



Università
Ca' Foscari
Venezia

Ph.D. Degree
in Computer Science
Cycle XXXV

Final Thesis

A generic framework for multilanguage analysis

Design of an abstract interpretation-based static
analyzer

Candidate

Luca Negrini

Supervisor

Prof. Agostino Cortesi

Co-supervisor

Elisa Burato, Ph.D.

Thesis Referees

Prof. Antoine Miné

Prof. Peter Müller

Abstract

Modern software engineering revolves around distributed applications. From IoT networks to client-server infrastructures, the application code is increasingly being divided into separate sub-programs interacting with each other. As they are completely independent of each other, each such program is likely to be developed in a separate programming language, choosing the best fit for the task at hand.

From a static program analysis perspective, taking on a mixture of languages is challenging. Traditionally, analyzers targeted a single language, or a family of similar ones, to accurately tune the analysis to its (or theirs) features. In this context, a whole-program semantic analysis on software built through multiple languages can only be achieved by a combination of analyzers, setting up a communication scheme between the tools to consider inter-language interactions. As multiple analyses are required, that might also need to get iterated several times to exchange information, this setup does not permit static analysis to have an appreciable and meaningful impact in real-world scenarios.

This thesis defines a generic framework where modular multilanguage static analyses can be defined through the abstract interpretation theory. The framework has been implemented in LISA (**L**ibrary for **S**tatic **A**nalysis), an open-source JAVA library that provides the complete infrastructure necessary for developing static analyzers. LISA strives to be modular, ensuring that all components taking part in the analysis are both easy to develop and highly interchangeable. LISA also ensures that components are parametric to all language-specific features: semantics, execution model and memory model are not directly encoded within the components themselves. LISA analyzes *CFGs* where the set of possible nodes is not fixed: users of the library can define language-specific node instances with customized semantics, enabling different behaviors for the same construct depending on the programming language it was written with.

The framework has been instantiated to analyze smart contracts written in Go and data science notebooks written in PYTHON. As nondeterminism is known to be troublesome for blockchain ecosystems, GoLISA (an analyzer for Go based on LISA) applies information flow analyses to detect when nondeterministic constructs can affect the blockchain state. Instead, PyLISA (an analyzer for PYTHON based on LISA) provides an abstraction for software that deals with *dataframes*, constructing a graph tracking all the operations that the program performs over them, thus unifying all syntactic constructs performing the same operation. Further abstractions can be developed relying on such a graph to derive properties about the original program, such as data leakages or data provenance. A third proof-of-concept instantiation is also provided, demonstrating LISA's capability to analyze multiple languages in a single analysis through the discovery of an IoT vulnerability spanning C++ and JAVA code.

Two additional contributions are part of this thesis to provide a full analysis

ecosystem: SARL and TARSIS. SARL is a domain-specific language that can be used to compactly model how frameworks and libraries interact with the analyzed application. Through SARL, one can produce concise specification files that an analyzer can exploit to automatically annotate a program, such that analysis components can use the presence (or absence) of specific annotations to agnostically react to the usage of a framework. Instead, TARSIS abstracts strings as regular languages, exploiting finite state automata operating on an alphabet of strings. Using such an alphabet shrinks the automata, resulting in a noticeable performance improvement w.r.t. existing automata-based abstractions.

Riassunto

Le tecniche moderne per lo sviluppo software sono ampiamente influenzate dalle applicazioni distribuite. Dalle reti IoT alle infrastrutture client-server, il codice che compone l'applicazione è sempre più suddiviso in sottoprogrammi separati che interagiscono tra loro. Poiché questi ultimi sono completamente indipendenti l'uno dall'altro, ciascuno di questi programmi può essere sviluppato in un linguaggio di programmazione diverso, scegliendo la soluzione migliore per l'attività da svolgere.

Dal punto di vista dell'analisi statica dei programmi, affrontare una combinazione di linguaggi è impegnativo. Storicamente, lo sviluppo di un analizzatore ha avuto come target un singolo linguaggio, o una famiglia di linguaggi simili, in modo da raffinare l'analisi sulle sue (o loro) caratteristiche. In questa situazione, l'analisi semantica di un'intera applicazione multilinguaggio può essere eseguita solo attraverso una combinazione di analizzatori, predisponendo un canale di comunicazione tra gli strumenti in modo da considerare le interazioni tra moduli scritti in linguaggi diversi. Poiché sono necessarie più analisi, che potrebbero dover essere iterate numerose volte per permettere lo scambio di informazioni, questa configurazione non consente all'analisi statica di avere un impatto apprezzabile e significativo in scenari reali.

Questa tesi definisce un framework generico in cui è possibile definire analisi statiche multilinguaggio utilizzando la teoria dell'interpretazione astratta. Il framework è stato implementato in LiSA (**L**ibrary for **S**tatic **A**nalysis), una libreria JAVA open-source che fornisce l'infrastruttura completa necessaria per lo sviluppo di analizzatori statici. LiSA è modulare, garantendo che tutti i componenti che prendono parte all'analisi siano facili da sviluppare e facilmente intercambiabili. LiSA garantisce inoltre che i componenti siano parametrici rispetto a tutte le funzionalità specifiche del linguaggio: semantica, modello di esecuzione e modello di memoria non sono codificati direttamente all'interno dei componenti stessi. Infatti, LiSA analizza CFGs in cui l'insieme dei possibili nodi non è prefissato: gli utenti della libreria possono definire istanze di nodi specifiche per ogni linguaggio, definendo una semantica personalizzata e consentendo comportamenti diversi per lo stesso costrutto a seconda del linguaggio di programmazione in cui è stato scritto.

Il framework è stato istanziato per analizzare smart contracts scritti in Go e notebook di data science scritti in PYTHON. Poiché è noto che il non-determinismo è problematico per gli ecosistemi blockchain, GoLiSA (un analizzatore per Go basato su LiSA) applica analisi di information flow per rilevare quando i costrutti non deterministici possono influenzare lo stato della blockchain. Invece, PyLiSA (un analizzatore per PYTHON basato su LiSA) fornisce un'astrazione per il software che si occupa di *dataframe*, costruendo un grafo che traccia tutte le operazioni che il programma esegue su di essi, unificando tutti i costrutti sintattici che eseguono le stesse operazioni. Ulteriori astrazioni possono essere sviluppate basandosi su tale grafo per derivare

proprietà sul programma originale, come data leakages o analisi della provenienza dei dati. Viene fornita anche una terza istanza dimostrativa, che evidenzia la capacità di LISA di analizzare più linguaggi in un'unica analisi attraverso la scoperta di una vulnerabilità IoT che coinvolge codice C++ e JAVA.

Inoltre, questa tesi contiene due contributi aggiuntivi, in modo da fornire un ecosistema di analisi completo: SARL e TARSIS. SARL è un linguaggio che può essere utilizzato per modellare in modo compatto le modalità in cui framework e librerie interagiscono con l'applicazione analizzata. Attraverso SARL, è possibile produrre file di specifica concisi che un analizzatore può sfruttare per annotare automaticamente un programma, in modo tale che i componenti dell'analisi possano sfruttare la presenza (o l'assenza) di annotazioni per reagire in modo agnostico all'uso di un framework. TARSIS è invece un'astrazione per stringhe che sfrutta linguaggi regolari, modellati tramite automi a stati finiti che operano su un alfabeto di stringhe. L'uso di un tale alfabeto riduce le dimensioni gli automi, risultando in un notevole miglioramento delle prestazioni rispetto ad astrazioni tradizionali basate su automi con alfabeti di singoli caratteri.

Contents

Abstract	i
Riassunto	iii
Contents	v
List of Figures	ix
List of Tables	xi
I Introduction	1
1 Introduction	3
1.1 Static analysis	4
1.2 Multilanguage systems	4
1.2.1 An illustrative example	6
1.3 Libraries and frameworks	7
1.4 Analyzing strings	8
1.5 Methodology	9
1.6 Contribution and publications	12
1.7 Thesis structure	14
2 Preliminaries	15
2.1 Sets and ordered structures	15
2.1.1 Sets	15
2.1.2 Relations and functions	16
2.1.3 Partitions	17
2.1.4 Ordered structures	17
2.2 Abstract interpretation	19
2.2.1 Fixpoints	19
2.2.2 Galois connections	20
2.2.3 Fixpoint abstraction	21
2.2.4 Convergence acceleration	21
2.3 Automata and abstractions	22
2.3.1 Finite state automata notation	22
2.3.2 A finite state automata abstract domain	24
2.4 Related work	25
2.4.1 Multilanguage analysis	25
2.4.2 Modeling libraries and frameworks	26

2.4.3	String analysis	28
II	Multilanguage analysis	29
3	Towards a multilanguage analyzer: LiSA	31
3.1	Overall architecture	32
3.2	The language of the analyzer	35
3.2.1	Control flow graphs	36
3.2.2	Symbolic expressions	37
3.3	The analysis state	39
3.3.1	Lattice	41
3.3.2	Semantic Domain	42
3.3.3	Value Domain	43
3.3.4	Heap Domain	47
3.3.5	Abstract State	48
3.3.6	Analysis State	50
3.4	Interprocedural Analysis	50
3.4.1	Call Graph	52
3.5	Frontends	53
3.6	Modeling library behavior: SARL	54
3.6.1	Julia	57
3.6.2	The SARL Language	58
3.6.3	Experimental Results	65
3.7	Multilanguage analysis	71
3.8	LiSA for teaching	74
3.9	Conclusion	77
4	Smart contracts analysis	79
4.1	Related Work	82
4.2	Blockchain frameworks	83
4.3	Sources and sinks of non-determinism	84
4.3.1	Sources of non-determinism	85
4.3.2	Sinks of non-determinism	86
4.4	Flow analysis for non-determinism detection	88
4.4.1	An Overview on Information Flow	89
4.4.2	GoLiSA for non-determinism detection	91
4.4.3	Detection of Sources and Sinks in GoLiSA	93
4.5	Experimental Evaluation	95
4.5.1	Quantitative evaluation	95
4.5.2	Qualitative evaluation	97
4.5.3	Limits	98

4.6	Commercio.network: an industrial case study	98
4.6.1	Commercio.network	99
4.6.2	Detecting non-determinism on Commercio.network	99
4.7	Conclusion	101
5	Analysis of data science programs	103
5.1	Related work	105
5.2	A concrete semantics for transformations	105
5.2.1	Obtaining the semantics of Python code	107
5.3	The dataframe graph domain	108
5.3.1	Abstract semantics	112
5.4	A first application: inferring dataframes shape	118
5.5	An early experiment using PyLiSA	121
5.6	Conclusion	122
III	String analysis	123
6	String analysis	125
6.1	The IMP language	126
6.2	The TARSIS abstract domain	126
6.2.1	Abstract domain and widening	126
6.2.2	String abstract semantics of IMP	129
6.3	Experimental Results	137
6.3.1	Precision of the domains on test cases	138
6.3.2	Evaluation on realistic code samples	139
6.3.3	Efficiency	141
6.4	Conclusion	142
IV	Conclusion	143
7	Conclusion	145
7.1	Thesis summary	145
7.2	Future directions	146
	Appendices	149
A	Soundness proofs of TARSIS's semantics	151
A.1	Soundness of Concat	151
A.2	Soundness of Length	152
A.3	Soundness of Contains	152
A.4	Soundness of IndexOf	153
A.5	Soundness of Replace	155

A.6 Soundness of Substring	156
Bibliography	161

List of Figures

1.1	Excerpt of the JoyCar source code	6
1.2	Program counting the occurrences of a sub in str	9
2.1	Examples of Hasse diagrams for different posets	17
2.2	Properties of α and γ in a GC	20
2.3	Sound semantic abstraction	21
2.4	An example automaton	23
2.5	Example of widening application	24
3.1	LiSA's architecture	33
3.2	Running example for LiSA's architecture overview	34
3.3	Statements and Edges	36
3.4	The SymbolicExpression hierarchy	38
3.5	Sequence diagram <i>Analysis State's assign</i>	40
3.6	The core LiSA interfaces	41
3.7	The ValueDomain hierarchy	43
3.8	The NonRelationalDomain hierarchy	44
3.9	The DataflowDomain hierarchy	46
3.10	The HeapDomain hierarchy	47
3.11	The AbstractState hierarchy	48
3.12	The AnalysisState hierarchy	50
3.13	InterproceduralAnalysis and CallGraph	51
3.14	ASP.NET specification	56
3.15	Schema of Julia's architecture with SARL	59
3.16	Windows Forms specification	60
3.17	SARL syntax	61
3.18	Disposable objects stored in fields of classes	68
3.19	UI fields generated by Visual Studio	68
3.20	Dispose() pattern of Form classes	69
3.21	Application_Start method	70
3.22	UI fields generated by Visual Studio	71
3.23	A simple Taint Analysis implementation	73
3.24	Analysis results on a minimal example	77
4.1	Cosmos SDK code affected by CVE-2021-41135	81
4.2	Cosmos SDK architecture	83
4.3	Examples of harmless and harmful non-determinism in blockchain	84
4.4	Non-determinism related to the blockchain response	87
4.5	ABCI methods and consensus flow	88

