \mathcal{L} \left[\mathcal{B} [\{\underline{fish}\}]^{1,1} \mathcal{B} [\{\underline{and fish}, \underline{and chips}\}]^{0,\text{inf}} \right]
 \end{aligned}$$

(b) The values the three string variables can take at the end of the program, represented as an over approximation using the Bricks domain.

$$\begin{aligned}
 u &\mapsto \{\underline{hello}\} \\
 v &\mapsto \{\underline{hello}, \underline{hello world}\} \\
 w &\mapsto \{\underline{fish}, \underline{fish and fish}, \dots, \overbrace{\underline{fish and \dots and fish}}^{M \text{ times}}, \\
 &\quad \underline{chips}, \underline{chips and chips}, \dots, \underbrace{\underline{chips and \dots and chips}}_{M \text{ times}}\}
 \end{aligned}$$

(c) The actual possible values the three string variables can take at the end of the program. Each variable will take exactly one value from the corresponding set of strings.

Figure 2.1: Example program illustrating the functionality of the Bricks analysis.

Chapter 3

Analysis

In this chapter we are going to develop an analysis for programs that utilize web services. Consider the example given in Figure 1.1. The example makes a request to the event listings service provided by The New York Times through their web API. A conservative analysis assumes the possible inaccessibility of the fields `result`, and `num_results` of the JSON-object returned by the web service. The analysis we are going to describe in this chapter is able to deduce the existence of those fields. However, the existence of those fields depends on the string that is returned for the HTTP request sent to the web API. The information on the returned value is described in the specification of the web service and may further depend on the parameters passed through the URL string.

Formally, we have to understand the semantics of web service specifications in the context of our program. In particular, we have to formalize the constraints that a web service specification imposes on the URL strings in the program. To solve this problem, we introduce Split Expressions as predicates in Section 3.3. The Split Expressions can express web service specification constraints on string values. Split Expressions can be used to divide a string into its components and express properties not only of the string but also of the underlying data encoded into the string (e.g. numeric values). In section Section 3.5, we define a generic web service specification language similar to WADL and use the previously introduced split expressions to define their semantics.

Since our goal is to check web service specification not against one concrete string but all possible strings produced by a program, we have to find a suitable string abstraction to precisely and efficiently infer and represent a potentially unbounded number of strings. In particular, we design an abstract domain that satisfies the following requirements:

- The domain can efficiently and precisely evaluate Split Expressions on strings.

- The domain is able to represent the semantics of operations that are typically used to create URL strings, e.g., conditional statements, string concatenation, substring replacement, URL encoding.
- The domain must be able to extract non-string values encoded in an URL string.

At the core of our analysis is a domain that represents URL strings. In Section 3.2, we propose a modification of the Bricks string analysis that was developed by Constatini et al. [10] to represent the values of the URL strings. An introduction to the original Bricks string analysis is given in Section 2.6. The Relational Bricks domain

- stores the string-representation of non-string values as a symbolic variable. The variable is used to store and access the original, non-converted value in a specialized domain. For example, the expression $(5 * x) \rightarrow \text{to string}$ stores the value of $5 * x$ as a symbolic variable.
- provides a split operation that divides a string into two parts at a given delimiter.
- can provide knowledge about what characters may be contained in a substring, even if there is no knowledge about the order in which these characters appear. This is especially useful in combination with URL encoding: The encoding guarantees that certain reserved characters, e.g., the ampersand, the question mark, and the equal sign, do not appear. The absence guarantee of certain characters in a substring can have a great impact on the precision of the split operation.

Web service specifications not only impose requirements on input strings, they also provide structural guarantees for the returned strings. Often, web services provide their response in a hierarchical data format such as JSON or XML. The format of the response and the fields that can possibly be accessed can depend on the input parameters. Our analysis must be able to deduce the structural guarantees of the response to a request.

3.1 Expressions

In this section we are going to introduce some general notation and the concept of an expression.

Expressions are the elementary building blocks of a program. Let \mathbb{E} denote the set of all expressions. The concept of an expression is refined by a type system. Each expression has a type that restricts the values the expression can take at runtime. Let $\mathbb{S} \subseteq \mathbb{E}$ denote the set of all expressions that take a string value. For short, we will call this set \mathbb{S} the set of all

string expressions and each element of \mathbb{S} will be called a string expression. Similarly, let $\mathbb{B} \subseteq \mathbb{E}$, and $\mathbb{N} \subseteq \mathbb{E}$ be the sets of all boolean, and numeric expressions, respectively.

The order in which the concrete expressions are executed is determined by the program code. Note that in this work we are not concerned with nondeterminism caused by concurrency.

We further distinguish pure and impure expressions. There are various definitions of purity in the literature with the most rigorous definition of a pure expression being an expression that does not alter the program state in any way and does return a deterministic value. Others have a more practical view and allow pure methods to alter the state in a very restricted way to account for caching and lazy evaluation. In this work, we stick to the rigorous definition which allows us to make a connection to mathematical functions. Inspired by Darvas and Leino [13], we try to give pure expressions a mathematical encoding if necessary.

Let us first define the atomic expressions user input and constants. For simplicity, let a numeric constant be any value in \mathbb{R} . There are exactly two boolean constants: `true` and `false`. While TouchDevelop has no special type for a single character (every single character is a string by itself), we define a string constant as a sequence of UTF-8 characters. The user input expression `ask string` can take any valid string value and the input expression `ask number` can take any value in \mathbb{R} .

In general, let the n -tuple (c_1, c_2, \dots, c_n) be the mathematical representation of a concrete string formed by the UTF-8 characters c_1, \dots, c_n . The special character \emptyset will be used to denote the empty string, which is represented by the 0-Tuple $()$. Because the tuple notation can be tedious, we introduce a notation for string constants: Let

$$\boxed{\text{blueberry}} := (b, l, u, e, b, e, r, r, y) \quad (3.1)$$

.

The string concatenation expression – denoted by the `//` operator in TouchDevelop – is a pure string expression, which can be encoded as the mathematical function

$$\text{concat}((a_0, \dots, a_n), (b_0, \dots, b_m)) = (a_1, \dots, a_n, b_1, \dots, b_m) \quad (3.2)$$

Further, we encode the pure string expressions `count`, `substring`, and `indexof`

as mathematical functions:

$$\text{count}((s_0, \dots, s_n)) = n + 1 \quad (3.3)$$

$$\text{substring}((t_0, \dots, t_n), s, l) = \begin{cases} () & \text{if } l < 1 \vee s < 0 \vee s > n \\ (t_s, \dots, t_n) & \text{if } s + l \geq n \\ (t_s, \dots, t_{s+l-1}) & \text{otherwise} \end{cases} \quad (3.4)$$

$$\text{indexof}((t_0, \dots, t_n), (a_0, \dots, a_m)) = \min_i \{-1, \arg_i\{(t_i, \dots, t_{i+m}) = (a_0, \dots, a_m)\}\} \quad (3.5)$$

The boolean and arithmetic operators are also pure expressions. We will use the obvious mathematical counterpart for the encoding and denote the encoding function by a conventional mathematical symbol with an overline. For example, the expression $x + y$ is encoded as $\bar{+}(x, y) = x + y$, and the expression **a and b** is encoded as $\bar{\wedge}(a, b) = a \wedge b$. Comparison operators may depend on the compared expressions. In case of string expressions, boolean expressions, and numeric expressions, comparison operators are also pure expressions and we use the obvious mathematical counterpart for encoding.

Formally, the representation of boolean expressions as a mathematical function is a predicate.

3.2 Relational Bricks

The Bricks analysis, as mentioned in Section 2.6 and developed by Costantini et al. [10], provides an abstract domain to represent string values. The domain captures information about strings that is similar to regular expressions. The Relational Bricks analysis is strictly more expressive than the Bricks analysis. The Relational Bricks analysis is supposed substitute the Bricks analysis where the additional expressiveness is needed. We are going to use the same notation as the concepts stay the same.

Figure 3.1 gives high-level view on the Relational Bricks analysis. The analysis is built hierarchical out of the Bricks domain, the Brick element, and the Abstract String domain. The Abstract String domain is introduced in Section 3.2.2. The domain is further refined in Section 3.2.3. The difference between the original Bricks analysis and the Relational Bricks analysis are the possible contents of the Brick elements: The original analysis uses string constants while the augmented Relational Bricks analysis stores a set of Abstract Strings.

A set of concrete string values is represented by a Relational Bricks domain \mathcal{L} , consisting of a list of Brick elements $\mathcal{B}[S]^{m,M}$. S is a set of Abstract String domains, which are also an abstract domain that capture information about concrete string values. The Abstract String domain is

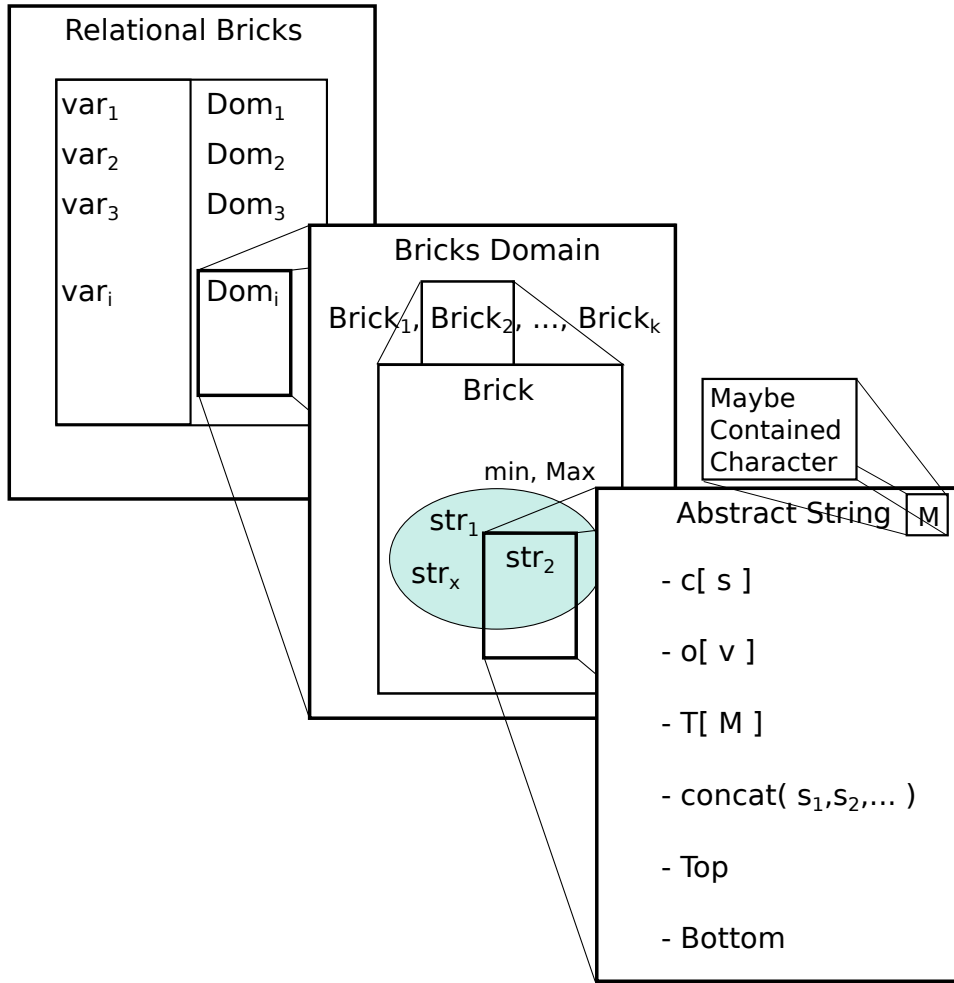


Figure 3.1: The big picture of the Relational Bricks analysis.

introduced in Section 3.2.2 and in Section 3.2.3. The Abstract String mainly provides a vessel for symbolic variables – introduced in Section 3.2.1 – which make the connection between values of an arbitrary type and their string representation.

The original Bricks analysis does not allow to build relations to other abstract domains. Suppose a program communicates with a web API that requires a parameter *lon* which must take a value between -180 and 180 . Suppose the corresponding substring of the URL string is built as $p := \text{"lat"} // x$, where the numeric variable x fulfills the requirement. The Bricks analysis could capture this string value of p as

$$p \mapsto \mathcal{L} \left[\mathcal{B} [\{\underline{\text{lat}}\}]^{1,1} \mathcal{B} [\{\square\}]^{0,1} \mathcal{B} [\{\underline{\text{I}}\}]^{0,1} \mathcal{B} [\{\underline{\square}, \dots, \underline{\square}\}]^{0,1} \mathcal{B} [\{\underline{\square}, \dots, \underline{\square}\}]^{1,1} \right]$$

However, this representation can only be calculated if the semantics of con-

```

1 var x = wall → ask number("factorial_of?")
2 var base = "http://api.calculator.com/factorial?"
3 var param = "val=" // x
4 if (x < 0) then
5   x = 0
6 var js = web → download json(base // param)
7 if (not js → is invalid) then
8   js → string("result") → post to wall

```

Figure 3.2: A TouchDevelop script using a fictitious web API at `api.calculator.com` that computes the factorial of `val`. Factorials can only be computed for positive integers. Negative numbers can potentially be passed to the web service, albeit the program is filtering negative values of `x`. There is a bug in the program: The invalid values of `x` are filtered after they are assigned to the substring in line 3. The value of the symbolic variable introduced by our analysis is equal to the value of `x` at line 3 and not necessarily equal to the value of `x` at line 6.

version to a string value are defined accordingly. One cannot define precise general semantics for conversion to string values of arbitrary-typed values. Each time a new domain is introduced, those semantics have to be defined anew. Furthermore, it is not straightforward to relate this string representation with other values of the original type; the semantics for the reverse conversion – or parsing of the string – need to be defined. In our example, we might want to assume that the string representation of `x` indeed corresponds to a value that is less than 180.

The relation between the original value and its string representation is kept in the Abstract String using a symbolic variable identifier. This allows to store the an abstract value for the string representation without any loss in precision and without the need to define additional semantics.

3.2.1 Symbolic Variables

For every expression that is transformed by either explicitly or implicitly calling to `string`, the analysis keeps the relation between the string value and the value of the original expression by introducing a symbolic variable. The purpose of the symbolic variable is to store the current value of the expression. In general, the value of an expression depends on the state it is executed in and we must assume that an expression evaluates differently in a different state. However, a variable that gets only assigned once is guaranteed to evaluate to the same value independent of other changes to the state.

Consider the example from Figure 3.2: The value that gets assigned to variable `param` is the concatenation of the string constant `val` and the

string representation of x . Hence, the analysis creates a new symbolic variable sv_1 when looking at the string concatenation in line 3. The analysis further assumes equality of sv_1 and x . Because of this, the changes to x in the following if-statement do not affect the value of sv_1 , meaning the introduction of a new symbolic variable is necessary for every non-constant expression. Assuming that the web service does only compute factorials for positive numbers, the program has a bug, because the value of x is kept positive only after its original value is used as the val-parameter in the URL query. Had the programmer put line 3 after the if-statement, the symbolic variable would have been restricted to a positive number.

3.2.2 Abstract String

The symbolic variables introduced in the previous section do not make a relational domain by themselves. They merely store information about the value of an expression that gets interpreted as a string value. The Bricks analysis should be able to somehow keep track about the relation between substrings and the values of the symbolic variables. Instead of memorizing a set of string constants in each Brick element, a set of Abstract Strings is memorized. Abstract Strings are elements of the Abstract String domain that is defined by the grammar

$$G = (\{S, C, O\}, \Sigma_T, R, S) \quad (3.6)$$

with the terminals

$$\Sigma_T := \left\{ \begin{array}{ll} \emptyset_{S_A}, & \text{the empty string } () \\ c[x], & \text{representation of string constant } x \neq () \\ o[sv] & \text{symbolic variable } sv \\ \top_{S_A} & \text{every valid string} \\ \perp_{S_A} & \text{an invalid string } \} \end{array} \right.$$

with the rules

$$R := \left\{ \begin{array}{l} S \mapsto \emptyset_{S_A} \mid C \mid \top_{S_A} \mid \perp_{S_A} \\ C \mapsto c[x] \mid c[x]O \mid O \\ O \mapsto o[sv]C \mid o[sv] \} \end{array} \right.$$

Hence, the grammar represents all possible sequences of Abstract String constants $c[\dots]$ and other Abstract Strings $o[\dots]$ without multiple consecutive constants. Giving the sequence of Abstract Strings the meaning of string

concatenation, we are able to uniquely represent all string values. The value of the variable `url` in the example given in Figure 3.2 is uniquely represented by the Abstract String

$$c \left[\overline{\text{http://api.calculator.com/factorial?val=}} \right] o [sv_1]$$

where sv_1 is assumed to be equal to `num` and sv_2 is assumed to be equal to `den`.

The Abstract String domain – denoted \mathcal{S}_A from now on – forms a complete lattice with $\top_{\mathcal{S}_A}$ being the maximum, $\perp_{\mathcal{S}_A}$ being the minimum and every other element being at the same height. Formally, \mathcal{S}_A forms the complete lattice $\langle \mathcal{S}_A, \leq_{\mathcal{S}_A}, \perp_{\mathcal{S}_A}, \top_{\mathcal{S}_A}, \sqcup_{\mathcal{S}_A}, \sqcap_{\mathcal{S}_A} \rangle$.

$$x \leq_{\mathcal{S}_A} y := \begin{cases} \text{true} & \text{if } x = y \vee x = \perp_{\mathcal{S}_A} \vee y = \top_{\mathcal{S}_A} \\ \text{false} & \text{otherwise} \end{cases} \quad (3.7)$$

$$x \sqcup_{\mathcal{S}_A} y := \begin{cases} y & \text{if } x = \perp_{\mathcal{S}_A} \vee y = \top_{\mathcal{S}_A} \vee x = y \\ x & \text{if } y = \perp_{\mathcal{S}_A} \vee x = \top_{\mathcal{S}_A} \\ \top_{\mathcal{S}_A} & \text{otherwise} \end{cases} \quad (3.8)$$

$$x \sqcap_{\mathcal{S}_A} y := \begin{cases} y & \text{if } y = \perp_{\mathcal{S}_A} \vee x = \top_{\mathcal{S}_A} \vee x = y \\ x & \text{if } x = \perp_{\mathcal{S}_A} \vee y = \top_{\mathcal{S}_A} \\ \perp_{\mathcal{S}_A} & \text{otherwise} \end{cases} \quad (3.9)$$

We need to define the semantics of string concatenation on Abstract Strings, because the semantics of string concatenation on Brick elements relies on this definition. Note that a sequence of Abstract Strings represents string concatenation by definition. However, not every sequence of Abstract Strings is an Abstract String as of the definition. To concatenate two Abstract Strings, first join the two Abstract Strings. Then, recreate an abstract string by applying the following rules:

- The definition prohibits two subsequent Abstract String constants. Replace all two subsequent Abstract String constants by a single Abstract String constant. The new constant is created by concatenating the concrete string constants.
- Remove all $\emptyset_{\mathcal{S}_A}$. If two empty strings were concatenated in the first place, the result is still $\emptyset_{\mathcal{S}_A}$ though.
- If the Abstract String contains $\top_{\mathcal{S}_A}$ return $\top_{\mathcal{S}_A}$.
- If the Abstract String contains $\perp_{\mathcal{S}_A}$ return $\perp_{\mathcal{S}_A}$.

Concatenation of Abstract Strings can formally be defined as follows: Let $a \oplus b$ denote the concatenation of the two Abstract Strings a and b . For arbitrary Abstract Strings $a_1, \dots, a_n, b_1, \dots, b_m$ and string literals u, v , the following equations hold:

$$\begin{aligned} a = a_1 \dots a_n c [u] \in \mathcal{S}_A \wedge b = c [v] b_1 \dots b_m \in \mathcal{S}_A &\Rightarrow \\ a \oplus b = a_1 \dots a_n c [\text{concat}(u, v)] b_1 \dots b_m &\quad (3.10) \end{aligned}$$

$$\begin{aligned} a = a_1 \dots a_n \in \mathcal{S}_A \wedge b = b_1 \dots b_m \in \mathcal{S}_A \wedge (a_n \neq c [u] \vee b_1 \neq c [v]) &\Rightarrow \\ a \oplus b = a_1 \dots a_n b_1 \dots b_m &\quad (3.11) \end{aligned}$$

$$a_1 \oplus \emptyset_{\mathcal{S}_A} = \emptyset_{\mathcal{S}_A} \oplus a_1 = a_1 \quad (3.12)$$

$$a_1 \oplus \perp_{\mathcal{S}_A} = \perp_{\mathcal{S}_A} \oplus a_1 = \perp_{\mathcal{S}_A} \quad (3.13)$$

$$a_1 \oplus \top_{\mathcal{S}_A} = \top_{\mathcal{S}_A} \oplus a_1 = \top_{\mathcal{S}_A} \quad (3.14)$$

Note that the Brick element containing $\top_{\mathcal{S}_A}$ represents all possible strings. Hence, $\mathcal{B}[\{\top_{\mathcal{S}_A}, \dots\}]^{m, M} = \top_{\mathcal{B}}$. Consequently, every Brick containing $\top_{\mathcal{S}_A}$ can be transformed into $\top_{\mathcal{B}}$. Analogous, the Brick element containing only $\perp_{\mathcal{S}_A}$ is equivalent to $\perp_{\mathcal{B}}$.

$$\forall m, M \in \mathbb{N}, \forall S \subseteq \mathcal{S}_A : m < M \wedge \top_{\mathcal{S}_A} \in S \rightarrow \mathcal{B}[S]^{m, M} \mapsto \top_{\mathcal{B}} \quad (3.15)$$

3.2.3 Abstract Strings with Maybe Contained Character Domain

The domain that is described in the previous section may not be precise enough in many cases. The example in Figure 3.3 utilizes URL encoding on user input. URL encoding replaces reserved characters such as $\text{\$}$ and \% by other characters. Similar to SQL injection, a user could otherwise manipulate the query and retrieve a web resource that was not intended by the programmer.

```

1 var base := "http://transport.opendata.ch/v1/connections?"
2 var f := web → url encode (wall → ask string ("Departure?"))
3 var t := web → url encode (wall → ask string ("Arrival?"))
4 var query := "from=" // f // "&to=" // t
5 var result := web → download json (base // query)

```

Figure 3.3: Example web service request with arbitrary string queries by the user. URL encoding prohibits injection with malicious user input.

To retain precision when analyzing programs where substring replacement is used, we propose an extension to the Abstract String domain. Another domain is added to each Abstract String: The Maybe Contained Char-

acter domain. The added domain is an augmentation to the domain proposed by [10] as described in Section 2.7. However, the original domain only stores a set of characters that may be contained, being infeasible for strings that can possibly contain many characters except a few. The modified domain stores both, a set domain of maybe contained characters and an inverse set domain of definitively not contained characters. Naturally, it is not possible to gain information from a combination of the two subdomains simultaneously, as every character not contained in the set of maybe contained characters is definitively not in the string, and hence an element of the inverse set domain. The opposite is likewise true: Each character that is not contained in the inverse set domain may be contained in the string and therefore is an element of the set domain. Nevertheless, the augmented Maybe Contained Character domain adds more expressiveness and flexibility. In some cases, the set of possibly contained characters is much smaller than the set of definitively not contained characters; sometimes it is the other way around. The string representation of a numeric value can for example only contain digits, a decimal point and a minus sign. On the other hand, the value of variable f in the example in Figure 3.3 can potentially contain every character except for the reserved characters.

From now on, let $\mathcal{M}[\{a_1, \dots, a_n\}]$ denote the Maybe Contained Character domain that may but need not contain the characters a_1 to a_n but no other characters, and let $\mathcal{M}[\overline{\{b_1, \dots, b_m\}}]$ denote the Maybe Contained Character domain that may contain every character except b_1 to b_m . $\mathcal{M}[S]$, where S is an Abstract String, denotes the Maybe Contained Character domain of S , and hence represents all characters that may be contained in S . The Maybe Contained Characters domain forms a complete lattice with the top element $\top_{\mathcal{M}}$ and the bottom element $\perp_{\mathcal{M}}$.

Let us describe in more detail how the Maybe Contained Character domain is used in combination with the Abstract String and what implication this combination has. The elementary building blocks still exist: Constants $c[x]$, other strings $o[sv]$ and the top and bottom element $\top_{\mathcal{S}_A}$ and $\perp_{\mathcal{S}_A}$ are combined to represent a set of possible string values. However, each of these elementary building blocks receives a Maybe Contained Character domain. For $\perp_{\mathcal{S}_A}$, this domain is always bottom, and the string constant $c[(x_1, x_2, \dots, x_n)]$ may contain exactly the characters x_1 to x_n . The Maybe Contained Character domain of $o[sv]$ contains exactly the characters that the string representation of sv can contain; for numeric values this is for example the set of all digits, the decimal point and the minus sign. Positive numbers cannot contain the minus sign and integers cannot contain the decimal point.

The changes are more severe for the top element $\top_{\mathcal{S}_A}$. Let $[\top_{\mathcal{S}_A}[M]]$, with the Maybe Contained Characters domain M , represent all strings that do only contain any of the characters contained in M . Assuming that C is

the set of all URL reserved characters, the Abstract String that represents f in the example in Figure 3.3 is $[\top_{\mathcal{S}_A}[\mathcal{M}[\overline{C}]]]$. To cope with the increased expressiveness, we modify the grammar that defines the Abstract String domain in Equation 3.6.

$$G = (\{S, C, O\}, \Sigma_T, R, S) \quad (3.16)$$

$$\Sigma_T := \{\emptyset_{\mathcal{S}_A}, c[x], o[sv], [\top_{\mathcal{S}_A}[M]], \top_{\mathcal{S}_A}, \perp_{\mathcal{S}_A}\}$$

$$\begin{aligned} R := \{ & S \mapsto \emptyset_{\mathcal{S}_A} \mid C \mid \top_{\mathcal{S}_A} \mid \perp_{\mathcal{S}_A} \\ & C \mapsto c[x] \mid c[x]O \mid O \\ & O \mapsto o[sv]C \mid o[sv] \mid [\top_{\mathcal{S}_A}[M]]C \mid [\top_{\mathcal{S}_A}[M]] \} \end{aligned}$$

The definition of concatenation of Abstract Strings must be updated accordingly. The new elementary building block $[\top_{\mathcal{S}_A}[C]]$ can be part of a sequence of building blocks. However, the sequence $[\top_{\mathcal{S}_A}[C_1]][\top_{\mathcal{S}_A}[C_2]]$ represents the same set of strings that $[\top_{\mathcal{S}_A}[(C_1 \sqcup C_2)]]$ does. The following equation is added to the definition of the string concatenation:

$$\begin{aligned} a = a_1 \dots a_n [\top_{\mathcal{S}_A}[U]] \in \mathcal{S}_A \wedge b = [\top_{\mathcal{S}_A}[V]] b_1 \dots b_m \in \mathcal{S}_A \Rightarrow \\ a \oplus b = a_1 \dots a_n [\top_{\mathcal{S}_A}[(U \sqcup V)]] b_1 \dots b_m \end{aligned} \quad (3.17)$$

Accordingly, the condition on Equation 3.11 must be updated and the equation is replaced by the following equation:

$$\begin{aligned} a = a_1 \dots a_n \in \mathcal{S}_A \wedge b = b_1 \dots b_m \in \mathcal{S}_A \wedge (a_n \neq c[u] \vee b_1 \neq c[v]) \\ \wedge (a_n \neq [\top_{\mathcal{S}_A}[U]] \vee b_1 \neq [\top_{\mathcal{S}_A}[V]]) \Rightarrow \\ a \oplus b = a_1 \dots a_n b_1 \dots b_m \end{aligned} \quad (3.18)$$

3.3 Split Expressions

In this section we define a new expression which we call Split Expression. The Split Expression shall provide a predicate on substrings. Consider the example string $\underline{length=14}$. Suppose this string is substring to a URL query and the corresponding specifications state that the value of *length* must be positive. These constraints are represented by the informal predicate: “If left of the equal sign is the value *length*, then the value right of the equal sign must be positive.”

```

1  action split(t: String, s: String, f: Comparison of Strings)
2  returns(p: Boolean)
3      var idx := t→index of(s)
4      if(idx>0) then
5          var l := t→substring(0,idx)
6          var idx2 := idx + s→count
7          var r := t→substring(idx2,t→count - idx2)
8          p := f→run(l, r) > 0
9      else
10         p := f→run(t, "") > 0
11     end

```

Figure 3.4: A concrete implementation of the Split expression as an action in TouchDevelop. The input parameter f is typed as a comparison of strings, which is an action taking two string arguments and returning one numeric argument.