4.6	Main store of Cosmos SDK	89
4.7	Example of explicit, implicit, and side channel flows	90
4.8	Commercio.network architecture	99
4.9	AssignMembership snippet from the <code>commerciokyc</code> module	100
4.10	BurnCCC snippet from the <code>commerciomint</code> module	100
5.1	Expressions added to the PYTHON language	106
5.2	Concrete semantics of the atomic transformations	107
5.3	Example PYTHON DS program and its instrumentation	108
5.4	Example $d^\#$ abstracting the code of Figure 5.3b	111
6.1	IMP syntax	126
6.2	Concrete semantics of IMP string expressions	127
6.3	Example of widening application	128
6.4	Example automata demonstrating <code>length</code> 's semantics	131
6.5	Example of may-replacement	133
6.6	Program samples used for domain comparison	138
6.7	Programs used for assessing domain precision	140

List of Tables

3.1	Analyzed applications	66
3.2	Difference in warnings on Windows Forms analyses	67
3.3	Warnings removed on Windows Forms applications	67
3.4	Difference in warnings on ASP.NET analyses	69
3.5	Warnings removed on ASP.NET applications	70
4.1	Potential non-deterministic behaviors related to Go	86
4.2	Main sinks for blockchain software written in Go	87
4.3	Analysis evaluation	96
4.4	Warnings triggered by the analyzers on HF	97
6.1	Values of <code>res</code> at the first assert of each program	138
6.2	Values of <code>res</code> and <code>count</code> at the first assert of the respective program	140
6.3	Execution times of the domains on each program	141

Part I

Introduction

1 Introduction

Chapter Contents

1.1	Static analysis	4
1.2	Multilanguage systems	4
1.2.1	An illustrative example	6
1.3	Libraries and frameworks	7
1.4	Analyzing strings	8
1.5	Methodology	9
1.6	Contribution and publications	12
1.7	Thesis structure	14

Software governs most aspects of everyday life. Almost every human action, regardless of it being for work or leisure, involves at least one device that is running a program. Proving these programs correct is as important as ever, as they can collect all sorts of sensitive information (for instance, contents of medical records) or govern critical processes (like driving a car). Proving the correctness and reliability of such software has also become increasingly difficult: each program might come with a set of libraries written by others, whose code is not always available or might execute in some special environments. On top of this layer of complexity, different portions of the same software ecosystem might be written in separate programming languages (e.g., in a web application, the frontend running in the user’s browser will likely be written in `JAVASCRIPT`, while the backend running on the server might have been developed in `PYTHON` or `JAVA`). The combination of these peculiarities of modern programs is troublesome for static analyzers. Users typically complain about false positives that arise from missing knowledge of the analysis w.r.t. the libraries and frameworks that they use, while often being reluctant in submitting more code to the analysis (that it will inevitably increase its resource — i.e., time and memory — requirements). Supporting multilanguage analysis not only translates to more libraries and frameworks to keep track of, but also to the analysis of code modules with different execution models, memory models, and semantics for the same statement.

This thesis aims at presenting a novel framework in which a complete application, made up of different components potentially written in separate languages, can be analyzed as a whole through a single analysis, thus achieving seamless *multilanguage analysis*. Furthermore, a novel abstract interpretation of string values is presented, with the intent of covering an area of static analysis that historically received less attention than others, but that is nonetheless important for multilanguage analysis as some inter-language interactions happen through string queries.

1.1 Static analysis

Static analysis allows one to verify properties of computer programs at compile time before they are executed. This is important for proving that programs do not behave incorrectly at runtime, leading to an exception or computing wrong results. Static analysis can also provide evidence of illicit information flows, a topic highly appreciated by companies that develop software dealing with sensitive data or that is exposed to external users' interaction. For these reasons, the initial scientific interest in static analysis is nowadays coupled with increasing industrial attention, as acknowledged by the growing number of commercial actors in the static analysis market.

Most existing commercial static analyzers are based on simple pattern matching and perform minimal semantic reasoning on the code under analysis. Pattern matching is fast and scales well with respect to software size. Although the analyzers implementing pattern matching do not guarantee soundness and are able to detect very few real bugs, usually producing a high number of false positives, their scalability is a key ingredient for their commercial success [22].

In the last decades, semantic static analyses have been successfully applied to industrial software. Here we recall a few notable examples. Historically, scientific effort focused on safety-critical embedded software, usually written in C or C++. Tools like CodeSonar¹, Polyspace², AbsInt³, and ASTRÉE [49] have been applied to programs of several hundred thousand lines of code, performing interprocedural analyses.

A stream of scientific work targeted object-oriented software developing modular analyses [48], that is, analyses that reason on each method *in isolation* (i.e., intraprocedurally) producing summaries for each of them that are in turn used when analyzing other methods. In this context, various analyzers, such as CodeContracts [91] and Spec# [20], rely on contracts [96] to modularly reason on each method.

Other tools provided global (that is, interprocedural) reasoning on object-oriented programs. For instance, SOOT [137] and WALA⁴ build a global abstract call graph and infer the heap structure, while various analyses apply different checks relying on similar call graphs (e.g., Julia [123]).

The tools reported in this section focus on a single language (or on a small set of similar ones), which allowed them to develop very specific and optimized analyses according to its (or theirs) execution and memory model.

1.2 Multilanguage systems

Software architecture has dramatically evolved in the last decades. The classical client-server architecture, which was characteristic of web applications, has recently

¹<https://www.grammotech.com/codesonar-cc>.

²<https://www.mathworks.com/products/polyspace.html>.

³<https://www.absint.com>.

⁴<http://wala.sourceforge.net>.

seen broader adoption with the advent of mobile applications. Moreover, the commercial drive to the *Software as a Service (SaaS)* [95], where vendors only distribute simple clients to customers while keeping all of the application logic remote, led to a huge increase in cloud computing solutions. In general, the success of the client-server pattern in all of its flavors consists in the clear cut between what has to be distributed to users (i.e., the *client* side, which must run on as many devices as possible), and what can be maintained on a unique coherent infrastructure (i.e., the *server* side, hidden to the final user and running in a controlled environment). This enables greater modularity, control over data, and protection against tampering of the intellectual property that the program carries. Since the two sides have very different purposes, the programming languages used to implement them are typically different. The client might run in a browser, exploiting languages such as JAVASCRIPT for portability across different systems, or directly on a device, requiring a language that must be compatible with the host system (e.g., JAVA in case of an Android application). On the other hand, the backend typically exploits languages that are most efficient for the system's purpose, as any requirement for those languages can be met on the controlled system: PYTHON, C++, JAVA, and RUST are just a few of the languages that are used in the wild.

The backend is also becoming less and less monolithic. Recent years have seen the rise of *microservices* infrastructures [33], where the atomic entity that was the server is split into smaller independent components that communicate with each other through APIs. Backend logic has also started being implemented through *serverless applications*, that is, code that runs in the cloud with (close to) no knowledge about the environment it runs into. Different components of these applications typically interact with each other through events, exploiting services offered by the cloud provider. Partitioning the server code into isolated entities also loosens the requirement of having those entities written in the same language, as different tasks might exploit different languages' peculiarities. One more possible segmentation of the backend comes with *blockchain-oriented applications*, that interact with code present on a blockchain [111]. Smart contracts are usually written with specific DSLs (e.g., SOLIDITY) dedicated to a particular blockchain in order to exploit its capabilities. Only recently a stream of blockchains adopted general purpose languages for writing smart contracts [9, 24, 86, 87] such as Go, JAVA, and JAVASCRIPT.

Besides the transformation of client-server architectures, the Internet of Things (IoT) has also risen in popularity. In an IoT system, *things* (that is, devices running embedded software) communicate with each other and with gateways, possibly accessing the internet. The IoT network can also interact with various backends or other devices (note that the backends of these networks might overlap, if not even match, the ones of client-server applications). IoT networks are becoming wildly adopted in several areas [139]: healthcare, smart homes, and manufacturing are just a few of the scenarios where they are applied. Once more, different programming languages can

```

1 class JoyCar {
2   public native int readUpDown();
3   public native void runMotor(int value);
4   public static void main(String[] args) {
5     JoyCar rc = new JoyCar();
6     //Initialization
7     ...
8     while(true){
9       rc.runMotor(rc.readUpDown());
10      //Turn based on joystick input
11      ...
12    }
13  }
14 }

```

(a) JAVA code

```

1 JNIEXPORT jint JNICALL Java_JoyCar_readUpDown(JNIEnv *env, jobject o){
2   return readAnalog(A1);
3 }
4 long map(long val, long fl, long fh, long tl, long th){
5   return (th - tl) * (val - fl) / (fh - fl) + tl;
6 }
7 void motor(int ADC){
8   int value = ADC - 130;
9   softPwmWrite(enablePin, map(abs(value), 0, 130, 0, 255));
10 }
11 JNIEXPORT void JNICALL Java_JoyCar_runMotor(JNIEnv *env, jobject o, jint val){
12   motor(val);
13 }

```

(b) Embedded C++ code

Figure 1.1: Excerpt of the JoyCar source code

(and likely will) be involved in the realization of the system: devices need to host fast and optimized programs (usually written in C or C++) due to their limited resources, a requirement that is not in place for the backend that can opt for completely different languages.

1.2.1 An illustrative example

Consider, for instance, the following minimal example. The code reported in Figure 1.1⁵ has been used to prove the usefulness of static analysis for discovering IoT vulnerabilities [63]. The code implements a system composed of a joystick and a robotic car that interact through a gateway. The JAVA fragment in Figure 1.1a, running on the gateway, initializes the whole system and then repeatedly queries the joystick for steer and throttle. The C++ fragment in Figure 1.1b instead, implementing the remaining two components, interacts with the joystick's sensors and the car's motor. The two fragments communicate through the *Java Native Interface* (JNI). Here, the authors are interested in detecting the IoT Injection attack that can happen if the sensors' outputs, that could be tampered with, can *flow* into the motor's input without being sanitized, exposing the car to attacks that could damage it. Authors resort to *Taint* analysis [133, 60] for the task, but they require more than one analyzer:

⁵Available at <https://github.com/ amitmandalnitdgp/IOTJoyCar>.

since the flow might span between the two codebases, analyses for JAVA and C++ are needed. Julia and CodeSonar were selected for the task, as they both were equipped with configurable *Taint* analysis engines able to receive a specification of sources and sinks from the user. The overall analysis presented by the authors proceeds as follows:

1. the value returned by function `readAnalog` is marked as a source of tainted information for CodeSonar;
2. the second parameter of function `softPwmWrite` is marked as a sink for tainted information for CodeSonar;
3. to detect tainted values flowing from C++ to JAVA code, the value returned by `Java_JoyCar_readUpDown` is marked as a sink for CodeSonar;
4. to detect tainted values flowing from JAVA to C++ code, the first parameter of `JoyCar.runMotor` is marked as a sink for Julia;
5. a first round of *Taint* analysis is run with both analyzers: Julia does not find any vulnerability (as no sources were present under JAVA), but CodeSonar finds a flow of tainted data going into `Java_JoyCar_readUpDown`;
6. a second round is run after marking `JoyCar.readUpDown`'s return value as a source for the JAVA analysis, and this time Julia detects a flow of tainted data going into the first parameter of `JoyCar.runMotor`;
7. a third and final round was run after marking `Java_JoyCar_runMotor`'s third parameter as a source for CodeSonar, that was now able to detect the IoT Injection vulnerability with `softPwmWrite` as the sink.

Despite the successful discovery of a vulnerability that spanned multiple languages, the limits of this approach are quite evident: since tools need to exchange information at each iteration, multiple rounds of analysis are needed to reach a fix-point over the shared information, each composed by one analysis for each tool involved. Moreover, tool communication is hard, even more if those come from different vendors, as they might not agree on how information is exported and imported. Furthermore, communicating the information might not be an easy task. In this example, the authors focused on a “binary” property: a value is either tainted or not. However, with more complex structures (e.g., Polyhedra [50]) finding the appropriate format to exchange information between analyses might not be trivial.