First, we are going to define the Split expression for concrete string values. Figure 3.4 shows a possible implementation of Split expressions for TouchDevelop. Later, in Section 3.3.1, we will define the Split expression as a mathematical predicate on abstract string values.

The semantics of the Split expression are the following: Given an input string t , a separation string s , and a predicate f , find the first occurrence of s in t and apply f to the substring left and right of this occurrence. For example, the Split expression applied to the input string "hello", the separation string "l" and the predicate f returns $f(\text{"he"}, \text{"lo"})$. If the separation string s does not occur in t , the whole string t is considered to be the left hand side while the right hand side is considered to be the empty string.

Figure 3.4 shows a working implementation of the Split expression in TouchDevelop to illustrate the semantics on a concrete string. However, the analyzed programs do not need to call this Split expression for our analysis to be able to work. The analysis does insert the Split expression itself. Note that the analysis is only allowed to modify the original program if the behavior of the program is not changed at all. In other words, all changes done by the analysis must preserve the state at any point in the program except for variables introduced by the analysis. Pure functions, as mentioned in Section 3.1, do not change the state. Fortunately, if we impose a purity restriction on f , we can show that the Split expression itself is in fact pure:

- The statements in line 3,5,6, and 7 in Figure 3.4 are pure, because `substring`, `count`, and `index of` are pure as defined in Section 3.1. The state is only changed by adding new temporary variables.
- Since we restrict f to be pure, only pure methods are executed in the

statements in line 8 and 10. p that gets assigned is a freshly created variable that gets returned.

- As each individual statement of the Split expression is pure, we can conclude that the Split expression itself is pure. As such, we can define a mathematical function with equivalent semantics, i.e., a predicate.

3.3.1 Split Expression as a Predicate

The purity property of the Split expression allows us to define a mathematical predicate with equivalent semantics. Equation 3.19 defines the Split expression as mathematical function with a boolean value domain. We will be referring to this mathematical function as Split expression.

$$\begin{aligned}
\mathfrak{h}: \mathbb{S} \times \mathbb{S} \times (\mathbb{S} \times \mathbb{S} \rightarrow \mathbb{B}) &\rightarrow \mathbb{B} \\
\mathfrak{h}(t, \sigma, f) &= f(l, r), \text{ where} \\
l &= \begin{cases} \text{substring}(t, 0, \text{indexof}(t, \sigma)) & \text{if } \text{indexof}(t, \sigma) \geq 0 \\ t & \text{otherwise} \end{cases} \\
r &= \begin{cases} \text{substring}(t, \text{indexof}(t, \sigma) + \text{count}(\sigma), \text{count}(t)) & \text{if } \text{indexof}(t, \sigma) \geq 0 \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned} \tag{3.19}$$

The Split expression is a predicate in higher-order logic. The argument f of \mathfrak{h} is a predicate itself.

Let us give a more practical example to show the usefulness of the Split expression. Often, a parameter to a web service is passed as a key-value pair. In the example given in Figure 3.2, the variable `param` takes a value from the abstract string representation $c[\overline{val} \equiv] o[sv]$, given the assumed equality of sv and x . The Split expression $\mathfrak{h}(\text{param}, \equiv, (l, r) \rightarrow l = \overline{val} \wedge r \geq 0)$ allows us to state that we assume `param` to be a key-value pair where the key is equal to \overline{val} and the value is a positive number.

3.3.2 Forall-Splits Quantifier

Using the fact that the Split expression itself is a predicate, we define a forall-splits quantifier \mathcal{A}

$$\begin{aligned}
\mathcal{A}: \mathbb{S} \times (\mathbb{S} \mapsto \mathbb{B}) \times \mathbb{S} \times \mathbb{S} &\mapsto \mathbb{B} \\
\mathcal{A}(\sigma, f, e_1, \emptyset) &:= f(e_1) \\
\mathcal{A}(\sigma, f, e_1, e_2) &:= \bar{\wedge}(f(e_1), \mathfrak{h}(\text{tostring}(e_2), \sigma, \mathcal{A}(\sigma, f)))
\end{aligned} \tag{3.20}$$

By definition, \mathcal{A} can be used as predicate function for a Split expression where it will recursively introduce a Split expression for the right split until the separation character σ does no longer exist.

With the forall-splits quantifier on splits we can create powerful boolean expressions on URL strings. Usually, parameters to web services are passed as a set of key value pairs. The parameter list is separated from the base URL by a question mark, and the individual parameters are separated by an ampersand. The key and the value of each parameter are then separated by an equal sign. Assume that an example program requests an answer from a web service that requires two input parameters with the key `lat` and `long` that take a valid latitude and longitude value. The order in which the parameters are passed is not important. Suppose the URL is stored in the variable `url`. A combination of Split expressions creates a boolean expression that states which values for this URL are valid:

$$\begin{aligned}
& \mathfrak{h}(\text{url}, \mathbb{Z}, (b, P) \rightarrow \\
& \quad \mathfrak{h}(P, \mathbb{E}, \mathcal{A}(\mathbb{E}, p \rightarrow \\
& \quad \quad \mathfrak{h}(p, \mathbb{E}, (k, v) \rightarrow \\
& \quad \quad \quad (k \neq \overline{\text{lat}} \vee (v \leq 90 \wedge v \geq -90)) \wedge \\
& \quad \quad \quad (k \neq \overline{\text{long}} \vee (v \leq 180 \wedge v \geq -180))))))
\end{aligned} \tag{3.21}$$

The Split expression from Equation 3.21 states that *on the right side* of the first occurrence of \mathbb{Z} , *each split* separated by \mathbb{E} , when split at \mathbb{E} complies with the restriction that *either the left side* of the split is not equal to the key (`lat`, or `long` respectively), *or the right side* of the split is within the respective bounds. In short, the above Split expression simply states that for each parameter, if the key is `lat`, then the value must be within $[-90, 90]$ and if the key is `long`, then the value must be within $[-180, 180]$.

Each concrete string value is finite. Hence, each string can only be split finitely many times and the execution of a Split expression on a concrete string is guaranteed to terminate even if a forall-splits quantifier is applied. However, the analysis is performing the Split expressions on an abstract representation of string values which can represent potentially infinitely many concrete string values. Termination of the evaluation of a Split expression depends on the abstract implementation in the corresponding string domains.

3.4 Split Expression in the Abstract Domain

This section describes how the Split expressions defined in Section 3.3 are used in the abstract interpretation framework. Notably, this section gives a formal definition of the execution of Split expressions in the relational Bricks analysis, accompanied by realistic examples.

A Split expression by itself, being a predicate and hence a pure expression, does not alter the state. As such they can however be assumed as program specification. An assumption reduces the over approximation of the real state, leading to a more precise analysis.

Split expressions that divide a string value at a delimiter of arbitrary length are difficult to formalize on the Bricks domain, because they require the observation of the Bricks domain as a whole. Dividing a Bricks domain at a delimiter string of size 1 greatly reduces the complexity. Fortunately, having the semantics of URL strings in mind we do not restrict ourself when reducing the expressiveness of the Split expressions by constraining the split delimiter to a single character. A more elaborate argument on why a formalization on the Bricks domain is much harder for delimiters of arbitrary length is given in Section 3.4.2, but from now on, just assume that the delimiter is a single character.

3.4.1 Introducing the Abstract Evaluation of Split Expressions

We need to define the semantics of the Split expression predicate on the Relational Bricks domain. The semantics should be sound and keep a reasonable degree of precision. Furthermore, computing the Split expression on the Relational Bricks domain shall terminate. Because a Bricks domain can represent infinitely many strings of arbitrary length, and because the forall-splits quantifier introduces predicates recursively, termination is no simply guaranteed. Nevertheless, we can design the semantics without having to sacrifice precision.

We make the following two observations about the Split expression $\pitchfork(t, \sigma, f)$:

1. f is a predicate function. Hence, the evaluation of f is deterministic.
2. If a Split expression is assumed on the string expression t , it must hold for every possible value of t .

The first observation is used when dealing with large or infinite sets of strings in combination with the forall-splits quantifier \mathcal{A} defined in Section 3.3.2 or similar recursive calls to the Split expression. Because of the deterministic evaluation, the domain does not need to evaluate Split expressions on the same strings and predicate function f multiple times. For example,

$$\begin{aligned} \pitchfork(\overline{5\mathcal{E}5\mathcal{E}5\mathcal{E}5}, \overline{\mathcal{E}}, \mathcal{A}(\overline{\mathcal{E}}, x \rightarrow x = 5)) = \\ \pitchfork(\overline{5\mathcal{E}5}, \&, (l, r) \rightarrow l = 5 \wedge r = 5) \end{aligned}$$

This property can be extended to Bricks domains that represent an infinite set of concrete strings. While a Bricks domain can represent infinitely many strings, the Bricks domain itself is finite as it can only contain finitely

many Brick elements, which themselves only contain a finite set of Abstract Strings. In Section 3.4.3, after completely introducing the semantics of the Split expression in the Relational Bricks domain, we give an informal reasoning on why applying the Split expressions on the domain is bound to terminate without sacrificing precision.

The second observation implies that we can divide the set of possible string values that t can take into multiple sets and look at them individually. For example, if a Split expression must hold for $\mathcal{B}[x]^{0,2}$, it must hold for both, $\mathcal{B}[x]^{0,1}$ and $\mathcal{B}[x]^{2,2}$, which both represent a subset of strings that are represented by the original Brick element.

Consider the example given in Figure 3.5. Suppose the specification of the web service requires v to be a list of comma-separated positive numbers. The Split expression making this statement would be $\uparrow (v, \boxplus, \mathcal{A}(\boxplus, p \rightarrow p \geq 0))$. While v can take infinitely many different string values, the representation in the Bricks domain remains finite.

```

1 var n := wall → ask number("positive_number")
2 var v := "0"
3 while n > 0 do
4   v := v // "," // n
5   n := wall → ask number("positive_number")
6 val base := "http://api.calculator.com/factorial?vector="
7 web → download(base // v)

```

Figure 3.5: Call to a fictitious web service located at `api.calculator.com`. The web service is supposed to compute the factorial for each element of a vector. The vector is supposed to be a comma-separated list of positive numbers. In the last line, the representation of v in the Bricks domain is $\mathcal{L} \left[\mathcal{B}[c[\boxplus]]^{1,1} \mathcal{B}[c[\boxplus] o [sv_1]]^{0,1} \mathcal{B}[c[\boxplus] o [sv_2]]^{0,\text{inf}} \right]$, where sv_1 is the symbolic variable created for the first assignment to n and sv_2 is the symbolic variable created for all assignments to n inside the loop.

According to the second observation, we can evaluate the domains

$$\mathcal{L} \left[\mathcal{B}[\{c[\boxplus]\}]^{1,1} \mathcal{B}[\{c[\boxplus] o [sv_1]\}]^{1,1} \mathcal{B}[\{c[\boxplus] o [sv_2]\}]^{0,\text{inf}} \right]$$

$$\mathcal{L} \left[\mathcal{B}[\{c[\boxplus]\}]^{1,1} \mathcal{B}[\{c[\boxplus] o [sv_2]\}]^{0,\text{inf}} \right]$$

separately. Evaluation of the second domain can again be separated as

$$\mathcal{L} \left[\mathcal{B}[\{c[\boxplus]\}]^{1,1} \right]$$

$$\mathcal{L} \left[\mathcal{B}[\{c[\boxplus]\}]^{1,1} \mathcal{B}[\{c[\boxplus] o [sv_2]\}]^{1,\text{inf}} \right]$$

Let us look at the last part of the separation to visualize the impact of the first observation. The comma is the first character in the second Brick

element. Hence, the left hand side of the first split is the concrete string $\overline{0}$ while the right hand side is again a set of infinitely many strings, represented by the Brick domain

$$\mathcal{L} \left[\mathcal{B} [\{o [sv_2]\}]^{1,1} \mathcal{B} [\{c [\overline{0}] o [sv_2]\}]^{0,\text{inf}} \right]$$

The second observation can be applied. Strings represented by this Bricks domain only contain a comma, if the second Brick element appears at least once. Those splits reduce to the left hand side

$$\mathcal{B} [\{o [sv_2]\}]^{1,1}$$

and the right hand side

$$\mathcal{L} \left[\{\mathcal{B} [o [sv_2]]\}^{1,1} \mathcal{B} [\{c [\overline{0}] o [sv_2]\}]^{0,\text{inf}} \right]$$

The analysis reached a repeating point; the last Bricks domain has already been evaluated and the first domain does not contain a comma. Because Split expressions evaluate deterministically, the Bricks domain can stop splitting once a repeating point is reached.

3.4.2 Split Operation on Relational Bricks

We define a Split Helper function $\Phi : \mathcal{D} \times c \subset \mathbb{S} \rightarrow 2^{\mathcal{D} \times \mathcal{D}}$ that takes an string domain, and a character and returns a set of tuples of string domains. The set c is defined as

$$c := \{s \mid s \in \mathbb{S} \wedge \text{count}(s) = 1\} \quad (3.22)$$

While defining the concrete Split expressions for split symbols of arbitrary length does not pose a problem, working with non-character symbols in the Bricks domain adds some difficulties. Detecting substrings of arbitrary length is much harder than detecting single characters. For example, the Bricks domain

$$\mathcal{L} \left[\mathcal{B} [\{\overline{apple}\}]^{1,1} \mathcal{B} [\{\overline{pie}\}]^{0,1} \right]$$

contains the string \overline{ep} . A different example where detecting strings of arbitrary length is harder would be the Bricks element

$$\mathcal{B} [\{\overline{mad}, \overline{ogre}\}]^{0,2}$$

which can be split at the substring \overline{dog} even though no individual element of the set of strings in the Brick element contains the substring. If we would allow delimiters of arbitrary length, we would always need to look at the Bricks domain as a whole. Imposing the restriction that the delimiter must have length 1, we can employ a divide and conquer approach and define

As already mentioned in the beginning of this section, let $\sigma \in c$. Note that this does not pose a great restriction regarding the analysis of URL strings as the important URL delimiters are single characters.

The helper function is defined for the Abstract String domain, the Brick elements and the Bricks domain but can be extended to other string domains. Restricting the length of σ to be 1, we can define Φ in a divide and conquer manner for each domain separately.

Abstract Strings

The helper function Φ is defined for Abstract Strings in the following way:

$$\Phi(\top_{\mathcal{S}_A}, \sigma) := \{([\top_{\mathcal{S}_A}[\mathcal{M}[\overline{\{\sigma\}}]]], \top_{\mathcal{S}_A}), ([\top_{\mathcal{S}_A}[\mathcal{M}[\overline{\{\sigma\}}]]], \perp_{\mathcal{S}_A})\} \quad (3.23)$$

$$\Phi(\perp_{\mathcal{S}_A}, \sigma) := \{(\perp_{\mathcal{S}_A}, \perp_{\mathcal{S}_A})\} \quad (3.24)$$

$$\forall \sigma \forall M : \Phi([\top_{\mathcal{S}_A}[M]], \sigma) :=$$

$$\begin{cases} \{([\top_{\mathcal{S}_A}[M]], \perp_{\mathcal{S}_A})\} & \text{if } \mathcal{M}[\overline{\{\sigma\}}] \sqcap M = M \\ \{([\top_{\mathcal{S}_A}[M \sqcap \mathcal{M}[\overline{\{\sigma\}}]]], \top_{\mathcal{S}_A}), ([\top_{\mathcal{S}_A}[M \sqcap \mathcal{M}[\overline{\{\sigma\}}]]], \perp_{\mathcal{S}_A})\} & \text{otherwise} \end{cases} \quad (3.25)$$

$$\forall \sigma \forall p = (p_0, \dots, p_k, \sigma, p_{k+2}, \dots, p_m), \text{indexof}(p, \sigma) = k : \quad (3.26)$$

$$\Phi(c[p], \sigma) := \{c[(p_0, \dots, p_k)], c[(p_{k+2}, \dots, p_m)]\}$$

$$\forall \sigma \forall p, \text{indexof}(p, \sigma) < 0 : \Phi(c[p], \sigma) := \{c[p], \perp_{\mathcal{S}_A}\} \quad (3.27)$$

$$\Phi(o[v], \sigma) :=$$

$$\begin{cases} \{(o[v], \perp_{\mathcal{S}_A}), ([\top_{\mathcal{S}_A}[\mathcal{M}[\overline{\{\sigma\}}]]], [\top_{\mathcal{S}_A}[\mathcal{M}[o[v]]]])\} & \text{if } \sigma \text{ may be contained in } o[v] \\ \{(o[v], \perp_{\mathcal{S}_A})\} & \text{otherwise} \end{cases} \quad (3.28)$$

$$\forall \sigma \forall a = a_1 a_2 \dots a_n \in \mathcal{S}_A : \Phi(a, \sigma) :=$$

$$\begin{aligned} & \{(a_1 \dots a_{k-1} \oplus x_1, x_2 \oplus a_{k+1} \dots a_n) \mid \forall i \in [1, k] \forall x \in \mathcal{S}_A : \\ & \{(x, \perp_{\mathcal{S}_A})\} \neq \Phi(a_i, \sigma) \wedge (x_1, x_2) \in \Phi(a_k, \sigma)\} \end{aligned} \quad (3.29)$$

The first two equations define the Split operation on the top and the bottom element of the Abstract String lattice. The Maybe Contained Character domain can be employed to capture the information that the left side of the split does not contain the split delimiter. Equation 3.25 also uses the Maybe Contained Character domain to give a more precise split: If the delimiter is guaranteed not to be contained in the string value then the right hand side of the split is the bottom element. Otherwise, the Split operation works similar to the Split operation on $\top_{\mathcal{S}_A}$ with a more restricted Maybe Contained Character subdomain.

Equation 3.26 and Equation 3.27 define the splitting of Abstract String constants. The definition is straightforward: If the split delimiter is contained the constant is split at the first occurrence into two constants, if it is not contained the constant is not split and the right hand side is $\perp_{\mathcal{S}_A}$.

Equation 3.28 defines the Split expression on the last elementary Abstract String. As described in Section 3.2.3, other abstract strings carry information about maybe contained characters. This information is stored when the value of the symbolic variable is set. Consider an example program that creates a positive integer variable and wants to split the string representation of this variable at the decimal point. Since it is an integer variable, the string representation may not contain a decimal point and the Split operation returns a set containing only the tuple $(o[x], \perp_{\mathcal{S}_A})$. In a different example, the string representation of an arbitrary numeric value is split at the decimal delimiter; the analysis does not know whether the decimal point is contained or not, hence it returns the set

$$\{(o[x], \perp_{\mathcal{S}_A}), ([\top_{\mathcal{S}_A}[\mathcal{M}[\{\{\emptyset, \dots, \emptyset, \emptyset\}]]], [\top_{\mathcal{S}_A}[\mathcal{M}[\{\{\emptyset, \dots, \emptyset, \emptyset, \emptyset\}]]]])\}$$

Brick Elements

$$\Phi(\top_{\mathcal{B}}, \sigma) := \{(\top_{\mathcal{L}}, \perp_{\mathcal{L}}), (\top_{\mathcal{L}}, \top_{\mathcal{L}})\} \quad (3.30)$$

$$\Phi(\perp_{\mathcal{B}}, \sigma) := \{(\perp_{\mathcal{L}}, \perp_{\mathcal{L}})\} \quad (3.31)$$

$$\begin{aligned} \Phi(\mathcal{B}[S]^{1,1}, \sigma) := \\ \{(\mathcal{L}[\mathcal{B}[\{x_1\}]^{1,1}], \mathcal{L}[\mathcal{B}[\{x_2\}]^{1,1}]) \mid (x_1, x_2) \in \Phi(s, \sigma), s \in S\} \end{aligned} \quad (3.32)$$

$$\begin{aligned} \Phi(\mathcal{B}[S]^{0,1}, \sigma) := \\ \{(\mathcal{L}[\mathcal{B}[\{\emptyset\}]^{1,1}], \perp_{\mathcal{L}})\} \cup \{(\mathcal{L}[\mathcal{B}[\{x_1\}]^{1,1}], \mathcal{L}[\mathcal{B}[\{x_2\}]^{1,1}]) \mid (x_1, x_2) \in \Phi(s, \sigma), s \in S\} \end{aligned} \quad (3.33)$$

$$\begin{aligned} \forall M > 1 \forall S \subseteq \mathcal{S}_A \forall x : (x, \perp_{\mathcal{S}_A}) \notin \Phi(s, \sigma) \wedge \Phi(\mathcal{B}[S]^{0,M}, \sigma) := \\ \{(\mathcal{L}[\mathcal{B}[\{\emptyset\}]^{1,1}], \perp_{\mathcal{L}})\} \cup \\ \{(\mathcal{L}[\mathcal{B}[\{x_1\}]^{1,1}], \mathcal{L}[\mathcal{B}[\{x_2\}]^{1,1} \mathcal{B}[S]^{0,M-1}]) \mid (x_1, x_2) \in \Phi(s, \sigma), s \in S\} \end{aligned} \quad (3.34)$$

Bricks that are not covered by the above definition can conservatively be split the same way the top element is split. Note that normalization of the Bricks domain containing the Brick element in question will remove any $\mathcal{B}[S]^{m,M}$ where $m \geq 1$, and replace it with the sequence $\mathcal{B}[S]^{m,m} \mathcal{B}[S]^{0,M-m}$.

The first two equations depict the splitting of the top and the bottom Bricks element. The definition is fairly natural: The bottom element will

never contain the character c while the top element may or may not contain c .

Equation 3.32 covers all Brick elements that represent strings containing exactly one string element from the set. This kind of Brick element is fairly common and conveys very precise information. The actual definition of the splitting operation for Brick elements depends on the definition of the operation for the Abstract String domain. Nonetheless, the next equation gives a concrete example on how to split such a Brick element. Because the sets inside the Brick elements are finite and usually small, the Split operation should not explode in most real cases.

$$\begin{aligned} \Phi \left(\mathcal{B} [\{\overline{ruby}, \overline{topaz}, \overline{malachite}\}]^{1,1}, \overline{\text{m}} \right) = \\ \left\{ \left(\mathcal{L} \left[\mathcal{B} [\{\overline{ruby}\}]^{1,1} \right], \perp_{\mathcal{L}} \right), \right. \\ \left(\mathcal{L} \left[\mathcal{B} [\{\overline{top}\}]^{1,1} \right], \mathcal{L} \left[\mathcal{B} [\{\overline{\text{z}}\}]^{1,1} \right] \right), \\ \left. \left(\mathcal{L} \left[\mathcal{B} [\{\overline{\text{m}}\}]^{1,1} \right], \mathcal{L} \left[\mathcal{B} [\{\overline{achite}\}]^{1,1} \right] \right) \right\} \end{aligned} \quad (3.35)$$

Equation 3.33 is just an application of the second observation described in Section 3.4.1. The Brick element is simply split into the two Brick elements which together cover the whole set of strings that the original element covers.

The last equation describes how Brick elements for concrete string values of arbitrary length that must contain the delimiter character are to be split. The following equation gives a concrete example.

$$\begin{aligned} \Phi \left(\mathcal{B} [\{\overline{departure=zurich}, \overline{arrival=bern}\}]^{0,5}, \overline{\text{=}} \right) = \\ \left\{ \left(\mathcal{L} \left[\mathcal{B} [\{\emptyset\}]^{1,1} \right], \perp_{\mathcal{L}} \right) \right. \\ \left(\mathcal{L} \left[\mathcal{B} [\{\overline{departure}\}]^{1,1} \right], \mathcal{L} \left[\mathcal{B} [\{\overline{zurich}\}]^{1,1} \mathcal{B} [1, 1]^{0,4} \right] \right), \\ \left. \left(\mathcal{L} \left[\mathcal{B} [\{\overline{arrival}\}]^{1,1} \right], \mathcal{L} \left[\mathcal{B} [\{\overline{bern}\}]^{1,1} \mathcal{B} [1, 1]^{0,4} \right] \right) \right\} \end{aligned} \quad (3.36)$$

The analysis is always allowed to combine multiple tuples in the set using the least upper bound operator. For example, the analysis can represent the splits in the last equation as

$$\begin{aligned} \Phi \left(\mathcal{B} [\{\overline{departure=zurich}, \overline{arrival=bern}\}]^{0,5}, \overline{\text{=}} \right) = \\ \left\{ \left(\mathcal{L} \left[\mathcal{B} [\{\emptyset\}]^{1,1} \right], \perp_{\mathcal{L}} \right) \right. \\ \left. \left(\mathcal{L} \left[\mathcal{B} [\{\overline{departure}, \overline{arrival}\}]^{1,1} \right], \mathcal{L} \left[\mathcal{B} [\{\overline{zurich}, \overline{bern}\}]^{1,1} \mathcal{B} [1, 1]^{0,4} \right] \right) \right\} \end{aligned} \quad (3.37)$$

```

1 var d := wall → ask string("Departure?")
2 var a := wall → ask string("Arrival?")
3 var base := "http://transport.opendata.ch/v1/connections?"
4 var query := "to=" // a // "&from=" // d
5 web → download json(base // query)

```

(a) Sample code that requests public transport connections between two input cities. The value of `query` is over approximated by the Bricks domain $\mathcal{L} \left[\mathcal{B} \left[\{c \left[\overline{\text{to}} \right] \} \right]^{1,1} \top_{\mathcal{B}} \mathcal{B} \left[\{c \left[\overline{\text{from}} \right] \} \right]^{1,1} \top_{\mathcal{B}} \right]$

```

1 var d := wall → ask string("Departure?")
2 var a := wall → ask string("Arrival?")
3 var base := "http://transport.opendata.ch/v1/connections?"
4 var query := "to=" // web → url encode(a)
5 query := query // "&from=" // web → url encode(d)
6 for 0 ≤ i < 5 do
7   var v := wall → ask string("via")
8   if not v → is empty then
9     query := query // "&via[" // web → url encode(v)
10 web → download json(base // query)

```

(b) Sample code that requests public transport connections between two input cities. The program allows up to five stopovers. Each input string is URL encoded before use. The value of `query` in the last line is over approximated by the Bricks domain $\mathcal{L} \left[\mathcal{B} \left[\{c \left[\overline{\text{to}} \right] \left[\top_{S_A} [W] \right] c \left[\overline{\text{from}} \right] \left[\top_{S_A} [W] \right] \} \right]^{1,1} \mathcal{B} \left[\{c \left[\overline{\text{via}} \right] \left[\top_{S_A} [W] \right] \} \right]^{0,5} \right]$

where $W = \mathcal{M} \left[\left\{ \overline{\text{Z}}, \overline{\text{E}}, \overline{\text{E}}, \dots \right\} \right]$

Figure 3.6

The loss in precision due to the least upper bound operator materializes in the 1,1 -Brick elements. After this reduction, the analysis is no longer able to match $\overline{\text{departure}}$ and $\overline{\text{zurich}}$, and $\overline{\text{arrival}}$ and $\overline{\text{bern}}$ together.