1.3 Libraries and frameworks

One of the principles of modern software engineering is to exploit reusable code, often in the form of third-party libraries and frameworks, avoiding re-implementation of common functionalities in favor of reusable and highly tested code already widely

used. Informally, a library consists of packaged (i.e., versioned, well-documented, and ready-to-run) code that implements standard functionalities, and that can be used by programs. Instead, software frameworks represent a wider concept:

“A software framework provides a standard way to build and deploy applications. (...) Software frameworks may include support programs, compilers, code libraries, tool sets, and application programming interfaces (APIs) which bring together all the different components to enable development of a project or system.”⁶

Frameworks have been applied to various contexts. For instance, ASP.NET⁷ allows a developer to implement and deploy a web application, while Windows Forms⁸ is designed to easily build desktop applications with modern UIs. Therefore, each framework provides a specific execution model.

Static analyzers may raise alarms on generated code or specific framework components since these usually follow non-standard patterns that are hardly detectable and might confuse the analyzers. Moreover, frameworks often offer ad-hoc execution models, that might rely on specific configuration files. Not being aware of these behaviors might lead to unsoundly ignoring relevant portions of the application code, or to considering too many methods as candidates for the reflective calls to preserve soundness.

Nowadays, each programming language has dozens of software frameworks (like Spring and Lombok in JAVA, or ASP.NET and Windows Forms in C#), each one with its execution model, with new ones keep emerging. This represents a challenge for static analyzers since customers expect the analysis to be up-to-date with modern technologies, while the effort of modeling even a single framework might not be negligible. Furthermore, to keep amplifying the range of supported frameworks, there is the need for a flexible mean to model new ones, as well as to improve the knowledge of the analyzer on already-known frameworks. Finally, new versions of the same framework might require different models. It is therefore essential to document which parts of a framework are supported and how to keep the models updated when new versions are released.

1.4 Analyzing strings

Strings play a key role in any programming language due to the many and different ways in which they are used, such as to dynamically access object properties, to hide the program code by using string-to-code statements and reflection, or to manipulate data-interchange formats such as JSON, just to name a few. Despite the

⁶https://en.wikipedia.org/wiki/Software_framework.

⁷<https://www.asp.net/>.

⁸<https://docs.microsoft.com/it-it/dotnet/framework/winforms/>.

```
1 int countMatches(String str, String sub) {
2   int count = 0;
3   int len = sub.length();
4   while (str.contains(sub)) {
5     int idx = str.indexOf(sub);
6     count = count + 1;
7     int start = idx + len;
8     int end = str.length();
9     str = str.substring(start, end);
10  }
11  return count;
12 }
```

Figure 1.2: Program counting the occurrences of a sub in `str`

great effort spent in reasoning about strings, static analysis often failed to manage programs that heavily manipulate them, mainly due to the inaccuracy of the results and the prohibitive amount of resources (time, space) required to retrieve useful information on strings. On the one hand, finite height string abstractions [41] are computable in a reasonable time, but precision is suddenly lost when using advanced (but common) string manipulations. On the other hand, more sophisticated abstractions (e.g., the ones reported in [14, 39]) compute precise results but they require huge, and sometimes unrealistic, computational resources, making realistic code intractable for them. A good representation of the latter abstractions is the finite state automata domain [14]. Over-approximating strings as regular languages using finite state automata has been shown to increase string analysis accuracy in many scenarios, but it does not scale to real-world programs dealing with statically unknown inputs and long text manipulations.

Consider the code of Figure 1.2, that counts the occurrences of string `sub` inside string `str`. This code is a simplification of the `StringUtils.countMatches` method from the Apache commons-lang library⁹, one of the most popular JAVA libraries providing extra functionalities over the core classes of the `java.lang` package (that contains class `String` as well). Proving properties about the value of `count` after the loop is particularly challenging since it requires correct modeling of a set of string operations (`length`, `contains`, `indexOf`, and `substring`) and their interaction. State-of-the-art string analyses fail to precisely model most of such operations since their abstraction of string values is not rigorous enough to deal with these situations. The loss of precision usually leads to failure in proving string-based properties (also on non-string values) in real-world software, such as the numerical bounds of the value returned by `countMatches` when applied to a given pair of strings.

1.5 Methodology

Statically analyzing software is becoming increasingly difficult. Achieving this means having an accurate model of the software's semantics: execution model, memory

⁹<https://commons.apache.org/proper/commons-lang/>.

model and the effect of each instruction provided by the programming languages used must be taken into account. On top of this, each language comes in different versions, and with a whole ecosystem of libraries and frameworks with their separate versions. This adds more constraints on the features to model for having an effective analysis. It should be intuitive that the sheer number of programming languages that exist and are being used to perform safety-critical tasks or to manipulate users' sensitive data already poses a challenge, as creating and maintaining an analyzer for each of those is impractical. If inter-language communication is also taken into account, the picture complicates even more: inside the same application, one might find portions of code with contrasting memory models, or containing similar instructions that have completely different semantics.

In this context, cooperation between different analyzers (i.e., the strategy illustrated in Section 1.2.1 to detect the IoT Injection attack) is not ideal, since the number of analyzers involved and the number of analysis rounds needed to reach a fixpoint will grow with the complexity of the program to analyze (recall that three rounds were required to analyze less than 100 lines of code). Ad-hoc solutions, that is, analyzers targeting a specific combination of programming languages, are also not viable: rewriting an analyzer from scratch each time a new combination arises leads to an excessive amount of work.

Despite providing an in-depth solution tailored only to the analysis of complete JEE applications (JAVA backend and a JSP/JSF front-end), the authors of [121] correctly identify five common challenges that have to be addressed regardless of the combination of languages that one wants to analyze:

- the analysis should rely on a standard representation that can model all programming languages;
- the analysis must detect when multi-language code is used and parse it following its language;
- the analysis must know the communication patterns available in each programming language, and it must also consider the versions of such languages;
- the different frameworks used to deploy different components must be analyzed to understand hidden dependencies, component life cycle and configuration, and callbacks invoked implicitly;
- the analysis must identify string literals and parse them following the application context.

Starting from these challenges, we lay out the five principles that a multilanguage static analyzer should, in our opinion, follow. These are not meant to be absolute laws, but rather architectural guidelines for the development of such analyzer.

Separation of syntax and semantics. No predefined semantic meaning should be attributed to syntactic constructs. This imposes a clear cut between the intermediate representation used for representing the semantics of a program and the syntactic constructs that reflect the structure of the source¹⁰ code. While we expect that most static analyzers follow this principle, it becomes mandatory for multilanguage ones: when coupling syntax with semantics, analyses will have to change whenever a new language is added to the analyzer, as its constructs must be handled appropriately. Instead, the two should be separated and the former must be dynamically rewritten into the latter.

Parametrization. Components of the analyzer taking part in the evaluation of a program's semantics at a specific program point must be parametric to the source language of the program point. This ensures that no language-specific assumptions are hardcoded inside the analyzer. Language-dependent semantics can be attached to program points at parse time. Analysis components can then resort to the attached information to properly evaluate the semantics of that program point.

Extensibility. There should not be a predefined set of syntactic constructs or semantic operations that the analyzer can support. The endgame is analyzing as many languages as possible: each of them can bring new instructions, that can have a completely new semantic meaning. Thus, (i) the intermediate representation used to model the program structure must be extensible, (ii) the set of semantic operations that the analyses can interpret must be augmentable, and (iii) all analysis components should *expect* the set of operations to be extended (i.e., that no component should raise errors when new constructs are added to the intermediate representation).

Modularity. Analysis components should be modularly defined, such that implementations of each component can be swapped with different ones without others needing modifications. This, other than leading to higher quality code (maintainable and testable), allows users to customize the analysis according to their needs: some might be willing to pay more computationally to get better results, while others might only require coarser analyses to run. This also enables composition of the analysis w.r.t. a specific language combination. Moreover, this will also simplify analysis combinations, as those can as well be defined modularly.

Ease of use. Implementation of analysis components should resemble as closely as possible their formal definition. This aims at making the implementation as clear as possible, thus lowering the time needed to learn how to extend the analyzer with new implementations. An implicit requirement of this principle is that analysis compo-

¹⁰Here, we identify as source code the language in which the program to analyze is written in. This does not prevent the analysis of compiled code, as it can be seen as a form of source code by itself.

nents should be agnostic of one another: for instance, one should not need to handle procedure calls when implementing an analysis for numerical values.

1.6 Contribution and publications

Following the five challenges and the five principles reported in Section 1.5, this thesis formalizes the structure of LiSA, a **Library for Static Analysis** aimed at achieving multilanguage analysis, and that can be used to create static analyzers by abstract interpretation [43, 45]. Roughly, LiSA provides the full infrastructure of a static analyzer: starting from an intermediate representation in the form of extensible control flow graphs (CFGs), LiSA lets users define analysis components and then takes care of orchestrating the analysis using a unique fixpoint algorithm over CFGs. Moreover, parsing logic is left to the user, that will define *frontends* translating source code into CFGs (modeling the syntax of the input program), while also providing rewriting rules for each CFG node into *symbolic expressions*, an internal extensible language representing atomic semantic operations (thus modeling the semantics of each instruction of the input program). LiSA comes with several implementations of each component, and with an example *frontend* for a simple object-oriented imperative language used for testing and demonstration. We then provide a proof-of-concept multilanguage analysis using LiSA on the example reported in Section 1.2.1.

We also show the effectiveness of LiSA in building analyzers for different languages. First, we show how the GoLiSA analyzer can tackle Go smart contracts, exploiting information-flow analyses to detect critical usages of non-deterministic constructs. We employ both *Taint* analysis and *Non-interference* to check if non-determinism can influence updates to the shared blockchain state, successfully analyzing a benchmark of almost 300 contracts pulled from GitHub. Then, we employ the PyLiSA analyzer to tackle JuPyter Notebooks (that is, PYTHON code) used in data science. We define an abstraction of such code that extrapolates the transformations applied to *dataframes*, unifying syntactic operations performed through various library functions. This abstraction can be used as a base to perform further analyses over which properties can be computed, such as inferring the shape of each dataframe or detecting data leakages.

Furthermore, we define a domain-specific language called SARL, designed to compactly specify the semantics of libraries and frameworks to generate annotations for a static analyzer. Small SARL files (in the order of tens of lines) can be passed to an engine that uses them to dynamically generate annotations whenever a given condition is satisfied on a program member, and the analyzer can then adapt its assumptions relying on these annotations. SARL has been shown to help Julia refine its results when the program under analysis was meant to be executed with C# frameworks that Julia had no previous knowledge of.

Lastly, to tackle the last challenge of Section 1.5, we formalize TARSIS, an abstract

domain for string properties. TARSIS exploits final state automata to precisely track string values that can be modeled through regular languages, while also providing good approximations for others. The key idea of TARSIS is to reduce the computational requirements typical of automata-based domains by considering alphabets of strings instead of single characters, thus compacting the resulting automata. TARSIS has been implemented into a prototypical analyzer, and early evaluations regarding precision and efficiency have been performed.

Publications. This thesis is partly based on the following published papers:

- L. Olivieri, F. Tagliaferro, V. Arceri, M. Ruaro, L. Negrini, A. Cortesi, P. Ferrara, F. Spoto, E. Tallin. “Ensuring Determinism in Blockchain Software with GoLiSA: An Industrial Experience Report”. In: *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2022)*. DOI: 10.1145/3520313.3534658
- P. Ferrara, L. Negrini, V. Arceri, A. Cortesi. “Static analysis for dummies: experiencing LiSA”. In: *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2021)*. DOI: 10.1145/3460946.3464316
- L. Negrini, V. Arceri, P. Ferrara, A. Cortesi. “Twinning Automata and Regular Expressions for String Static Analysis”. In: *Proceedings of the 22nd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2021)*. DOI: 10.1007/978-3-030-67067-2_13
- P. Ferrara, L. Negrini. “SARL: OO Framework Specification for Static Analysis”. In: *Software verification. Springer, Cham, 2020. PP. 3-20*. DOI: 10.1007/978-3-030-63618-0_1 (VSTTE 2020).