Brick Domains

Finally, we define the helper function for Bricks domains. A Bricks domain consisting of a single Brick element represents the same strings the Bricks element represents. Let $b_1, \dots, b_n \in \mathcal{L}$:

$$\Phi(\top_{\mathcal{L}}, \sigma) := \Phi(\top_{\mathcal{B}}, \sigma) \quad (3.38)$$

$$\Phi(\perp_{\mathcal{L}}, \sigma) := \Phi(\perp_{\mathcal{B}}, \sigma) \quad (3.39)$$

$$\Phi(\mathcal{L}[\mathcal{B}[S]^{m,M}], \sigma) := \Phi(\mathcal{B}[S]^{m,M}, \sigma) \quad (3.40)$$

$$\begin{aligned} \Phi(\mathcal{L}[b_1 \dots b_n], \sigma) := & \\ & \bigcup_{i \in [1, n]} \left\{ (\mathcal{L}[b_1 \dots b_{i-1}] \oplus x, y \oplus \mathcal{L}[b_{i+1} \dots b_n]) \mid \right. \\ & \left. \forall j \in [1, i] \forall z \in \mathcal{L} : (z, \perp_{\mathcal{L}}) \in \Phi(b_j, \sigma) \wedge (x, y) \in \Phi(b_i, \sigma) \right\} \end{aligned} \quad (3.41)$$

To get a better understanding of the Split operation we are going to apply the Split operation at the examples given in Figure 3.6. Let us evaluate $\Phi(\text{query}, \mathbb{E})$.

In the example given in Figure 3.6a, the input strings are not URL encoded. Hence, the string may or may not contain the delimiter \mathbb{E} . The Split operator evaluates to

$$\begin{aligned} \Phi(\text{query}, \mathbb{E}) = & \\ & \left\{ \left(\mathcal{L} \left[\mathcal{B} \left[\{c \text{ [to=]} [\top_{S_A}[\mathcal{M}[\mathbb{E}]]]\} \right]^{1,1} \right], \mathcal{L} \left[\top_{\mathcal{B}} \mathcal{B} \left[\{c \text{ [from=]} o[sv_2]\} \right]^{1,1} \right] \right), \right. \\ & \left. \left(\mathcal{L} \left[\mathcal{B} \left[\{c \text{ [to=]} [\top_{S_A}[\mathcal{M}[\mathbb{E}]]]\} \right]^{1,1} \right], \mathcal{L} \left[\mathcal{B} \left[\{c \text{ [from=]} \} \right]^{1,1} \top_{\mathcal{B}} \right] \right) \right\} \end{aligned} \quad (3.42)$$

The example again emphasizes the importance of the Maybe Contained Character domain for the precision of the analysis. The second example, given in Figure 3.6b can be split much more precisely. The Split operator for this example, given $W = \mathcal{M}[\{\mathbb{E}, \mathbb{Z}, \dots\}]$ evaluates to

$$\begin{aligned} \Phi(\text{query}, \mathbb{E}) = & \\ & \left\{ \left(\mathcal{L} \left[\mathcal{B} \left[\{c \text{ [to=]} [\top_{S_A}[W]]\} \right]^{1,1} \right], \right. \\ & \left. \mathcal{L} \left[\mathcal{B} \left[\{c \text{ [from=]} [\top_{S_A}[W]]\} \right]^{1,1} \mathcal{B} \left[\{c \text{ [via=]} [\top_{S_A}[W]]\} \right]^{0,5} \right] \right) \right\} \end{aligned} \quad (3.43)$$

Let us continue the last example. An important part of analyzing URL strings is splitting the query at each occurrence of an ampersand. At some point the analysis would need to apply the Split operation to the right hand

side of the first split.

$$\begin{aligned}
& \Phi \left(\mathcal{L} \left[\mathcal{B} \left[\{c \text{ [from=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{1,1} \mathcal{B} \left[\{c \text{ [via/=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{0,5} \right], \mathcal{B} \right) = \\
& \left\{ \left(\mathcal{L} \left[\mathcal{B} \left[\{c \text{ [from=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{1,1} \right], \perp_{\mathcal{L}} \right) \right. \\
& \left. \left(\mathcal{L} \left[\mathcal{B} \left[\{c \text{ [from=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{1,1} \right], \mathcal{L} \left[\mathcal{B} \left[\{c \text{ [via/=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{1,1} \mathcal{B} \left[\{c \text{ [E'via/=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{0,4} \right] \right) \right\} \\
& \tag{3.44}
\end{aligned}$$

A repeating point is reached with the next split of the right hand side.

$$\begin{aligned}
& \Phi \left(\mathcal{L} \left[\mathcal{B} \left[\{c \text{ [via/=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{1,1} \mathcal{B} \left[\{c \text{ [E'via/=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{0,4} \right], \mathcal{B} \right) = \\
& \left\{ \left(\mathcal{L} \left[\mathcal{B} \left[\{c \text{ [via/=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{1,1} \right], \right. \right. \\
& \left. \left. \mathcal{L} \left[\mathcal{B} \left[\{c \text{ [via/=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{1,1} \mathcal{B} \left[\{c \text{ [E'via/=]} [\top_{\mathcal{S}_A}[W]]\} \right]^{0,3} \right] \right) \right\} \\
& \tag{3.45}
\end{aligned}$$

3.4.3 Termination of Split Expressions

Let us provide an informal reasoning about why a Split expression needs only be evaluated finitely many times on an arbitrary Bricks domain. Since the Bricks domain consists of a sequence of Bricks elements, the claim can only hold, if it holds for any single Brick element. The statement holds trivially for $\perp_{\mathcal{B}}$, because the delimiter cannot be contained in an invalid string. The statement also holds for $\top_{\mathcal{B}}$, because when dividing an unknown string, the remainder is again unknown and we reached a repeating point as the same predicate gets applied. Because of normalization of Bricks domains, we only need to cover Brick elements in the form of $\mathcal{B} [\{s_1, \dots, s_n\}]^{1,1}$ and in the form of $\mathcal{B} [\{s_1, \dots, s_n\}]^{0,m}$. For the first Brick element, the statement holds if any contained Abstract String respects the statement. However, the statement is also trivially true for both kinds of Brick elements, if the delimiter is not contained. Hence, for the sake of the argument, assume the Brick element can contain the delimiter. The second Brick element can be reduced to the set

$$\mathcal{B} [\{s_1, \dots, s_n\}]^{0,m} = \left\{ \mathcal{L} \left[\mathcal{B} [\{s_1, \dots, s_n\}]^{1,1} \mathcal{B} [\{s_1, \dots, s_n\}]^{0,m-1} \right], \mathcal{B} [\emptyset]^{0,0} \right\} \tag{3.46}$$

For a finite m , the statement holds inductively. However, a repeating point is reached after the second iteration. Hence, the statement also holds for infinite m .

Now we need to show that the statement also holds for every Abstract String. Trivially, it holds for $\perp_{\mathcal{S}_A}$. Furthermore, it holds for $\top_{\mathcal{S}_A}$ for the

same reasons it holds for $\top_{\mathcal{B}}$. The statement also trivially holds for any Abstract String constant, because they are just a wrapper for concrete string constants. The right hand side of the division of $[\top_{\mathcal{S}_A}[M]]$ is either $[\top_{\mathcal{S}_A}[M]]$, if the delimiter may be contained in M , or $\perp_{\mathcal{S}_A}$ otherwise. In the first case, a repeating point is reached, in the second case, the argument is trivially true. When dividing $o[x]$, the right hand side is either $[\top_{\mathcal{S}_A}[\mathcal{M}[o[x]]]]$ or $\perp_{\mathcal{S}_A}$ and the statement holds. Because the length of a sequence of Abstract Strings is finite (otherwise it would need an infinite amount of memory), since every Abstract String that is contained in the sequence guarantees termination, termination is guaranteed for all Abstract Strings.

We reasoned that the Split expression is bound to terminate on Abstract Strings. We further statet, that the Split expression is bound to terminate on $\mathcal{B}[\{s_1, \dots, s_n\}]^{1,1}$, if it terminates on all Abstract Strings. Hence, it must terminate on all Brick elements and the necessary termination condition on Bricks domains is fulfilled. Since we restrict the delimiter to strings of length 1, each Brick element in a Bricks domain can be observed separately. Hence, the statement is also implied on the Bricks domain given it holds on each individual Brick.

3.5 Web Service Specification

An URL string by itself is just a regular string. The semantics of a request to a web service given an URL string only become apparent when looking at the web service specification. The specification imposes constraints on the value of the URL string and on the value of the response string. This section defines a language for an intermediate representation of web service specification, and describes the semantics that are induced by the web service specification.

In general, the specification of a method tells about the semantics of the method without revealing the internal implementation. This fact makes modular programming possible; the implementation of a method can be modified without affecting the semantics of a caller who must satisfy the contracts of the specification. Without specification, there are no guarantees about the behavior of a method call and the caller does not know how to call the method. Hence, every web service has to provide some form of specification. There is no unified standard of how to write web service specification. In this section we are going to introduce a representation of web service specification which is used by the analysis. The intermediate representation of the web service specification is inspired by the web application description language [19]. The two main additions are

1. The possibility to restrict the values of parameters beyond the type.
2. A more detailed description of the returned value.

Let us make a connection between calls to a web service and regular method calls. In object oriented programming, a regular method call consists of an identifier, a receiver and a list of parameters. Furthermore, the method call may return one or more (depending on the programming language) values. For simplicity we can treat the receiver the same as any other input parameter. The web services we are going to look at show similar features: They have an identifier, can take input parameters and return values. The identifier can be viewed as the host-part of the URL while we consider the input parameters to be given as a list of key-value pairs in the query-part of the URL. For example, the call of the web service at the URL *http://transport.opendata.ch/v1/connections?from=Lausanne&to=Bern* identifies the method *connections* with the two input parameters *from* and *to* taking the values *Lausanne* and *Bern*.

The specification of a method call can give restrictions on any value in the state before the method call that is visible to the method (Precondition). Furthermore, the specification can give a guarantee on any value in the state after the method call that is visible to the method (Postcondition). Since a web service is running in its own process, the web service does not know anything about the state of the caller. If the web service is stateless, no knowledge about the state of the caller implies that the postconditions may only give guarantees about the return values while the preconditions can only restrict the value of the input parameters. In the following we will restrict ourself to stateless web services. A stateless architecture style for web services called Representational State Transfer – REST for short – was introduced by Fielding [15]. The REST architectural style has since gained ground in the industry. Nonetheless, stateful web services still exist and their analysis might be interesting in future projects.

3.5.1 Intermediate Representation of Web Service Specification

An intermediate representation of web service specification must be able to express the following:

- The location of the web service, e.g., <http://api.nytimes.com/svc/events/v2/listings.json>
- A list of required and optional parameters with their key, i.e., identifier.
- Restrictions on the type and the value of the parameters.
- Conditional guarantees on the value of the returned string. We are interested about structural guarantees provided as annotations to strings (see Section 3.6). The guarantees can be conditioned on the value of the actual input parameters, e.g., if there is a parameter with key

\overline{format} and value \overline{json} , the response is guaranteed to return the string representation of a JSON object.

The language defined in this section is inspired by WADL [19], a standard for web service specification in the machine processable XML-format. While WADL itself is extensible through XML schemas, there are some differences and some similarities between the original WADL and our defined language.

- Both, WADL and our language represent the location of the web service.
- WADL allows parameters to be specified anywhere in the URL string by introducing template. For simplicity, we want parameters to only appear in the query part of the URL, i.e., after the question mark.
- Both, WADL and our language allow specifying type restrictions on parameters, and restriction of the value to a set of concrete inputs.
- Our language is more expressive in specifying restrictions on the values of parameters than WADL. Our language can restrict values to a range of values, which is particularly useful for numeric parameters. Our language can also restrict the values to being a list of values, or a tuple of values that is separated by a specified delimiter. For example, the language can the value of a parameter to be a comma separated tuple of numeric values with their individual restrictions. Lastly, the language can also declare that no guarantees can be made about the validity of an actual parameter. This unknown constraint is useful in case we must assume a request to fail for any input value, e.g., when the input parameter is an API key for which we cannot statically know the validity.
- WADL can specify the type of the response and condition it on a single parameter. Our language further provides structural annotations for the response and can condition the response on multiple input parameters.