Moreover, portions of the thesis rely on papers that have yet to appear:

- Chapter 3 is based on a book chapter to appear in the “Challenges of Software Verification”¹¹;
- Chapter 4 is based on a paper under second revision for ECOOP¹².

Note that, whenever a chapter or section is based on one or more of the listed works, an explicit note will be made.

¹¹The book will contain contributions presented at the CSV workshop (<https://ssv.dais.unive.it/events/challenges-of-software-verification-workshop/>).

¹²<https://conf.researchr.org/home/ecoop-2023>.

1.7 Thesis structure

Chapter 2 introduces basic notions that will be used throughout the rest of the thesis, and discusses the related work, most notably Mopsa [83], that is the main alternative to the approach taken in this thesis.

Chapter 3 illustrates the overall architecture of LiSA. In particular, a high-level overview is first introduced in 3.1, to give the reader a broad idea of how the components discussed in later sections will fit into the complete schema. Then, the CFG structure that LiSA uses to represent generic programs is first introduced in Section 3.2, together with the internal language of *symbolic expressions*. The following sections introduce the *Analysis State* (3.3), *Interprocedural Analysis* (3.4), and *frontends* (3.5). Section 3.6 defines SARL, reporting its implementation in Julia that has been experimented with two popular C# frameworks. The chapter concludes with a demonstrative example of seamless multilanguage analysis on the IoT network of Section 1.2.1, followed by a discussion on our teaching experience with LiSA in Master Courses.

In Chapters 4 and 5, we report the usage of LiSA to develop *frontends* and analyses in different contexts. Chapter 4 defines an information-flow based analysis to detect critical usages of non-deterministic constructs in smart contracts. Experiments on a benchmark of almost 300 contracts are reported, followed by an industrial case study on the Commercio.network blockchain in Section 4.6. Instead, Chapter 5 illustrates an ongoing work targeting JuPyter Notebooks used in data science. An abstraction for the PYTHON code they contain is defined as a base for determining properties of such code. The abstraction constructs a graph reporting the transformations applied to all *dataframes* used in the notebook, abstracting away the syntactic constructs used to manipulate them. We implement our abstraction and experiment it as a base to infer the shape of the dataframes used by the program.

Chapter 6 introduces TARSIS, an abstract domain for string properties. TARSIS has been defined over a minimalistic IMP language, whose definition can be found in the same chapter, together with preliminary experimental results.

Finally, Chapter 7 concludes and discusses future work, and Appendix A reports soundness proofs for the abstract semantics of Chapter 6.

2 Preliminaries

Chapter Contents

2.1	Sets and ordered structures	15
2.1.1	Sets	15
2.1.2	Relations and functions	16
2.1.3	Partitions	17
2.1.4	Ordered structures	17
2.2	Abstract interpretation	19
2.2.1	Fixpoints	19
2.2.2	Galois connections	20
2.2.3	Fixpoint abstraction	21
2.2.4	Convergence acceleration	21
2.3	Automata and abstractions	22
2.3.1	Finite state automata notation	22
2.3.2	A finite state automata abstract domain	24
2.4	Related work	25
2.4.1	Multilanguage analysis	25
2.4.2	Modeling libraries and frameworks	26
2.4.3	String analysis	28

In this chapter, we report the mathematical notation that will be used throughout the thesis. We start by recalling notions on sets and ordered structures, followed by an overview of abstract interpretation. We then discuss finite state automata and an abstract domain based on them. We conclude the chapter by discussing the related work.

2.1 Sets and ordered structures

2.1.1 Sets

A set, denoted with a capital letter, is an unordered and possibly infinite collection of elements, denoted with lowercase letters. We state that element x is part of a set X with $x \in X$. The cardinality of a set X , denoted by $|X| \in \mathbb{N}$, is the number of elements it contains. A set defined as $\{ x \in X \mid \phi(x) \}$ is composed of all elements in X that satisfy predicate ϕ . Given two sets X and Y , we define the following:

- $X \subseteq Y \iff \forall x \in X. x \in Y$ is the set inclusion (we say that X is a subset of Y);
- $X \subset Y \iff X \subseteq Y \wedge \exists y \in Y. y \notin X$ is the strict set inclusion (we say that X is a strict subset of Y);
- $X \cup Y \triangleq \{ z \mid z \in X \vee z \in Y \}$ is the set union;

- $X \cap Y \triangleq \{ z \mid z \in X \wedge z \in Y \}$ is the set intersection;
- $X \setminus Y \triangleq \{ x \in X \mid x \notin Y \}$ is the set difference;

Moreover, \emptyset is the set containing no elements. It follows that $|\emptyset| = 0$ and that, given any set X , $\emptyset \subseteq X$, $\emptyset \cup X = X \setminus \emptyset = X$, and $\emptyset \cap X = \emptyset \setminus X = \emptyset$. $\wp(X)$ is the powerset of X , that is, the set containing all of its subsets, and X^* is the Kleene-closure of X , that is, the infinite set of all finite sequences of its elements. The Cartesian product between X and Y is $X \times Y \triangleq \{ \langle x, y \rangle \mid x \in X \wedge y \in Y \}$.

A family of sets, denoted with a calligraphic capital letter, is a set of sets. Given such a family \mathcal{X} , we extend the union and intersection operators to such families as $\bigcup \mathcal{X} \triangleq \bigcup_{X \in \mathcal{X}} X \triangleq \{ x \mid \exists X \in \mathcal{X} . x \in X \}$ and $\bigcap \mathcal{X} \triangleq \bigcap_{X \in \mathcal{X}} X \triangleq \{ x \mid \forall X \in \mathcal{X} . x \in X \}$, respectively.

2.1.2 Relations and functions

A relation over sets X and Y is defined as $R \subseteq X \times Y$. If $X = Y$, then R is a relation over X , denoted R_X . $x R y$ and $\langle x, y \rangle \in R$ are equivalent notations to state the membership of a pair of elements to a relation. R_X can have various properties; we report here the ones of interest for this thesis:

- reflexivity: $\forall x \in X . x R x$;
- symmetry: $\forall x, y \in X . x R y \implies y R x$;
- antisymmetry: $\forall x, y \in X . x R y \wedge y R x \implies x = y$;
- transitivity: $\forall x, y, z \in X . x R y \wedge y R z \implies x R z$.

A relation R_X is a partial order if it is reflexive, antisymmetric, and transitive. It is an equivalence relation if it is reflexive, symmetric, and transitive. Given a relation $R \subseteq X \times Y$, we denote as its domain $dom(R) \subseteq X$ the set $\{ x \in X \mid \exists y \in Y . \langle x, y \rangle \in R \}$, and as its co-domain $codom(R) \subseteq Y$ the set $\{ y \in Y \mid \exists x \in X . \langle x, y \rangle \in R \}$. Given $x \in dom(R)$, $R(x) = \{ y \in codom(R) \mid \langle x, y \rangle \in R \}$ is the image of x . The image notation can be used to define a relation as the application of an expression ρ that depends on an element of its domain: if $R(x) = \{ y \in codom(R) \mid y \in \rho(x) \}$, then $R = \{ \langle x, y \rangle \mid x \in dom(R) \wedge y \in codom(R) \wedge y \in R(x) \}$.

A function (also called map) $f : X \rightarrow Y$ is a relation $f \subseteq X \times Y$ such that $\forall x \in dom(f) . \langle x, y \rangle \in f \wedge \langle x, z \rangle \in f \implies y = z$. It follows that $\forall x \in dom(f) . |f(x)| = 1$: abusing notation, we write $f(x) = \rho(x)$. We denote with $f[x \mapsto y]$ a copy of f where $f(x)$ is set to y . If $f : X \rightarrow Y$ and $g : Y \rightarrow Z$, $g \circ f : X \rightarrow Z$ is the composition of g after f , defined as $g \circ f(x) = g(f(x))$. We define the iterates f^n , $n \in \mathbb{N}$ of a function $f : X \rightarrow X$ as $f^0(x) = x$, $f^{n+1}(x) = f^n \circ f(x)$. Visually, such iterates can behave in one of three possible ways:

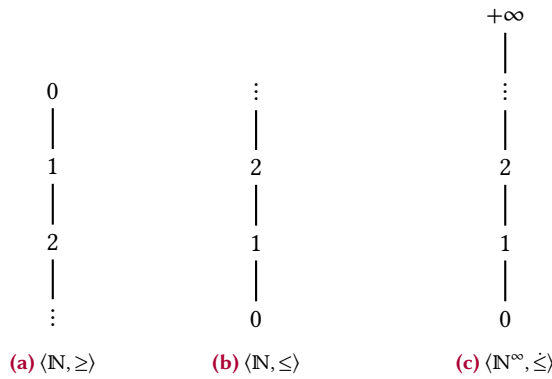
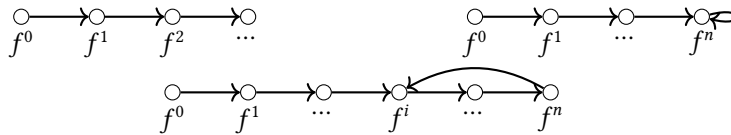


Figure 2.1: Examples of Hasse diagrams for different posets



2.1.3 Partitions

A partition of a set X , denoted as $\mathcal{P} \subseteq \wp(X)$, is a family of subsets of X such that $\forall P \in \mathcal{P}. P \neq \emptyset, \bigcup \mathcal{P} = X$, and $\forall P_1, P_2 \in \mathcal{P}. P_1 \cap P_2 = \emptyset$. An equivalence relation R over a set X induces a partition \mathcal{P}_R on X by grouping elements that are equivalent to each other into subsets. Each element of \mathcal{P}_R is called an equivalence class, denoted as $[x]_R \triangleq \{ y \in \text{dom}(R) \mid x R y \}$, and x is called the representative of the class.

2.1.4 Ordered structures

A set X with a partial ordering relation \sqsubseteq_X is called poset, denoted by $\langle X, \sqsubseteq_X \rangle$. A poset can be represented as a directed graph, where an edge's direction represents the imposed ordering (e.g., $x \rightarrow y \iff x \sqsubseteq_X y$). One can also encode such ordering in the topological structure of the graph. For instance, a *Hasse diagram* (the de-facto standard graphical representation of posets), hides self-edges encoding reflexivity ($x \rightarrow x$), represents direct ordering using undirected edges and exploiting vertical coordinates ($x \rightarrow y \implies x \sqsubseteq_X y$ if y is higher, $y \sqsubseteq_X x$ otherwise), and removes edges that are transitively implied by direct relationships.

Example 2.1.1. Examples of Hasse diagrams are reported in Figure 2.1, that depicts the posets $\langle \mathbb{N}, \geq \rangle$, $\langle \mathbb{N}, \leq \rangle$, and $\langle \mathbb{N}^\infty, \leq \rangle$, respectively (where $\mathbb{N}^\infty = \mathbb{N} \cup \{+\infty\}$ and \leq is the regular \leq extended such that $\forall n \in \mathbb{N}. n \leq +\infty$).

Given $Y \subseteq X$, $u \in X$ is an upper bound of Y if $\forall y \in Y. y \sqsubseteq_X u$. If $u \in Y$, then u is called maximal. We denote as $ub(Y)$ the set of all upper bounds of Y . The least upper bound (lub) of Y is the element $\bigsqcup_X Y$ such that $\forall u \in ub(Y). \bigsqcup_X Y \sqsubseteq_X u$, if it exists. If

$\bigsqcup_X Y \in Y$, then it is the maximum (or top, denoted as \top) element. By duality we can define the lower bound, minimal, greatest lower bound (or glb, denoted as $\bigsqcap_X Y$), and minimum (or bottom, denoted as \perp). Note that, when they exist, \top and \perp are unique thanks to antisymmetry. Given $x, y \in X$, $x \sqcup_X y \triangleq \bigsqcup_X \{x, y\}$ and $x \sqcap_X y \triangleq \bigsqcap_X \{x, y\}$. A complete partial order (cpo), denoted as $\langle X, \sqsubseteq_X, \perp_X, \sqcup_X \rangle$, is a poset $\langle X, \sqsubseteq_X \rangle$ where $\perp_X \in X$ and $\forall Y \subseteq X. Y \neq \emptyset \wedge |Y| \in \mathbb{N} \implies \bigsqcup_X Y \in X$.

Given a poset $\langle X, \sqsubseteq_X \rangle$, a subset $C \subseteq X$ is called chain if $\forall x, y \in C. x \sqsubseteq_X y \vee y \sqsubseteq_X x$. A chain can be ordered: if $C = \{x_k \mid k \in \mathbb{N}^\infty \wedge x_k \in X\}$, then $\forall i, j \in \mathbb{N}, i, j \leq |C|$ the chain is ascending, denoted C^\uparrow , if $i \leq j \implies x_i \sqsubseteq_X x_j$ or descending, denoted C^\downarrow , if $i \leq j \implies x_j \sqsubseteq_X x_i$. A poset satisfies the ascending chain condition (ACC) if every infinite ascending chain C^\uparrow is not strictly increasing, that is if $\forall C^\uparrow \subseteq X \exists i \in \mathbb{N}. \forall j > i. x_j = x_i, x_i, x_j \in C^\uparrow$. Conversely, it satisfies the descending chain condition (DCC) if every infinite descending chain is not strictly decreasing.

A join semi-lattice $\langle X, \sqsubseteq_X, \sqcup_X \rangle$ is a poset $\langle X, \sqsubseteq_X \rangle$ such that $\forall x, y \in X. x \sqcup_X y \in X$. Conversely, a meet semi-lattice $\langle X, \sqsubseteq_X, \sqcap_X \rangle$ is a poset $\langle X, \sqsubseteq_X \rangle$ such that $\forall x, y \in X. x \sqcap_X y \in X$. A lattice $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X \rangle$ is both a join and meet semi-lattice. Moreover, it is complete if $\forall Y \subseteq X. \bigsqcup_X Y \in X \wedge \bigsqcap_X Y \in X$, denoted by $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X, \top_X, \perp_X \rangle$. An alternative (but equivalent) requirement for a poset to be complete is the existence of top and bottom elements. A complete lattice can always be derived from a set X by considering its powerset: $\langle \wp(X), \subseteq, \cup, \cap, \emptyset, X \rangle$ is complete since \subseteq is a partial ordering relation, \cup and \cap are closed w.r.t. $\wp(X)$, and $\forall Y \in \wp(X). \emptyset \subseteq Y \subseteq X$. One can derive several complete lattices starting from a given one. For instance, given $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X, \top_X, \perp_X \rangle$ and a set Y , the *functional lift* [45] of X w.r.t. Y is the complete lattice $\langle Y \rightarrow X, \sqsubseteq, \sqcup, \sqcap, \perp, \top \rangle$ of total functions $Y \rightarrow X$, that is, of functions defined on all elements of Y . Lattice operators are defined as point-wise applications of operators over X on all $y \in Y$. Furthermore, given a finite set of complete lattices $\langle Y_i, \sqsubseteq_{Y_i}, \sqcup_{Y_i}, \sqcap_{Y_i}, \perp_{Y_i}, \top_{Y_i} \rangle, i \in \Delta \subset \mathbb{N}$, their Cartesian (or direct) product (Chapter 36 of [42]) is the complete lattice $\langle \prod_{i \in \Delta} Y_i, \sqsubseteq, \sqcup, \sqcap, \perp, \top \rangle$, where lattice operators are component-wise applications of the operators over each Y_i .

Example 2.1.2. Posets in Figure 2.1a and 2.1b are both lattices (intuitively, given any pair of elements, both lub and glb can be uniquely determined in \mathbb{N} by using the ordering relation on the elements themselves), but are not complete: given an infinite subset of \mathbb{N} , no given element exists in \mathbb{N} that is a lower bound (or upper bound) of the whole set. Instead, the poset $\langle \mathbb{N}^\infty, \leq \rangle$ in Figure 2.1c is a complete lattice since any infinite subset will have, in the worst case, 0 as glb and $+\infty$ as lub.

Given $\langle X, \sqsubseteq_X \rangle$ and $\langle Y, \sqsubseteq_Y \rangle$, $f : X \rightarrow Y$ is:

- monotone, if $\forall x_1, x_2 \in X. x_1 \sqsubseteq_X x_2 \implies f(x_1) \sqsubseteq_Y f(x_2)$;
- Scott-continuous, if $\forall C^\uparrow \subseteq X. \bigsqcup_X \{x \mid x \in C^\uparrow\} \in X, f(\bigsqcup_X \{x \mid x \in C^\uparrow\}) = \bigsqcup_Y \{f(x) \mid x \in C^\uparrow\}$;

- Scott-co-continuous, if $\forall C^\perp \subseteq X. \bigsqcup_X \{x \mid x \in C^\perp\} \in X, f(\bigsqcup_X \{x \mid x \in C^\perp\}) = \bigsqcup_Y \{f(x) \mid x \in C^\perp\}$;
- co-additive, if $\forall Z \subseteq X. f(\bigsqcup_X Z) = \bigsqcup_Y f(Z)$;

Instead, if $f : X \rightarrow X$, f is:

- extensive, if $\forall x \in X. x \sqsubseteq_X f(x)$;
- reductive, if $\forall x \in X. f(x) \sqsubseteq_X x$.

2.2 Abstract interpretation

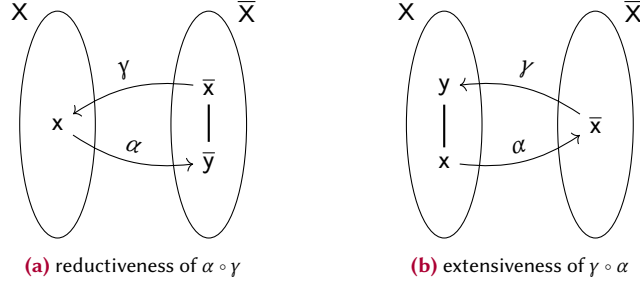
Abstract interpretation [43, 45] is a mathematical framework for reasoning about programs' semantics. Proving non-trivial properties of such semantics is, in general, undecidable [115]. Abstract interpretation overcomes this by reasoning on a sound over-approximation of the uncomputable real semantics, referred to as *concrete*, transforming it into so-called *abstract* semantics that is instead computable. While approximation restores computability, it does come with the cost of imprecision, as more executions are considered w.r.t. the actual ones exhibited by the program. However, thanks to the over-approximation, properties proven to hold for the abstract semantics are guaranteed to hold also for the concrete one. The main idea behind abstract interpretation is to define the concrete semantics as the fixpoint of a monotone function. Such a function can then be abstracted to a simpler one, gaining computability, that has to be proven sound. A detailed and accurate description of the theoretical bases of abstract interpretation can be found in [42].

In this section, we depict Hasse diagrams using ellipses to avoid binding our examples to particular poset structures. Inside such ellipses, only relevant elements of the posets and their relations are drawn (an example is visible in Figure 2.2).

2.2.1 Fixpoints

Given $\langle X, \sqsubseteq_X \rangle$ and $f : X \rightarrow X$, an element $x \in X$ is a fixpoint of f if $f(x) = x$, it is a pre-fixpoint of f if $f(x) \sqsubseteq_X x$, and it is a post-fixpoint of f if $x \sqsubseteq_X f(x)$. $FP_f, FP_f^{\leq}, FP_f^{\geq}$ are the sets of fixpoints, pre-fixpoints, and post-fixpoints of f , respectively. An element $x \in FP_f$ is the greatest fixpoint (gfp_f) if $\forall y \in FP_f. y \sqsubseteq_X x$. Conversely, it is the least fixpoint (lfp_f) if $\forall y \in FP_f. x \sqsubseteq_X y$. Several theorems prove the existence of gfp and lfp . Here, we report Kleene iterative fixpoint theorem [44] as it provides a constructive method for calculating them: given $\langle X, \sqsubseteq_X, \perp_X, \sqcup_X \rangle$ and $f : X \rightarrow X$, if f is Scott-continuous then lfp_f exists and it is the lub of the increasing chain $\perp_X \sqsubseteq_X f(\perp_X) \sqsubseteq_X f^2(\perp_X) \sqsubseteq_X \dots$, that is $lfp_f = \bigsqcup_X \{f^n(\perp_X) \mid n \in \mathbb{N}\}$. By duality, $gfp_f = \bigsqcap_X \{f^n(\top_X) \mid n \in \mathbb{N}\}$. Visually:



Figure 2.2: Properties of α and γ in a GC

2.2.2 Galois connections

Accordingly to [43, 45], a Galois connection (GC) $\langle X, \sqsubseteq_X \rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{X}, \sqsubseteq_{\bar{X}} \rangle$ between two posets $\langle X, \sqsubseteq_X \rangle$ and $\langle \bar{X}, \sqsubseteq_{\bar{X}} \rangle$ is formed by a pair of functions $\alpha : X \rightarrow \bar{X}$ and $\gamma : \bar{X} \rightarrow X$ such that $\forall x \in X, \bar{x} \in \bar{X}. \alpha(x) \sqsubseteq_{\bar{X}} \bar{x} \implies x \sqsubseteq_X \gamma(\bar{x})$. In the context of abstract interpretation, $\langle X, \sqsubseteq_X \rangle$ is the concrete poset (or domain), $\langle \bar{X}, \sqsubseteq_{\bar{X}} \rangle$ is the abstract poset (or domain), α is the abstraction function and γ is the concretization function. Whenever it is not clear from context if an object pertains to the concrete or abstract world, abstract objects will be over-lined. Given two elements x, y of the same poset, if $x \sqsubseteq y$ we say that x is more precise than y . GCs have several useful properties, like the existence of the best abstraction $\alpha(x)$ for all concrete objects x , composability, and the ability to determine α from γ and vice versa (Proposition 7 of [46]). Specifically, if $\langle X, \sqsubseteq_X \rangle$ and $\langle \bar{X}, \sqsubseteq_{\bar{X}} \rangle$ are posets, and $\alpha : X \rightarrow \bar{X}$ is a complete join-preserving function (that is, if and only if $\forall X' \subseteq X, \alpha(\bigsqcup_X X') = \bigsqcup_{\bar{X}} \alpha(X')$ whenever $\bigsqcup_X X'$ exists), then $\langle X, \sqsubseteq_X \rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{X}, \sqsubseteq_{\bar{X}} \rangle$ exists with $\gamma(\bar{y}) = \bigsqcup_X \{ x \mid \alpha(x) \sqsubseteq_{\bar{X}} \bar{y} \}$.

An equivalent characterization of GCs exploits properties of α and γ : $\langle X, \sqsubseteq_X \rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{X}, \sqsubseteq_{\bar{X}} \rangle$ holds if and only if α and γ are both monotone, $\alpha \circ \gamma$ is reductive, and $\gamma \circ \alpha$ is extensive. Figures 2.2a and 2.2b show reductiveness and extensiveness in action. Being reductive, $\alpha \circ \gamma$ ensures that concretizing an element $\bar{x} \in \bar{X}$ and then abstracting back does not lose precision. Conversely, the extensiveness of $\gamma \circ \alpha$ ensures that abstracting an element $x \in X$ and then concretizing back does not gain precision.

Soundness. While a GC is not strictly required within the abstract interpretation framework [47], its existence ensures soundness and eases soundness proofs for the semantics abstraction: given $\langle X, \sqsubseteq_X \rangle \xleftrightarrow[\alpha]{\gamma} \langle \bar{X}, \sqsubseteq_{\bar{X}} \rangle$, $f : X \rightarrow X$ and $\bar{f} : \bar{X} \rightarrow \bar{X}$, \bar{f} is a sound approximation of f in \bar{X} if $\forall x \in X. \alpha(f(x)) \sqsubseteq_{\bar{X}} \bar{f}(\alpha(x))$ (or equivalently $\forall \bar{x} \in \bar{X}. f(\gamma(\bar{x})) \sqsubseteq_X \gamma(\bar{f}(\bar{x}))$). Equivalent conditions can be established when a GC is not present: in fact, it is sufficient to prove one of the following:

- $\forall x. f(x) \sqsubseteq_X \gamma \circ \bar{f} \circ \alpha(x)$;
- $\forall \bar{x}. \alpha \circ f \circ \gamma(\bar{x}) \sqsubseteq_{\bar{X}} \bar{f}(\bar{x})$;

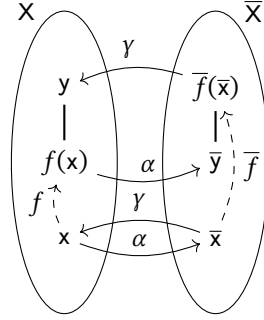


Figure 2.3: Sound semantic abstraction

- $\forall x. \alpha \circ f(x) \sqsubseteq_{\bar{X}} \bar{f} \circ \alpha(x)$;
- $\forall \bar{x}. f \circ \gamma(\bar{x}) \sqsubseteq_X \gamma \circ \bar{f}(\bar{x})$.