Let us define the language for the intermediate representation of the web service specification.

$$G := (\Sigma_N, \Sigma_T, R, S) \tag{3.47}$$

$$\Sigma_N := \{S, H, P, Q, P_2, R, R_v, R_T, R_L, O, C, X\}$$

$$\Sigma_T := \{ident, typ, q, n_+, n, \\ \text{withParameters, returns, required, optional, parameter, st, value,} \\ \text{element, each, rd, is, unrestricted, isTupleSeparatedBy, when,} \\ \text{isListSeparatedBy, default, success, fail, or, ::, (,), ; \}$$

$$R := \{ \\ S \mapsto H \text{ withParameters } P \text{ returns } Q \\ H \mapsto ident \\ P \mapsto \text{required } P_2 \mid \text{optional } P_2 \mid P; P \\ P_2 \mapsto \text{parameter } ident :: typ \text{ st } R_v \\ R_v \mapsto \text{value } R \\ R_T \mapsto n_+ \text{ rd element } R \mid n_+ \text{ rd element } :: typ R \mid R_T, R_T \\ R_L \mapsto \text{each element } R \mid \text{each element } :: typ R \\ R \mapsto \text{in } (n; n) \mid \text{in } (O) \mid \text{is unrestricted} \mid \text{is unknown} \mid \text{isTupleSeparatedBy } ident \text{ st } R_T \mid \\ \text{isListSeparatedBy } ident \text{ st } R_L \\ O \mapsto O; O \mid ident \\ Q \mapsto \text{when } CX \mid \text{default } X \mid \text{when } C \text{ or default } X \\ C \mapsto C \text{ or } C \mid \text{parameter } ident R \\ X \mapsto \text{success } q \text{ fail } q \quad \}$$

The grammar in Equation 3.47 defines a language to represent web service specification. The first six terminal symbols in Σ_T are placeholders for a much larger set of possible values. The placeholder *ident* refers to all possible identifiers, i.e., various string values. The placeholder *typ* refers to a set of types that can be used to annotate parameters of the web service. The placeholder *q* represents structural information about a string value. Lastly, *n* can be replaced by any numeric value while *n₊* can take any positive integer value.

The meaning of the web service specification language becomes apparent when looking at the set of rules *R*. The specification consists of an identifier for the host (*H*), some preconditions on the parameters (*P*) and some postconditions giving guarantees about the return value (*Q*). Parameters are marked as either required or optional. Each parameter has an identifier and is restricted by a type and a semantic restriction on the value (*R_v*). The latter takes one of the following forms:

- An interval restriction, requiring the actual parameter to be within a numeric interval

- An option restriction, requiring the actual parameter to take one value from a set of given options
- A tuple restriction, requiring the actual parameter to be a tuple with a further semantic restriction for each tuple element
- A list restriction, requiring the actual parameter to be a list with a semantic restriction for each list element
- Not restricted – the actual parameter can take any value
- Unknown restriction – the web service is always expected to return an erroneous response. The unknown restriction comes in handy when the set of possible options is huge, e.g., when the parameter is required to take a valid city name as value. Furthermore, a lot of web services require an API key to be executed. Generally, valid API keys are not known statically.

Note the subtle difference between a list restriction and a tuple restriction: In a list restriction each list element must satisfy the same restriction and there is no information about the length of the list while a tuple restriction has a predefined length, an order among the elements and possibly a different restriction for each tuple element.

The guarantees on the return value can further be conditioned on the input parameters. For example, a web service may provide its output formatted as a JSON object or a XML tree based on whether the input parameter `format` takes on the value `json` or `xml`. Each guarantee about the output must further distinguish between success and failure of the execution. Basically, we expect every web service to possibly throw a checked exception. Note that inherently, every call to a web service can also throw an unchecked exception, e.g., if no connection to the web service can be established. In this case, when calling a web service in TouchDevelop, the `invalid` value is returned.

3.5.2 Derived Pre- and Postconditions

Let us call the requirements on the input parameters of a web service its success condition P_{succ} . Further, let us enumerate all possible conditional guarantees about the return value Q^1, \dots, Q^n with the i -th condition denoted P_i and the corresponding pair of guarantees named Q_{succ}^i and Q_{fail}^i . The following can be assumed about a request to the web service using the actual parameters p_1, \dots, p_m and receiving the response q :

$$\bigvee_i (P_{succ}(url, p_1, \dots, p_m) \wedge P_i(url, p_1, \dots, p_m) \wedge Q_{succ}^i(q) \vee \neg P_{succ}(url, p_1, \dots, p_m) \wedge P_i(url, p_1, \dots, p_m) \wedge Q_{fail}^i(q)) \quad (3.48)$$

Note that the actual parameters p_1, \dots, p_m do not have to be in a specific order. The enumeration is done to be able to talk about general parameters. Nonetheless, the parameters are identified by a name.

The precondition P_{succ} is generated from the web service specification for a given URL string. From the specification, a set of accepted parameters is known with value restrictions for each parameter. Informally, the generated precondition states that for each key-value pair in the query, if the key is equal to the name of a parameter, then the value must conform the restriction of said parameter. More formally, the generated precondition is

$$\begin{aligned}
P_{succ}(\mathbf{url}, p_1, \dots, p_m) := & \mathfrak{h}(\mathbf{url}, \mathbb{E}, ((h, L) \rightarrow \\
& \mathfrak{h}(L, \mathbb{E}, (\mathcal{A}(\mathbb{E}), e \rightarrow \mathfrak{h}(e, \mathbb{E}, ((k, v) \rightarrow \\
& k = p_1^{\text{name}} \rightarrow p_1^{\text{Restriction}}.P(v) \wedge \dots \wedge k = p_m^{\text{name}} \rightarrow p_m^{\text{Restriction}}.P(v))))))
\end{aligned} \tag{3.49}$$

A beautiful inherent property of the precondition predicate is, that it automatically normalizes the parameter list of the URL string. The application of the forall-splits quantifier with the delimiter \mathbb{E} removes the explicit ordering of the parameters. Hence, the value of the Split expression predicate is independent on the ordering of the parameters. More formally, given $b_i \neq ?$, $k_i = k_i^1, \dots, k_i^{m_i}$, $k_i^j \neq =$, $k_i^j \neq \&$, $v_i = v_i^1, \dots, v_i^{n_i}$, $v_i^j \neq =$, and $v_i^j \neq \&$:

$$\begin{aligned}
& \forall i \in \mathbb{N} : \\
& P_{succ}((b_1, \dots, b_n, ?, \\
& \quad k_0, =, v_0, \&, \dots, \&, \\
& \quad k_r, =, v_r), x_1, \dots, x_r) = \\
& P_{succ}((b_1, \dots, b_n, ?, \\
& \quad k_{i \bmod r}, =, v_{i \bmod r}, \&, \dots, \&, \\
& \quad k_{(i+r) \bmod r}, =, v_{(i+r) \bmod r}), x_1, \dots, x_r) =
\end{aligned} \tag{3.50}$$

p_x^{name} refers to the name of parameter x , and $p_x^{\text{restriction}}$ refers to the restriction of parameter x . Table 3.1 lists all restriction types and the predicate they generate. The predicate and the parameter name are both used in the precondition P_{succ} .

An example shall give a demonstration. Suppose a web service located at <http://example.com> requires the parameters *len*, a positive number, *key*, a valid api-key, and *format*, either \overline{json} or \overline{xml} . The parameter *format* is optional with \overline{json} being the default value. The text in Figure 3.7 expresses the corresponding specification in the defined language.

The placeholders S1, Q1, S2, and Q2 are structural string domains that are introduced in the next section. From the parameter list, the following

Restriction	Restriction.P(v)
Interval(min, max)	$v \geq \min \wedge v \leq \max$
Option($\{o_1, \dots, o_n\}$)	$v = o_1 \vee \dots \vee v = o_n$
NTuple(σ, r_1, \dots, r_n)	$\text{h} (v, \sigma, ((x_1, x'_2) \rightarrow r_1.P(x_1) \wedge$ $\text{h} (x'_2, \sigma, (x_2, x'_3) \rightarrow r_2.P(x_2) \wedge \dots))$
List(σ, r)	$\text{h} (v, \sigma, \mathcal{A}(\sigma, r.P))$
Unrestricted	True
Unknown	Nondeterministic True or False

Table 3.1: Collection of parameter restrictions and their generated boolean expression dependent on the parameter value v

```

http://example.com withParameters
  required parameter len::number st value in (0; inf)
  optional parameter format::string st value in (json; xml)
  required parameter key::string st value is unknown
returns
  when parameter format in (json) or default
    success S1 fail Q1
  when parameter format in (xml)
    success S2 fail Q2

```

Figure 3.7: Example of a fictitious web service specification in the defined language.

precondition is generated for an URL string `url`:

$$\begin{aligned}
P_{succ} = & \text{h} (\text{url}, \mathbb{Z}, ((h, L) \rightarrow \\
& \text{h} (L, \mathbb{E}, (\mathcal{A}(\mathbb{E}), e \rightarrow \text{h} (e, \mathbb{E}, ((k, v) \rightarrow \\
& k = \text{len} \rightarrow (v \geq 0) \wedge \\
& k = \text{format} \rightarrow (v = \text{json} \vee v = \text{xml}) \wedge \\
& k = \text{key} \rightarrow \text{nondeterministic True or False))))))
\end{aligned} \tag{3.51}$$

Furthermore, the following two result conditions are generated

$$\begin{aligned}
P_1 = & \text{h} (\text{url}, \mathbb{Z}, ((h, L) \rightarrow \\
& \text{h} (L, \mathbb{E}, (\mathcal{A}(\mathbb{E}), e \rightarrow \text{h} (e, \mathbb{E}, ((k, v) \rightarrow \\
& k = \text{format} \rightarrow v = \text{json}
\end{aligned} \tag{3.52}$$

(If the parameter with the name `format` exists, then its value must be `json`)

$$\begin{aligned}
P_2 = & \text{h} (\text{url}, \mathbb{Z}, ((h, L) \rightarrow \\
& \text{h} (L, \mathbb{E}, (\neg \mathcal{A}(\mathbb{E}), e \rightarrow \text{h} (e, \mathbb{E}, ((k, v) \rightarrow \\
& k \neq \text{format} \wedge v \neq \text{xml}
\end{aligned} \tag{3.53}$$

(The parameter with the name *format* must exist. It must take the value `xml`.)

Combining these conditions and placing them into Equation 3.48 yields the following assumption on the state:

$$(P_{succ} \wedge ((P_1 \wedge S1) \vee (P_2 \wedge S2))) \vee (\neg P_{succ} \wedge ((P_1 \wedge Q1) \vee (P_2 \wedge Q2))) \quad (3.54)$$

3.6 Structural Annotations for String Values

Many web services do not simply return a collection of data. For example, the event listings API by The New York Times returns a message containing the status, the copyright, the number of results, and an array of results, each containing various values such as an event identifier, and a description of the event. Hierarchical data formats such as XML and JSON are suited to contain this data. Nonetheless, communication between the client and the web service is based on passing string values.

Mainly, we are interested in annotating strings that represent XML or JSON objects with their structural information. Luckily, the structure of those two formats are very similar. Our annotations store the type of the represented object. Furthermore, the annotations represent a set of children that are guaranteed to exist.

The example given in Figure 3.8 motivates the needs for these annotations. The pictured program creates a call to The New York Times event listings API and stores the returned string in variable `s`. If `s` is valid, the program parses the string as a JSON object in line 4. But how does our analysis know that the string can be parsed as a JSON object? How does it know whether the created object has a field *status*? Domains that capture the value of a string as a sequence of characters, i.e., analyze the string purely syntactically, such as the Bricks domain do not capture this information directly. The analysis had to apply the parse semantics at the abstract string value. Furthermore, even if we can represent the string as a sequence of characters would the domain capture a lot of irrelevant information and most likely be very costly.

Instead of tracking information about strings as sequence of characters, we want to annotate strings that are returned by a web service with structural annotations. For those annotations, we propose a domain in the abstract interpretation framework. The top element in the domain is an unknown structure. The unknown structure conveys no information about the string. We also define a bottom element with the additional purpose of marking invalid string values.

The annotation for a hierarchical structure has a name, and a type (such as JSON or XML), a possible value, information about whether it is an array-like object (T – true, or F – false, and contains a set of structures, the children. Mathematically, the structural annotation is a 4-tuple

(N, T, V, A, C) . The type tells the analysis that the string can be parsed into an object of this type. The children guarantee which fields can be accessed after the object is created. Say, we want to prove that `status` is valid at the end of the example in Figure 3.8. The annotation of the string value of `web→download(url)` is produced by the web service specifications. For a successful API request, i.e., a request that is true to the constraints on the parameters, a simplified structural annotation could be represented as:

$$\left(\begin{array}{l} \underline{ROOT}, \text{JSON}, \top, F, \{ \\ \quad (\underline{status}, \text{String}, \underline{OK}, F, \emptyset), \\ \quad (\underline{num_results}, \text{Integer}, \{x : x \geq 0\}, F, \emptyset) \\ \quad (\underline{results}, \text{JSON}, \top, T, \{ \\ \quad \quad (\underline{event_id}, \text{Integer}, \top, F), \\ \quad \quad \dots \\ \quad \quad \}) \} \\ \} \} \end{array} \right)$$

The analysis may determine a set of possible return structures. In our example, the analysis cannot guarantee that the preconditions of the web service are satisfied, because the API-key cannot be statically verified. Other times, an analysis may not be able to decide whether a JSON or an XML object is returned. In those cases, a least upper bound operator is needed to join the possible annotations. Equation 3.55 defines the least upper bound operator on the structural string annotations. Two annotations with a different name are not comparable and the least upper bound is the top element. The name does further function as key, which is needed to define the least upper bound operator on the children. In the definition, $T_1 \sqcup T_2$ is the least upper bound operator as defined on the types, and $V_1 \sqcup V_2$ is the least upper bound operator on the values, which are just a set domain of expressions. Both domains are not defined by us.

$$(N, T_1, V_1, A, C_1) \sqcup (N, T_2, V_2, A, C_2) := (N, T_1 \sqcup T_2, V_1 \sqcup V_2, A, C_1 \sqcup C_2) \quad (3.55)$$

$$\forall N_1, N_2, A_1, A_2 : (N_1 \neq N_2 \vee A_1 \neq A_2) \rightarrow (N_1, T_1, V_1, A_1, C_1) \sqcup (N_2, T_2, V_2, A_2, C_2) := \top \quad (3.56)$$

Functionally, the children are an inverse set domain of structures; the least upper bound of the two children $\{a, b\}$ and $\{b, c\}$ is $\{b\}$. The idea is the following: A structure with child a shall guarantee that the corresponding collection object contains the value a . Equation 3.57 gives a formal definition of the least upper bound operator on the children. Note that, since the name of the structure is also its key in the inverse set domain, no two structures in

```

1 var url := "http://api.nytimes.com/svc/events/v2/listings.json
   ?&ll=40.756146,73.99021&api-key=" // data → key
2 var s := web → download(url)
3 if not s → is invalid then
4   var js := web → json(s)
5   var status := js → string("status")

```

Figure 3.8: Download of a listing of events from The New York Times API. The results are sent in JSON format.

the same set can have the same name. The least upper bound operator only keeps structures of which the name is present in both sets that are joined. Those structures are then joined according to the definition in Equation 3.55.

$$\begin{aligned}
& \left\{ (N_1^1, T_1^1, V_1^1, A_1^1, C_1^1), \right. \\
& \quad (N_1^2, T_1^2, V_1^2, A_1^2, C_1^2), \\
& \quad \dots, \\
& \quad \left. (N_1^m, T_1^m, V_1^m, A_1^m, C_1^m) \right\} \sqcup \\
& \left\{ (N_2^1, T_2^1, V_2^1, A_2^1, C_2^1), \right. \\
& \quad (N_2^2, T_2^2, V_2^2, A_2^2, C_2^2), \\
& \quad \dots, \\
& \quad \left. (N_2^m, T_2^m, V_2^m, A_2^m, C_2^m) \right\} := \\
& \left\{ x \sqcup y \mid \exists i, j : \right. \\
& \quad \left. x = (N_1^i, T_1^i, V_1^i, A_1^i, C_1^i) \wedge y = (N_2^j, T_2^j, V_2^j, A_2^j, C_2^j) \wedge N_1^i = N_2^j \right\} \\
& \tag{3.57}
\end{aligned}$$

The information about whether the structure is an array or not defines how the children are accessed. The *results*-element in our example is an array containing various events. There are no guarantees about the number of elements the array contains, but each element in the array *results* is a JSON object containing the value field *event_id* and so on.