The combination of these conditions results in the scheme depicted in Figure 2.3, that intuitively shows that abstract computations over-approximate concrete ones.

2.2.3 Fixpoint abstraction

As stated at the beginning of this section, abstract interpretation is about defining the concrete semantics of a program as a fixpoint computation over a monotone function, and then abstracting it to regain computability. This result is the abstract semantics of the program, and it has to be proven a sound over-approximation of the concrete one. Proving the soundness of the abstract semantics can be achieved in a number of ways [47], depending on the initial hypotheses on the concrete and abstract domains, and on the concrete and abstract semantics. Here, we report Tarski's fixpoint transfer theorem, that requires both domains to be complete lattices: given two domains $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X, \top_X, \perp_X \rangle$ and $\langle \bar{X}, \sqsubseteq_{\bar{X}}, \sqcup_{\bar{X}}, \sqcap_{\bar{X}}, \top_{\bar{X}}, \perp_{\bar{X}} \rangle$, a pair of functions $f : X \rightarrow X$ and $\bar{f} : \bar{X} \rightarrow \bar{X}$, and an abstraction $\alpha : X \rightarrow \bar{X}$, if (i) f and \bar{f} are monotone, (ii) α is a co-additive, and (iii) $\alpha \circ f \sqsubseteq_X \bar{f} \circ \alpha$, then $\forall x \in \text{FP}_f^{\geq} \exists \bar{x} \in \text{FP}_{\bar{f}}^{\geq}. \alpha(x) = \bar{x} \implies \alpha(\text{lfp}_f) = \text{lfp}_{\bar{f}}$.

2.2.4 Convergence acceleration

If the abstract domain has finite height or satisfies ACC, termination of fixpoint computations over it is guaranteed by the finiteness of ascending chains: upward iterations using lub traversing the whole chain will converge in a finite number of steps, corresponding to the chain's length. When this is not the case, a tool to enforce convergence is required. Such a tool, that must over-approximate the lub to preserve soundness, is called widening: given $\langle X, \sqsubseteq_X \rangle$, a widening operator $\nabla_X : X \rightarrow X$ is an upper bound operator that, given an ascending chain $C^\dagger = \{x_0, x_1, x_2, \dots\}$, can be used to construct a new chain $C_V^\dagger = \{w_k \mid w_i = w_{i-1} \nabla_X x_i, w_0 = x_0\}$ that is ultimately stationary, i.e., $\exists i \in \mathbb{N}. \forall j > i. w_j = w_i$. Such an operator can be used to define the

sequence of iterations of a function $f : X \rightarrow X$, with $\perp_X \in X$, as:

$$x_0 \triangleq \perp_X$$

$$x_{n+1} \triangleq \begin{cases} x_n & \text{if } f(x_n) \sqsubseteq_X x_n \\ x_n \nabla_X f(x_n) & \text{otherwise} \end{cases}$$

Such sequence is ultimately stationary on element x^∇ , that is a post-fixpoint of f (i.e., a sound over-approximation of lfp_f). Note that the widening operator can be nonetheless used when not explicitly needed to accelerate convergence on long, but still finite, ascending chains.

2.3 Automata and abstractions

We start this section by recalling notions and notations about finite state automata that we will use in Chapter 6, followed by an abstract of the finite state automata string abstraction [14] that we base our work upon.

2.3.1 Finite state automata notation

In the context of automata, a set Σ of symbols is called an alphabet. Here, we avoid defining what a symbol is since it can take different forms depending on the context. Σ^* is the Kleene-closure of Σ , that is, the set of finite strings (i.e., sequences) that can be built by concatenating symbols of Σ . An element $\sigma \in \Sigma^*$ is called a string and it is composed of the symbols $\sigma_0 \dots \sigma_n \in \Sigma$, with σ_i being the i -th element of σ . We define $|\sigma| = |\sigma_0| + \dots + |\sigma_n|$ to be the length of a string (that is, the sum of the lengths of its symbols), and $\sigma[x/y]$ to be the string obtained replacing all occurrences of the symbol x in σ with y . If $\sigma = \sigma_0 \dots \sigma_n, \sigma' = \sigma'_0 \dots \sigma'_m \in \Sigma^*$, σ' is a substring of σ , written $\sigma' \curvearrowright_s \sigma$, if $\exists i, j. 0 \leq i \leq j \leq |\sigma|. \sigma = \sigma_0 \dots \sigma_i \sigma'_0 \dots \sigma'_m \sigma_{j+1} \dots \sigma_n$. Moreover, it is a prefix of σ' , written $\sigma' \curvearrowright_s^< \sigma$, if $\exists i. 0 \leq i \leq |\sigma|. \sigma = \sigma'_0 \dots \sigma'_m \sigma_i \dots \sigma_n$, or a suffix, written $\sigma' \curvearrowright_s^> \sigma$, if $\exists i. 0 \leq i \leq |\sigma|. \sigma = \sigma_0 \dots \sigma_i \sigma'_0 \dots \sigma'_m$. Furthermore, $\sigma \cdot \sigma'$ (or equivalently $\sigma\sigma'$) is the end-to-start concatenation of σ and σ' , that is $\sigma_0 \dots \sigma_n \sigma'_0 \dots \sigma'_m$. We define ϵ to be the empty string, with $|\epsilon| = 0$. It follows that, for all $\sigma \in \Sigma^*$, $\epsilon \curvearrowright_s \sigma$, $\epsilon \curvearrowright_s^< \sigma$, $\epsilon \curvearrowright_s^> \sigma$, and $\epsilon \cdot \sigma = \sigma \cdot \epsilon = \sigma$. The n -times repetition of the string σ is denoted by $\sigma^n, n \geq 0$. Note that $|\Sigma^*| = +\infty$.

A language \mathcal{L} is a set of strings over an alphabet Σ . It follows that $\mathcal{L} \subseteq \Sigma^*$. If $\mathcal{L} = \emptyset$, \mathcal{L} is called the empty language. Since languages are effectively sets, all standard set operators apply also to languages. We extend these operators by writing, abusing notation, $\mathcal{L} \cdot \mathcal{L}' \triangleq \{ \sigma \cdot \sigma' \mid \sigma \in \mathcal{L} \wedge \sigma' \in \mathcal{L}' \}$. One important operation over languages that will be used in this thesis is the factorization: given $\mathcal{L} \in \Sigma^*$, the factors $\text{FA}(\mathcal{L})$ of \mathcal{L} is the set of all substrings of \mathcal{L} , that is, $\text{FA}(\mathcal{L}) \triangleq \{ \sigma \in \Sigma^* \mid \exists \sigma', \sigma'' \in \Sigma^*. \sigma' \sigma \sigma'' \in \mathcal{L} \}$.

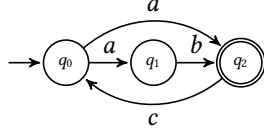


Figure 2.4: An example automaton

A finite state automaton (FA) is a tuple $A = \langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ is a finite alphabet of symbols, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is the set of final states. Intuitively, an automaton is a weighted directed rooted graph, where Q is the set of nodes, δ is the set of edges having a symbol in Σ as weight, and q_0 is the root. An example of such a graph is reported in Figure 2.4, where the automaton is built over a set of single characters, the initial state is depicted by an incoming arrow with no source, and the only final state has a double border. We denote by Fa the set of all possible FAs. Given δ , its transitive closure $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ fully determines the behavior of the automata:

$$\hat{\delta}(q, \sigma) \triangleq \begin{cases} q & \text{if } \sigma = \epsilon \\ \delta(\hat{\delta}(q, \sigma_0 \dots \sigma_{n-1}), \sigma_n) & \text{otherwise} \end{cases}$$

A path $\pi \in \delta^*$ is a sequence of transitions connecting the initial state q_0 to a final state $q_n \in F$. We denote as $\text{paths}(A) \in \wp(\delta^*)$ the set of all possible paths. When A is cycle-free, the set $\text{paths}(A)$ is finite and computable. We denote with $\text{cyclic}(A)$ the predicate that holds if and only if the given automaton contains a cycle. Given $\pi = t_0 \dots t_n \in \text{paths}(A)$, σ_{π_i} is the symbol read by transition t_i , $i \in [0, n]$, and $\sigma_\pi = \sigma_{\pi_0} \dots \sigma_{\pi_n}$ is the string recognized by such path. $|\pi| = |\sigma_\pi|$ is the length of the path. Furthermore, $|\text{minPath}(A)| \in \mathbb{N}$ denotes the (unique) length of a minimum path. If $\neg \text{cyclic}(A)$, $|\text{maxPath}(A)| \in \mathbb{N}$ denotes the (unique) length of a maximum path. Note that the set of strings recognized by all the paths of an automaton A is the *regular* language $\mathcal{L}(A)$. For instance, the automaton reported in Figure 2.4 recognizes the language $\{a, ab, aca, abca, \dots\}$. From the Myhill-Nerode theorem [104], for each regular language there uniquely exists a minimum FA (w.r.t. the number of states) recognizing the language. Given a regular language \mathcal{L} , $\text{Min}(A)$ is the minimum FA A such that $\mathcal{L} = \mathcal{L}(A)$. Abusing notation, given a regular language \mathcal{L} , $\text{Min}(\mathcal{L})$ is the minimal FA recognizing \mathcal{L} .

If $\delta : Q \times \Sigma \rightarrow Q$ is a function then A is called deterministic finite state automaton. It can be shown that deterministic and non-deterministic final state automata are equivalent w.r.t. the languages that they recognize. Moreover, since an algorithm for determinization of non-deterministic automata [113] is well known, we will only refer to deterministic ones.

Throughout the thesis, it could be more convenient to refer to a finite state automaton by its regular expression (regex for short), being equivalent. A regular ex-



Figure 2.5: Example of widening application

pression \mathbf{r} identifies a regular language $\mathcal{L}(\mathbf{r})$. Given an alphabet Σ , \emptyset represents the empty language \emptyset , ϵ denotes the language $\{\epsilon\}$, and $\sigma \in \Sigma$ is the singleton language $\{\sigma\}$. Moreover, given two regexes \mathbf{r}_1 and \mathbf{r}_2 , $\mathbf{r}_1 \parallel \mathbf{r}_2$ is the disjunction between \mathbf{r}_1 and \mathbf{r}_2 ($\mathcal{L}(\mathbf{r}_1) \cup \mathcal{L}(\mathbf{r}_2)$), $\mathbf{r}_1 \mathbf{r}_2$ is the concatenation of \mathbf{r}_1 with \mathbf{r}_2 ($\mathcal{L}(\mathbf{r}_1) \cdot \mathcal{L}(\mathbf{r}_2)$), and $(\mathbf{r}_1)^*$ is the Kleene-closure of \mathbf{r}_1 ($\mathcal{L}(\mathbf{r}_1)^*$). For the sake of clarity, parentheses might be added to group regular expressions whenever there is ambiguity in the order of operations. A regex equivalent to the automaton of Figure 2.4 is $(a \parallel ab)(c(a \parallel ab))^*$.

2.3.2 A finite state automata abstract domain

The FA-based abstract domain presented in [14] over-approximates string properties as the minimum deterministic finite state automaton recognizing them. Given an alphabet Σ of characters (that is, all symbols have length one), the finite state automata domain is defined as $\langle \text{Fa}_{/\equiv}, \sqsubseteq_{\text{Fa}}, \sqcup_{\text{Fa}}, \sqcap_{\text{Fa}}, \text{Min}(\emptyset), \text{Min}(\Sigma^*) \rangle$. The set $\text{Fa}_{/\equiv}$ of elements of the lattice is the quotient set (i.e., partition) of Fa w.r.t. the equivalence relation induced by language equality: here, automata recognizing the same language \mathcal{L} are clustered together. We abuse notation by representing such classes by one of its automata (usually the minimum): when we write $A \in \text{Fa}_{/\equiv}$ we mean $[A]_{\equiv}$. \sqsubseteq_{Fa} is the partial order induced by language inclusion, \sqcup_{Fa} and \sqcap_{Fa} are the lub and the glb, respectively, implemented as the automata union and intersection. The bottom element is $\text{Min}(\emptyset)$, that is, the minimum automaton recognizing the empty language, and the top element is $\text{Min}(\Sigma^*)$, that is, the minimum automaton recognizing any possible string over Σ . $\text{Fa}_{/\equiv}$ does not satisfy ACC since it contains infinite ascending chains. For this reason, it is equipped with the parametric widening ∇_{Fa}^n . Intuitively, such operator is defined in terms of a state equivalence relation, that merges states recognizing the same language up to a fixed length $n \in \mathbb{N}$ (a parameter used for tuning the widening precision [21, 57]) into a single one.

Example 2.3.1. Let us consider the automata $A, A' \in \text{Fa}_{/\equiv}$ in Figure 2.5a and 2.5b respectively, recognizing languages $\mathcal{L} = \{\epsilon, a\}$ and $\mathcal{L}' = \{\epsilon, a, aa\}$. The widening application starts by first applying the lub, obtaining exactly A' (Figure 2.5c). The final result is obtained by applying ∇_{Fa}^1 to A' , resulting in $A \nabla_{\text{Fa}}^1 A' = A''$ such that $\mathcal{L}(A'') = \{a^n \mid n \in \mathbb{N}\}$ (Figure 2.5d), where all the states have been merged together since they all recognize the same 1-language $(\{\epsilon, a\})$.

2.4 Related work

As the field grew and matured over decades, a vast literature about static analysis and abstract interpretation is available (most notable results are referenced in [42]), reporting a wide spectrum of techniques and their applications to prove software correct. Here, we focus on work strictly related to the core contributions of this thesis: multilanguage analysis, library modeling, and string analysis. Work that is related to LISA’s instantiations for Go and PYTHON is instead discussed in Chapter 4 and Chapter 5, respectively.

2.4.1 Multilanguage analysis

The initial focus on multilanguage analysis targeted combinations of similar languages. Julia [123] analyzes JAVA bytecode and has been extended to also analyze CIL bytecode resulting from the compilation of C# code by means of a semantic translation into JAVA bytecode [62]. Infer [56] analyzes JAVA, C, C++, and OBJECTIVE-C programs by statically translating them into a proprietary intermediate representation, called SIL, composed of only four instructions. However, these approaches are intrinsically limited by the expressiveness of the intermediate representation (JAVA bytecode and SIL in the case of Julia and Infer): since the set of constructs in those representations is predefined, they might not be enough to represent features of some languages. For instance, JAVA bytecode cannot express pointer arithmetic, while SIL is not suited for dynamic typing.

Another stream of work instead considers a scenario where one central portion of the application, written in a single programming language, interacts with *native code*, that in this context can be considered as a collection of functions written in different programming languages. [142] performs a summary-based analysis of Android applications as a whole (that is, JAVA code and JNI-exposed native code), while [130] compiles C code into an extended JAVA bytecode that can be analyzed by existing JAVA analyzers. [71, 72] perform type inference on so-called *foreign function interfaces*, that is, inter-language communication frameworks like JNI, discovering type errors otherwise visible only at runtime. [90] instead detects mishandling of exceptions and errors raised in native code and then propagated back to JAVA. The work presented in [89] computes semantics summaries from *guest* programs, to be used during the analysis of a *host* program. All of these approaches share the same underlying idea: to compute summaries among a family of “secondary” programs and use them to analyze the main one (here, programs are defined as coherent modules written in the same language). While modular summary-based analyses are powerful and scalable, not all properties can be proven bottom-up, and often require precise context-sensitive analyses. Moreover, not all programs can be described as a single processing entity exploiting auxiliary codebases: for instance, mobile apps contain logic on both the app and the backend, with back-and-forth communication between

the two.

More recently instead, solutions for multilanguage analyses have been proposed. [28, 27] provide an algebraic framework for multilanguage abstract interpretations by combining existing language-specific ones through *boundary functions* that perform state conversion when switching context between languages. [131] provides LARA [109] source-to-source compilation to transform JAVA, C, C++, and JAVASCRIPT programs towards a common syntax, over which static analyses can be run. Authors however focus on source-code metrics (that is, syntactic properties), with no reasoning on runtime behaviors (that is, semantic properties).

The major alternative to the approach described in this thesis is undoubtedly Mopsa [83] (**M**odular **O**pen **P**latform for **S**tatic **A**nalysis), a static analyzer based on the abstract interpretation theory written in OCAML. Mopsa is designed to compute fixpoints by induction on a program's syntax. A program is an extensible abstract syntax tree (AST) that initially contains the original source code, but that can be syntactically and semantically rewritten during the analysis. Abstract domains share a common interface, and are thus easy to compose and extend. Moreover, the domains are responsible for dynamically rewriting fragments of the AST exploiting semantic information, avoiding static translation towards an internal language. Depending on both the target programming languages and the properties of interest, Mopsa's analyses need to be configured by composing a chain of abstract domains that will dynamically rewrite portions of the AST until a common syntax is reached, over which the remaining domains can operate independently from the source language. Mopsa has been successfully used to analyze C and PYTHON programs [83], showcasing its ability to target completely different languages, including dynamic ones. Moreover, analyses on a combination of the two have been performed [100].

2.4.2 Modeling libraries and frameworks

Several static analyzers, like Julia and FindBugs [79] allow a developer to instruct the analysis about specific runtime behaviors of a framework through annotations. The goal of our work is to automatically produce such annotations and apply them to the code under analysis. In this way, we decouple the framework specification from the program.

Specification languages such as the Java Modeling Language [88] allow to specify pre- and post-conditions and object invariants following the design-by-contract methodology. Different verification tools can then check if the program satisfies the given specification. Therefore, these languages are aimed at specifying the properties of interests that one might want to check on a program, rather than the behavior of frameworks (that is instead our goal).

SLIC is a specification language developed about two decades ago “designed to specify the temporal safety properties of APIs implemented in the C programming language” [19]. SLIC was designed to specify the behavior of libraries, and in partic-

ular safety temporal requirements of the APIs. Instead, our work is focused towards frameworks that might both provide external libraries and modify the runtime execution model of the program, and it targets various safety and security properties of such programs. Our approach can straightforwardly be extended to frameworks and libraries in different contexts.

Previous works [30, 133] relied on hardcoded knowledge of specific framework features, hence building an a priori model for each framework. However, handling new frameworks required a modification of the analysis engine, expanding both the size and complexity of the product and requiring a developer with expertise both in the framework itself and in static analysis. Moreover, this solution did not provide a fast and reliable way for supporting new frameworks, and it did not document which features of a specific framework are taken into account by the analysis and how.

More recent works exploited a framework's configuration files. These often restrict the possible executions, allowing to (almost always) precisely resolve the targets of reflective calls in the framework. F4F [127] aimed at building an application-specific model of the framework's behavior automatically, and this can be used by the analysis to react accordingly to modifications of the execution model made by the framework itself. However, building a model generator for each framework does not keep the actual pace of releases of new frameworks, and each analysis needs to be modified to be model-aware during its execution. Concerto [132] combined mostly-concrete interpretations of the framework code and abstract interpretation of application code, providing sound and accurate analysis on the overall program. Both of the above approaches targeted frameworks whose behavior depends on some application-specific configuration files, and that is not the case for any framework. Moreover, the code from the framework itself needs to be submitted to the analysis, and this will eventually slow down the analyzer due to the significantly larger amount of code to analyze. Lastly, if the format of the configuration file changes among different versions of the same framework, a new parser for such a file must be built, and the logic of the newly introduced constructs must be embedded into the analyzer.

StubDroid [16] built up data flow summaries of Android libraries for taint analyzers. If on the one hand such approach is completely automatic, on the other hand it is specific for taint analysis and it required an ad-hoc static analysis to infer the data flow summaries. While annotations specifying behaviors of frameworks and libraries might be automatically inferred with static and dynamic analyses, this is not the focus of our work and is left as future work. Averroes [4] introduced a new approach that, starting from the code of application and libraries, built up a placeholder library that soundly approximates the library behaviors. The construction of the placeholder library relied on the separate compilation assumption, and it handles reflection. Such an approach sensibly improved the efficiency of the analysis without affecting its precision and soundness. However, more recent frameworks rely

on ad-hoc runtime environments that extend the execution model of the programming languages. These runtime environments are outside the code of the library, and therefore they cannot be handled with this approach.

2.4.3 String analysis

The original finite state automata abstract domain [14] that our work builds upon was defined in the context of dynamic languages, providing an automata-based abstract semantics for common ECMAScript string operations. The same abstract domain has been integrated also for defining a sound-by-construction analysis for string-to-code statements [13].

The problem of statically analyzing strings has been also tackled in different contexts [39, 14, 108, 36, 93, 2, 41]. The authors of [6] provided an automata abstraction merged with interval abstractions for analyzing JAVASCRIPT arrays and objects. In [36], the authors proposed a static analysis of JAVA strings based on the abstraction of the control-flow graph as a context-free grammar. Regular strings [35] is an abstraction of the finite state automata domain and approximates strings as a strict subset of regular expressions. Even if it does not tackle the problem of analyzing strings, in [97] a lattice-based generalization of regular expressions was proposed, showing a regular expression-based domain parametric from the lattice of reference. An interesting automata-based model is symbolic automata [51], that differs from the standard one having an alphabet of predicates (that can potentially be infinite) instead of single characters. Examples of applications of symbolic automata in static analysis are regex processing, sanitizer analysis [138], and their usage as program model for mixing syntactic and semantic abstractions over the program [112].

Finally, orthogonally to static analysis of strings by abstract interpretation, a big effort was spent in the context of string constraints verification, focusing on the study of decidable fragments of the string constraint formulas [3] or proposing new efficient decidable procedures or string constraints representations [3, 34, 7] also based on automata, such as [140, 145], or involving type conversion string constraints [1].

Part II

Multilanguage analysis

3 Towards a multilanguage analyzer: LiSA

Chapter Contents

3.1	Overall architecture	32
3.2	The language of the analyzer	35
3.2.1	Control flow graphs	36
3.2.2	Symbolic expressions	37
3.3	The analysis state	39
3.3.1	Lattice	41
3.3.2	Semantic Domain	42
3.3.3	Value Domain	43
3.3.4	Heap Domain	47
3.3.5	Abstract State	48
3.3.6	Analysis State	50
3.4	Interprocedural Analysis	50
3.4.1	Call Graph	52
3.5	Frontends	53
3.6	Modeling library behavior: SARL	54
3.6.1	Julia	57
3.6.2	The SARL Language	58
3.6.3	Experimental Results	65
3.7	Multilanguage analysis	71
3.8	LiSA for teaching	74
3.9	Conclusion	77

The core contribution of this thesis is the design and implementation of LiSA (**Library for Static Analysis**), an open-source library written in Java that eases the creation of static analyzers by abstract interpretation. LiSA is structured to target as many programming languages as possible, also enabling the analysis of applications written in several of them. It puts a lot of emphasis on modularity: analysis components are strongly decoupled from one another, making their implementations easily interchangeable and fast to develop. LiSA is available on GitHub¹ and stable releases are published on Maven².

In this chapter, we distinguish logical components of the analysis from the classes and interfaces defining them by using *italics* instead of `typewriter`. The core classes and interfaces are depicted through class diagrams. Whenever one of such diagrams is parametric to one or more types, they are shown in a dashed rectangle in the top-right corner. We will use the JAVA notation to bind type parameters: if classes A, B,

¹<https://github.com/lisa-analyzer/lisa>.

²<https://search.maven.org/search?q=g:io.github.lisa-analyzer>.

and C exist, and B is parametric to a type T , we write “ A is subtype of $B\langle C \rangle$ ” to state that A inherits from B and binds its type parameter T to the type C . The binding is also displayed on the diagram with the label $T: B$ on the inheritance relation.

The remainder of this chapter is structured as follows. A high-level overview is first introduced in Section 3.1, depicting the role of all the analysis components and how they cooperate to perform analyses, with the following sections providing an in-depth description of each. *CFGs* and *symbolic expressions* are discussed in Section 3.2, defining the languages that LiSA uses for syntax and semantics, respectively. Section 3.3 provide a bottom-up description of the modular *Analysis State* used to represent pre- and post-states. The *Interprocedural Analysis* that orchestrates LiSA’s fixpoint is described in Section 3.4. In Section 3.5, we define the role of *frontends* in compiling the source programs into LiSA’s *CFGs*. We conclude the chapter with the formalization of SARL in Section 3.6, reporting its implementation in Julia, followed by a proof-of-concept multilanguage analysis on the IoT network of Section 1.2.1 in Section 3.7, and a discussion on our teaching experience with LiSA in Section 3.8.