Chapter 4

Evaluation

In this chapter, we want to demonstrate how our analysis works on TouchDevelop scripts. Note that a sound analysis will report every error in the program. Hence, when analyzing TouchDevelop scripts with our proposed analysis disabled, every error is reported. However, we expect this conservative analysis to be imprecise. By enabling our analysis, but keeping the setup otherwise identical, we expect less errors to be reported.

The goal of the evaluation is

- to gain insight about constructs where our analysis is able to improve precision over the existing analyses.
- to find shortcomings of our analysis, i.e., constructs where our analysis is not precise enough.
- to provide an estimation on the cost of our analysis.
- to gain insight about web service related bugs in real programs.

We evaluated the developed analysis against some real TouchDevelop programs that were written by other people. All TouchDevelop scripts that were created and published in the official online programming environment provided by Microsoft are publicly available. We preselected possibly relevant scripts by searching the published scripts for uses of

- `web→download`
- `web→download json`
- `web→download xml`
- `web→create request`

We randomly selected various scripts that communicate with at least one web service. Each published script has its own public identifier. Published

Script ID	# Warnings without anal- ysis	# Warnings with analysis	Execution time [ms] without analysis	Execution time [ms] with analysis
udch	10	9	5931	10 360
quqncjye	4	4	92	259
<u>izwjb</u>	3	2	101	119
frtd	9	9	27 261	34 288
ledzijdu	25	25	3129	6967
<u>lkpy</u>	21	21	3628	7646
<u>dewz</u>	21	17	4131	5232
clgpxcwx	12	11	1712	2238
ydsn	18	17	2479	3345
<u>xnjn</u>	17	16	2439	3354

Table 4.1: Results of the evaluation. The second column lists the number of warnings if our analysis is disabled, the third column lists the number of warnings with the analysis enabled. The last two columns show the best execution times of 5 executions. Some programs have genuine bugs; the underlined scripts are corrections to the previous script.

scripts can be installed and modified. Modified scripts are published with a new public identifier. However, a reference to the original script is stored. Often, the modified script does not differ too much from its original. Hence, we only want to evaluate one of the scripts derived from the same original, or the original script itself. The exception to this are scripts that we modified ourself to fix genuine bugs revealed by the analysis.

Each chosen TouchDevelop script is analyzed twice – once with our proposed analysis enabled and once with the analysis disabled, but with otherwise identical settings. We compare the returned warnings and the execution time of both runs. Table 4.1 shows an overview of our findings. The execution time is measured as the time in milliseconds it takes from initializing the analysis until the results are returned. The data provided in the table is the shortest out of 5 time measurements.

The proposed analysis can only provide value to the TouchBoost tools if web service specifications are provided. Unfortunately, as there is no unified standard on how to write web service specification, we had to create the specification in the intermediate representation proposed in Section 3.5 for each web service our evaluation scripts are using.

```

1 private action getRandomJokeWithName (firstName:String,lastName
   :String) returns joke:String
2   var s := "http://api.icndb.com/jokes/random?firstName=" //
   firstName // "&lastName=" // astName
3   var js := web→download json(s)
4   if (js→is_invalid) then
5     joke := "Unable to download json"
6   else
7     var value := js→field("value")
8     joke := value→string("joke")

```

Figure 4.1: Excerpt from the published TouchDevelop script with the public script identifier *udch*. The program communicates with The Internet Chuck Norris Database, a RESTful web API. The specification [3] guarantees that the response is a string representing a json object which contains the two fields *status* and *value*. Our analysis is able to guarantee that the variable *value* is valid, reducing the number of false alarms.

4.1 Results

The first example, which is given in Figure 4.1, uses the Internet Chuck Norris Database, a RESTful web API, to retrieve a random joke and change the word “Chuck” to the given first name, and the word “Norris” to the given last name. The specification guarantee that a JSON object containing the fields *value* and *status* is returned. Each string value for the two input parameters is valid. However, since the input strings are not URL encoded, the analysis cannot guarantee that the variable *joke* is valid at the end. Nevertheless, our analysis can guarantee that the call `value→string("joke")` does not operate on an invalid value. Hence, opposed to a fully conservative analysis without knowledge about the web service, our analysis does not raise an alarm in line 7.

The second example, which is given in Figure 4.2, retrieves a weather forecast for 5 days from the World Weather Online Free API. Our analysis gives a warning about every access of a JSON object, starting at line 10. However, those warnings are justified and reveal genuine bugs in the program. Firstly, the value of `js1` is not checked against an invalid value before line 10. Each request to a web service may potentially return an invalid value. Secondly, the value of `data→City` can literally take any string value and is never url encoded before it is used in the URL string in line 8. Consequently, the response is not guaranteed to be a JSON object. Assume a user inputs the string `Zürich&format=xml` when prompted to give a city. The URL string then contains the format-parameter twice. According to the specification of the web service, the returned value is JSON object, if the format-parameter takes the value `json`, and an XML object, if the format-parameter takes the value `xml`. Simplified, the analysis uses the following

```

1 private action firstAction()
2   if (records → Weather_index → singleton → City → is_empty) then
3     data → City := wall → ask_string("Please type in the City")
4
5 private action checkTemp()
6   var url1 := "http://api.worldweatheronline.com/free/v1/
   weather.ashx?q="
7   var url2 := "&format=json&num_of_days=5&key=
   vxdr7kb2sbejx4fn7c7vk6cr"
8   var url := url1 // data → City // url2
9   var js1 := web → download json(url)
10  var js2 := js1 → field("data")
11  var js3 := js2 → field("current_condition")
12  var js4 := js3 → at(0.0)
13  data → tempC := js4 → string("temp_C")

```

Figure 4.2: Excerpt from the published TouchDevelop script with the public script identifier *ledzijdu*. The code communicates with the World Weather Online Free web API [4]. Our analysis reveals various bugs in the program.

semantics to represent this contract: If there exists a key-value pair where the key is equal to \overline{format} and the value is equal to \overline{json} , then the response is a JSON object, or if there exists a key-value pair where the key is equal to \overline{format} and the value is equal to \overline{xml} , then the response is an XML object. In this example, both conditions hold, hence the response can be either of the two. Equation 4.1 shows the actual split expression that implements the condition on the JSON response.

$$\begin{aligned}
& \text{split}(\text{url}, \overline{[]}, ((l, r) \rightarrow \\
& \quad \neg \text{split}(r, \overline{[]}, \mathcal{A}(\overline{[]}, x \rightarrow \\
& \quad \quad \text{split}(x, \overline{[]}, (k, v) \rightarrow k \neq \overline{format} \wedge v \neq \overline{json})))
\end{aligned} \tag{4.1}$$

We manually corrected the bugs in the example. The code snippet of the corrected program is shown in Figure 4.3. After encoding the input string, the analysis can guarantee that the response is a JSON object, because no additional key-value pair can exist. Furthermore, the program checks whether the communication with the web service was successful by checking the validity of the response. The analysis can guarantee that *js1* is a valid JSON object and does not report a warning when a field is accessed in line 14. However, the analysis is not able to guarantee that the field *current_condition* exists and gives a warning in line 15. The reason for this is that an unsuccessful response is not guaranteed to contain this field. Assume a user requested the weather for alpha-centauri. In this case, the web service returns

```
{ "data":
```



```
{ "error": [
  {"msg": "Unable to find any matching weather location
    to the query submitted!" } ] ]}}
```

This example showed a lot of cases where the analysis is able to discover bugs and why the bugs are discovered. A slight modification to the program code also makes shortcomings of static analysis visible: Assume we had statically determined that `data→City` takes the value `Zürich`. In this case, the request is guaranteed to be successful. However, it is very impractical to specify all valid values the parameter `q` can take. Furthermore, it is impossible to specify all valid values for the parameter `key`, because this would mean that the web service had revealed valid keys, completely ruining the purpose of the API key. Unfortunately, a lot of stateless web services employ an API key to deny individual users from excessively requesting their service. Hence, checking preconditions on the passed parameters can often not give guarantees on the success of the call.

```
1 private action firstAction()
2   if (records→Weather_index→singleton→City→is_empty) then
3     data→City := wall→ask_string("Please type in the City")
4
5 private action checkTemp()
6   var url1 := "http://api.worldweatheronline.com/free/v1/
7     weather.ashx?q="
8   var url2 := "&format=json&num_of_days=5&key=
9     vxdr7kb2sbejx4fn7c7vk6cr"
10  var url := url1 // web→url_encode(data→City) // url2
11  var js1 := web→download_json(url)
12  if js1→is_invalid then
13
14  else
15    var js2 := js1→field("data")
16    var js3 := js2→field("current_condition")
17    var js4 := js3→at(0.0)
18    data→tempC := js4→string("temp_C")
```

Figure 4.3: Manual correction of the previous example. The whole script has the public script identifier `dewz`.

Figure 4.4 shows a TouchDevelop script that communicates with a web service that does not require input parameters and accordingly has no preconditions. Our analysis is able to locate the host and retrieve information about the returned JSON object. Thereby, the analysis does not report a warning in line 12, as `addr` cannot take an invalid value. Note that the way the split expressions are defined, the analysis would have also been able to locate the host if the URL string did not contain a question mark and the optional parameter. If a string does not contain the split delimiter symbol, then the left hand side of the split is the string itself.

```

1 private action display info()
2   wall→clear
3   wall→set reversed(true)
4   if (web→is connected) then
5     ("Network_details_for_the_" // web→connection type // "
      connection:")→post to wall
6     var r := math→random(10000.0)
7     var data1 := web→download_json("http://ifconfig.me/all.
      json?" // r
8     if data1→is invalid then
9       "No_network_connection."→post to wall
10    else
11      var addr := data1→string("ip_addr")
12      var s := "IP_Address:_" // addr
13      code→word picture(s,72.0)→post to wall
14      __key_index := 0
15      __key_collection := data1→keys→copy
16      while ((__key_index) < (__key_collection→count)) do
17        if not __key_collection→at_index(__key_index)→equals(
          "ip_addr") then
18          (__key_collection→at_index(__key_index) // ":_" //
            data1→string(__key_collection→at_index(
              __key_index)))→post to wall
19          __key_index := __key_index + 1
20    else
21      "No_network_connection."→post to wall

```

Figure 4.4: Excerpt of a TouchDevelop script that requests the service of ifconfig.me [5]. This is a non-standard example, as it is a service that does not require input parameters and hence the caller has no preconditions to fulfill. The public identifier of the script is *rxbeb*.

4.2 Discussion

In this section we try to answer the questions that were implicitly posed at the beginning of this chapter. That is, we want to answer setups, where our analysis is precise enough, and setups where the analysis fails to prove correctness. Furthermore, we want to discuss the impact of our analysis on execution time, and what bugs real programmers made with respect to values related to calls to web services.

Analysis of calls to web services depends above all else on the accessibility of web service specification. Hence, an analysis must be able to determine which web service is invoked. Our analysis makes a simple check against the base path of each registered web service. The check compares the left hand side of the first occurrence of a question mark with the base path. If no question mark is contained in the URL string, the whole string is compared against the base path. Fortunately, in most cases, the base path can statically be determined in the URL string, because most scripts store

this information as a constant.

Once the specification are retrieved, the analysis checks the preconditions. Here, static analysis is often not beneficial, because a lot of stateless web services require an API key. Also, many parameters, such as city names, have unknown preconditions which must statically assumed to be violated.

On the other hand, the analysis is theoretically able to prove that the preconditions are always violated. However, dynamic testing with an arbitrary input would reveal such a bug. Hence, programs that are guaranteed to violate the preconditions should not occur in reality.

For the sake of the discussion, let us look at a syntetic example in Figure 4.5 where our analysis is able to prove the preconditions. The string value of `url` is represented as

$$\mathcal{L} \left[\mathcal{B} \left[\{c \left[\overline{\text{http://testnumericvalues.com?length=}} \right] o[\bar{v}]\} \right]^{1,1} \right]$$

where \bar{v} is a symbolic variable that has the same value as v , i.e., a number greater than or equal to 0. The analysis then probes for the web service specification by trying to invalidate the predicate

$$\text{h} \left(\text{url}, \overline{\text{url}}, (l, r) \rightarrow l \neq \overline{\text{http://testnumericvalues.com}} \right)$$

It proves the predicate false and asks the intermediate representation of the specification for the conditions on the parameters. The conditions are given as the predicate

$$\begin{aligned} &\text{h} \left(\text{url}, \overline{\text{url}}, ((l, r) \rightarrow \right. \\ &\quad \text{h} \left(r, \overline{\text{url}}, (\mathcal{A}(\&, e \rightarrow \right. \\ &\quad \quad \text{h} \left(e, \overline{\text{url}}, (k, v) \rightarrow k \neq \overline{\text{length}} \vee v \geq 0) \right)) \end{aligned}$$

The fail-condition is the negation of this predicate. The predicate is translated as

$$\begin{aligned} &\neg \text{h} \left(\text{url}, \overline{\text{url}}, ((l, r) \rightarrow \right. \\ &\quad \text{h} \left(r, \overline{\text{url}}, (\mathcal{A}(\&, e \rightarrow \right. \\ &\quad \quad \text{h} \left(e, \overline{\text{url}}, (k, v) \rightarrow k \neq \overline{\text{length}} \vee v \geq 0) \right)) = \\ &\neg \text{h} \left(\mathcal{L} \left[\mathcal{B} \left[\{c \left[\overline{\text{length=}} \right] o[\bar{v}]\} \right]^{1,1} \right], \overline{\text{url}}, (\mathcal{A}(\&, e \rightarrow \right. \\ &\quad \text{h} \left(e, \overline{\text{url}}, (k, v) \rightarrow k \neq \overline{\text{length}} \vee v \geq 0) \right)) = \\ &\neg \text{h} \left(\mathcal{L} \left[\mathcal{B} \left[\{c \left[\overline{\text{length=}} \right] o[\bar{v}]\} \right]^{1,1} \right], \overline{\text{url}}, (k, v) \rightarrow k \neq \overline{\text{length}} \vee v \geq 0 \right) = \\ &\neg (\mathcal{L} \left[\mathcal{B} \left[\{c \left[\overline{\text{length}} \right]\} \right]^{1,1} \right] \neq \overline{\text{length}} \vee \mathcal{L} \left[\mathcal{B} \left[\{o[\bar{v}]\} \right]^{1,1} \right] \geq 0) = \\ &\mathcal{L} \left[\mathcal{B} \left[\{c \left[\overline{\text{length}} \right]\} \right]^{1,1} \right] = \overline{\text{length}} \wedge \mathcal{L} \left[\mathcal{B} \left[\{o[\bar{v}]\} \right]^{1,1} \right] < 0 = \\ &\text{True} \wedge \text{False} = \text{False} \end{aligned}$$

```
1 action handconstructed()
2   var v := wall → ask_number("length?")
3   if(v < 0) then
4     v := 0
5   var url := "http://testnumericvalues.com?length=" // v
6   var r := web → download(url)
7   if(not r → is_invalid) then
8     var js := web → json(r)
9     var status := js → string("status")
```

Figure 4.5: Synthetic TouchDevelop program that communicates with the fictitious testnumericvalues API

While the usefulness of parameter checking for the sake of proving or disproving preconditions is limited, it can be a useful tool to derive the format of the response. Various web services provide an option to choose the format of the response. For example, the World Weather Online Free API, as shown in Figure 4.3, returns either a JSON object or an XML object, depending on the value of the parameter *format*. Usually, such a *format*-parameter can only take a small amount of different values.

Our analysis is also able to detect possible URL injection with malicious parameters. Thanks to the Maybe Contained Character codomain in the Abstract Strings, the analysis can detect whether the value of a parameter can possibly contain an ampersand as parameter separator, accompanied by various other, unwanted parameters.

Many genuine bugs in TouchDevelop scripts are due to not checking the response for validity. Even if the URL satisfies the constraints imposed by the specification, one must always assume the possibility of the web service not being available. For example, the script in Figure 4.5 must check whether the response is valid in line 7, even though the predicate for the fail state is false. Other subtle bugs that often occur are due to not encoding strings that are plugged into an URL string. Even if a parameter has no constraints and can take on arbitrary string values, we may not be able to prove the preconditions because the substring may itself contain other parameters. These kinds of bugs are generally very bad, because they do not appear to be a problem when a user sends expected input, but they let users manipulate the execution with malicious input.

The additional cost of the analysis is only estimated based on measured execution times. The measurements show a clear increase in execution time if the analysis is enabled. The mean slowdown is about 50% of the execution time without the analysis enabled. The cost seems to not scale over proportional in the size of the program, however, the set on which we evaluated our analysis is not large enough to make an informed guess on the complexity. The additional cost comes from maintaining a Relational Bricks analysis for

the Split expressions, from keeping the structural string annotations, and from evaluating the Split expression predicates on the domain.

Chapter 5

Related Work

The analysis we are going to propose in this work is heavily dependent on static information about string values. The process of deducing and keeping track of information about string values is called string analysis. Depending on the goal of the analysis, different information is useful. Hence, various different kind of string analyses exist. In this work, we are concerned about static analysis, that is, knowledge about the behavior of a program without actually executing it. In that regard we are mostly interested in static string analysis.

Christensen et al. [8] describe a string analysis that abstracts string values using a regular language. Their analysis has been implemented to analyze Java strings. However, the analysis has difficulties dealing with strings stored in heap variables.

Choi et al. [7] solve the problem of dealing with heap variables and integrate constant propagation by designing a string analyzer that uses abstract interpretation techniques. Abstract interpretation is a mathematical framework for static analysis in which we also fit our approach. One of their main contributions is the design of a widening operator. Crafting this widening operator is made possible by choosing to represent only a subset of regular languages.

Kim and Choe [20] have developed a string analysis that can analyze context-free languages. The analysis uses abstract interpretation techniques. The abstract domain they propose is a predicate domain on transitions of pushdown automata that read the string. However, this analysis can only check whether a string belongs to a certain grammar which only allows some syntactic checks.

Constantini et al. [10] also build their analysis in the abstract interpretation framework. They develop different abstract domains with different trade-offs in complexity and expressiveness. One of their less expressive but in return cheaper analysis approximates string values as a set of maybe contained characters. While information about maybe contained characters

does not sound like much, we actually employ this domain in our analysis. Their two most expressive abstract domains are called the Bricks domain and the String Graph domain. Both domains capture character inclusion and order. Their approach is generic and – thanks to abstract interpretation – modifiable. The Bricks domain is an approximation to regular expressions. In this work, we leverage the benefits of the Bricks analysis, i.e., the extensibility provided by its formalization in the abstract interpretation framework, the simplicity and concurrent expressiveness of the representation, and the fact, that it has already been implemented in the Sample analyzer. The Bricks analysis is representing possible values of a string through building blocks containing concrete string values. Our augmentation to the Bricks analysis allows the storage of relational information, including knowledge of non-string values, by linking it with other domains. A more detailed introduction to the Bricks analysis is provided in Section 2.6. Our proposed augmentation is described in Section 3.2.

One particular use of string analysis is the analysis of executable code. SQL queries, HTML pages, Latex documents or even Java code is statically just plain text that can be generated by a program. In our analysis, we want to formalize the constraints that the specification of a procedure that is executed on a remote machine over a network poses on the URL string. Recent work by Tripp et al. [23] is also concerned about the analysis of string values used in URLs. Their analysis addresses the question of whether a malicious agent can possibly manipulate an URL String in a JavaScript environment to redirect a seemingly harmless request to a fake web host. Our analysis also produces a warning, if it cannot guarantee the absence of such behavior, e.g., if the code does not filter user input used in parameters by encoding URL reserved characters. However, our analysis is not especially targeted towards URL safety. Furthermore, our analysis is purely static while Tripp et al. propose a hybrid approach.

Chapter 6

Conclusion

In this work we developed a static analysis for string values in the context of communication over a network between the analyzed client program and a remote software system, i.e., a web service. The analysis does not track the state of the web service, hence, the analysis can only be precise if the web service is stateless, i.e., if the state is kept at the client side. All information must be passed to the web service encoded in an URL string.

We developed an augmentation to the Bricks string analysis, a string analysis based on regular expressions that operates in the abstract interpretation framework. The augmentation relates the string representation of arbitrary-typed values to their original value by introducing symbolic variables. The relation is necessary to check non-string parameters that are plugged into the URL query in a specialized domain.

We formalized a predicate logic for URL strings. The logic compares individual key-value pairs against the constraints on the formal parameters imposed by the web service specifications. Based on the observation that the individual values are separated in the URL string by special delimiter characters, our higher-order logic predicate *Split expression* divides the string at those delimiters and applies a predicate to both substrings. We also introduced a *forall-splits quantifier*, a predicate that recursively applies Split expressions. An inherent property of the forall-splits quantifier is the loss of ordering of the string parts. The ordering of parameters in the URL string is irrelevant; the parameter list $\overline{key1=val1\&key2=val2}$ and the parameter list $\overline{key2=val2\&key1=val1}$ are equivalent. By applying the forall-splits quantifier on the parameter list of an URL string, normalization is obtained for free.

Inspired by the Web Application Description Language, we defined a language for web service specifications. The language is more expressive in terms of constraints on values of both, the parameters encoded in the URL string and the string returned by the web service. To specify the guarantees on the values of the returned strings, we defined structural string annotations. The annotations are defined as a lattice and fit in the abstract

interpretation framework.

We implemented our analysis as part of the TouchBoost static analysis tool suite. The implementation works on a control flow graph representation of TouchDevelop scripts.

We evaluated the implementation on real TouchDevelop scripts that were written by others. Often, the analysis was able to either dispose of a false alarm, or reveal a genuine bug.

6.1 Future Work

Our implementation is restricted to analyzing programs that communicate with stateless web services. In a stateless web service, the state must be kept in the client and all necessary information must be provided by each call. Hence, there is no additional protocol on the order in which different methods of the web service must be called – each call can be observed in isolation and each method can be seen as an individual web service. The contracts given by the specifications may further only depend on the parameters passed in a single request.

Analyzing programs that communicate with a stateful web service could be interesting. However, reasoning about multiple states poses new challenges. Let us briefly discuss changes imposed on the analysis and possible solutions. In summary, we can think of the following new challenges:

- The analysis must maintain the state of the web service separately from the state of the client.
- Semantics must be defined that transition the state of the web service after, a HTTP request.
- Nondeterminism such as the elapsed time between two subsequent HTTP requests may alter the state of the web service before an HTTP request.

An analysis that investigates communication between a program and a stateful web service must maintain both, the state of the client and the state of the web service. A basic approach would be to keep the state of the program and the state of the web service in a cartesian product domain. However, a program might communicate with multiple web services. Hence, either is the domain that keeps the state of the web service able to extend itself every time the analysis discovers communication with a new web service, or an analysis that collects all accessed web services before the actual state is initialized. However, the collecting analysis must already capture the state of the client for URL analysis. In practice, our proposed analysis could be applied to collect the accessed web services.

Keeping a separate state for the web service does not suffice. Additional semantics for HTTP requests must be developed based on the web service specifications. Note that statements that do not send a request to the web service must not alter the state of the web service. Nondeterminism, e.g., a timeout if too much time passes between two subsequent requests, could be modeled by over approximation. For example, we do not statically know how much time elapsed between two subsequent requests. Hence, before transitioning the state of the web service using the semantics of the request, an upper bound on possible pre-states needs to be computed.

Bibliography

- [1] URL: <http://developer.nytimes.com/>.
- [2] URL: <https://www.touchdevelop.com/>.
- [3] URL: <http://www.icndb.com/api/>.
- [4] URL: <https://developer.worldweatheronline.com/page/documentation>.
- [5] URL: <http://ifconfig.me/>.
- [6] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>, 2004.
- [7] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A practical string analyzer by the widening approach. In *APLAS*, pages 374–388, 2006.
- [8] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *SAS’03*, pages 1–18, 2003.
- [9] Agostino Cortesi. Widening operators for abstract interpretation. In *Software Engineering and Formal Methods, 2008. SEFM’08. Sixth IEEE International Conference on*, pages 31–40, 2008.
- [10] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Static analysis of string values. In *ICFEM’11*, 2011.
- [11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL’77*, pages 238–252, 1977.
- [12] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *Programming Languages and Systems*, pages 21–30. 2005.
- [13] Ádám Darvas and K Rustan M Leino. Practical reasoning about invocations and implementations of pure methods. In *Fundamental Approaches to Software Engineering*, pages 336–351. 2007.

- [14] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, pages 115–150, 2002.
- [15] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
- [16] Raphael Fuchs. Inferring counter-examples from abstract error states via backward analysis. Master’s thesis, ETH Z”urich, 2014.
- [17] Gartner. Worldwide pc shipments declined 6.9 percent in fourth quarter of 2013. <http://www.gartner.com/newsroom/id/2647517>, 2014.
- [18] Gartner. Worldwide tablet sales grew 68 percent in 2013, with android capturing 62 percent of the market. <http://www.gartner.com/newsroom/id/2674215>, 2014.
- [19] Marc Hadley. Web application description language. <http://www.w3.org/Submission/2009/SUBM-wadl-20090831/>, 2009.
- [20] Se-Won Kim and Kwang-Moo Choe. String analysis as an abstract interpretation. In *VMCAI’11*, pages 294–308, 2011.
- [21] Daniel Schweizer. Overapproximating the cost of loops. Master’s thesis, ETH Z”urich, 2012.
- [22] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, pages 49–60, 2011.
- [23] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *ISSTA’14*, 2014.