It is worth recalling here the five principles reported in Section 1.5: separation of syntax and semantics (**P1**), parametrization (**P2**), extensibility (**P3**), modularity (**P4**), and ease of use (**P5**). Throughout this chapter, whenever a component is defined following one or more of such principles, these will be mentioned using the labels reported inside parentheses.

This chapter is based on a book chapter to appear in “Challenges of Software Verification”³.

3.1 Overall architecture

We begin by providing a high-level overview of the analysis pipeline, that is shown in Figure 3.1. The analysis starts by logically partitioning the input application P into programs P^i , each written in a single programming language L^i . L^i -to-*CFG* compilers, called *frontends* (top-left corner of Figure 3.1), are invoked on such programs to obtain a uniform representation of all the code to analyze in the form of a LiSA program P_L^i . *Frontends* are more than compilers, as they also provide logic to the analysis, such as language-specific semantics of the instructions appearing in *CFGs*, semantic models for library code, and language-specific algorithms for common language features (e.g., call resolution and inheritance rules). The final version P_L of the translated program to analyze is the union of all P_L^i . At this point, LiSA can be invoked on P_L with a configuration of the analysis features and the implementations of the various components that are to be executed, namely:

- the *Interprocedural Analysis*, responsible for the computation of the overall program fixpoint and for computing the results of function calls;

³The book will contain contributions presented at the CSV workshop (<https://ssv.dais.unive.it/events/challenges-of-software-verification-workshop/>)

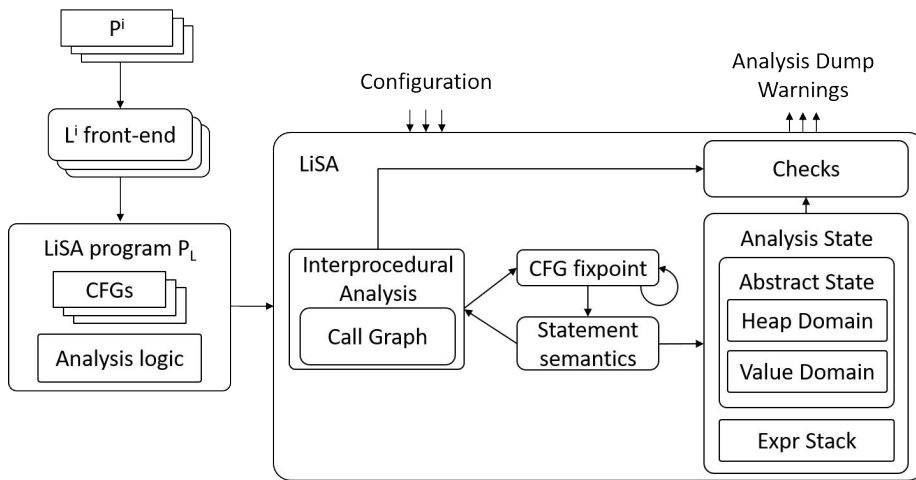


Figure 3.1: LiSA's architecture

- the *Call Graph*, that can be used by the *Interprocedural Analysis* to find call targets;
- the *Abstract State*, that computes the abstract values during the analysis;
- the set of *Checks*, that produce warnings for the user based on the result of the analysis.

P_L is fed to the *Interprocedural Analysis* (left-most block within LiSA in Figure 3.1), that will compute a fixpoint over it. When the *Interprocedural Analysis* needs to analyze an individual *CFG*, it will invoke a unique fixpoint algorithm defined directly on *CFGs* (central portion of LiSA in Figure 3.1). As the language-specific semantics of instructions is embedded in *CFG* nodes, called *Statements*, the fixpoint algorithm will use such semantics as *transfer function*. If the *Statement* performs calls as part of its semantics, it will interact back with the *Interprocedural Analysis* to determine the returned values, as how those are evaluated depends directly on how the overall fixpoint is computed. If the call's targets are unknown (for instance, if the call happens in a language with dynamic method dispatching), the *Interprocedural Analysis* can delegate targets resolution to the *Call Graph* (inner component of *Interprocedural Analysis* in Figure 3.1), that will use type information together with the language-specific execution model to determine all possible targets. Alternatively, a non-calling *Statement*'s semantics can also rewrite the node into a sequence of *symbolic expressions*, that is, atomic instructions with precise semantics, that can be passed to the *Analysis State* for evaluation. LiSA's *Analysis State* (right-most block of LiSA in Figure 3.1) is composed of an *Abstract State* modeling the memory state at a given program point, together with a collection of *symbolic expressions* that are left on the stack after evaluating it. An *Abstract State* is a flexible entity whose main duty is to make the *Value Domain*, responsible for tracking values of program variables,

```

1 class A {
2   int f;
3   int g;
4   void main(String[] args) {
5     A a = new B();
6     a.foo(10);
7   }
8   int foo(int w) {
9     return w + 2;
10  }
11 }
12 class B {
13   int foo(int w) {
14     this.f = w + 3;
15     this.g = this.f + 2;
16     return this.g + this.f;
17   }
18 }

```

Figure 3.2: Running example for LiSA’s architecture overview

communicate with the *Heap Domain*, that instead tracks how the dynamic memory of the program evolves at runtime. Whenever an expression is to be evaluated by an *Abstract State*, the latter first passes it to the *Heap Domain*, that will record all of its effects on the memory, such as the allocation of new regions or the access to some object’s fields. Then, the *Heap Domain* will rewrite all portions of the original expression that deal with the memory. According to the implementation-specific logic, one or more instrumented variables will be used to replace such portions, modeling their resolution to memory addresses that can be treated as regular variables. After the rewriting has been performed, the resulting expression will be passed to the *Value Domain*, that will track values and properties regarding the variables appearing in it. Note that, with this architecture, each component simplifies the program for the rest of the analysis pipeline. *Interprocedural Analysis* abstracts away calls from the program to analyze, leaving the *Analysis State* and its sub-components with non-calling programs. Successively, the *Heap Domain* removes every expression that deals with dynamic memory, substituting it with synthetic variables. At this point, the *Value Domain* has to deal with programs containing only variables, constants, and operators between them.

When an overall fixpoint is reached, the computed pre- and post-states for each Statement, together with the *Call Graph* that has been built up, are passed to the *Checks* (top-right corner within LiSA in Figure 3.1) that have been provided to the analysis. These are simply visitors of the program’s syntax, that can use the information computed by the analysis to issue warnings to the user. Since these are standard components of static analyzers, they will be omitted from this work.

Example 3.1.1. Consider the example JAVA code from Figure 3.2. To analyze it, a *JAVA frontend* will first parse the code and produce three different *CFGs*, one for each method. Supposing that a context-sensitive [120] approach has been selected for the *Interprocedural Analysis*, the analysis could follow call-chains starting

from the `main CFG`, analyzing callees are they are invoked. Thus, the first fixpoint algorithm to be invoked would be the one of the `main CFG`. Here, when the call to `foo(10)` at line 6 is encountered, an entry state for the targets of the call would be set up by assigning 10 to `w`. How the call would be resolved to its targets depends once again on the configuration. Supposing that the *Call Graph* implementation uses the runtime types inferred for the receiver [129], the call will be resolved to `B.foo` that can be analyzed (that is, whose fixpoint can be executed) using the prepared state.

The code of `B.foo` deals with heap structures. The assignment at line 15 could be rewritten as `l_0 = l_1 * 2` if the *Heap Domain* is precise enough to distinguish between different fields of the same object, or as `l_0 = l_0 * 2` if it is not. Nonetheless, the resulting expression will not contain any reference to memory structures, and can be then processed by the *Value Domain* (e.g., *Intervals* [43]) agnostically w.r.t. if and how a rewriting happened.

3.2 The language of the analyzer

Before examining the separate components of `LISA`, we introduce and discuss (i) the *CFG* structure that `LISA` uses for representing programs, and (ii) the *symbolic expression* language used as intermediate representation for the analysis.

As programming languages come with wildly different syntaxes, it is important to find a common ground to model their semantics so that analyses are not required to handle constructs from all languages. This is a common practice among static analyzers, even ones targeting a single language: moving to a uniform and more convenient intermediate representation (IR) that is usually enriched with additional information (e.g., runtime typing) can make writing analyses much easier. Different syntactic constructs with the same (or similar) meaning can be represented by the analyzer as a unique IR construct, and complex ones can be decomposed as a sequence of them. Analyses can then attribute semantic meanings to such IR constructs with no knowledge of the original syntactic ones they represent.

Rewriting towards the IR can typically be achieved at parse time, after ingesting the target application, or at analysis time, before passing the code to the abstract domains. `LISA` implements hybrid rewriting: first, source code is compiled to *control flow graphs (CFGs)* by *frontends*, then each *CFG* node is rewritten into one or more *symbolic expressions* during the analysis. The double rewriting is in place to address **P1**, as *CFGs* embed syntactic structures and language-specific constructs within them (i.e., `+` is still a syntactic construct that might represent numeric addition, string concatenation, ...), while each *symbolic expression* has a unique semantic meaning.

`LISA`'s programs are thus composed of *CFGs*, that can be logically grouped in `CompilationUnits`, a generalization of the concept of *classes* in object-oriented software. As both the structure and meaning of `CompilationUnits` mirror the one of

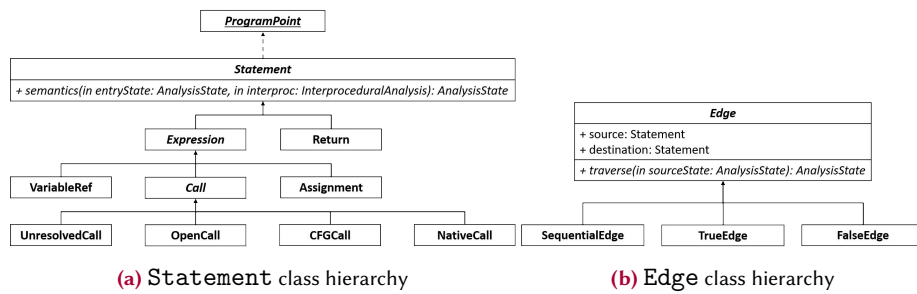


Figure 3.3: Statements and Edges

classes (with some additional parametrization for language-specific features like multiple inheritance), and do not directly influence the infrastructure of LiSA, their definition is omitted in this work. Thus, we will refer to LiSA programs as a collection of *CFGs*.

3.2.1 Control flow graphs

Control flow graphs [5] (CFGs) are directed graphs that express the control flow of functions. In a CFG, nodes contain the instructions of the functions, and edges express how the execution *flows* from one node to another. This means that all the syntactic constructs that form loops, branches, and arbitrary jumps are directly encoded in the CFG structure, simplifying the code to analyze. Following **P3**, LiSA's *CFGs* are extensible: *Statement* (Figure 3.3a) and *Edge* (Figure 3.3b) are the base definitions of what nodes and edges are, respectively, while concrete instances are defined in *frontends*.

Figure 3.3a shows a portion of the class hierarchy of the *Statement* class natively provided by LiSA, that is the base class for *CFG* nodes. A *Statement* represents an instruction appearing in a function, and thus corresponds to a syntactic construct that does not modify the control flow (that is, it is not a loop, a branch, or an arbitrary jump). When the evaluation of a *Statement* leaves a value on the operand stack, it is called an *Expression* whose type is the one of the generated value. Examples of *Statements* are *return* and *assert*, while an *Expression* can be a reference to a local variable by its name, an assignment, or a sum. The *Call* expression, together with its descendants, plays a central role in LiSA and will be further discussed in Section 3.4. As advocated by **P1**, no *Statement* has predefined semantics: in fact, the class defines a *semantics* method where implementers can define language-specific reasoning (thus also fulfilling **P2**) and interact with *entryState* (instance of *AnalysisState* representing the pre-state) and *interproc* (instance of *InterproceduralAnalysis* offering interprocedural reasoning) to compute the post-state for the *Statement*. A further mean to fulfill **P2** is the *ProgramPoint* interface, implemented by the *Statement* class. This interface provides a simplified view of an instruction (real or instrumented) that can be given to analysis compo-

nents that need high-level knowledge about syntactic constructs (e.g., knowing if an instruction is part of a loop's body).

The `Edge` class (whose hierarchy is depicted in Figure 3.3b) is the base class for *CFG* edges. The `traverse` method defined by this class expresses how the post-state of its source node, in the form of an `AnalysisState` instance, is transformed when the edge is traversed. `LiSA` comes with three default implementations for edges: `SequentialEdge`, `TrueEdge`, and `FalseEdge`. `SequentialEdge` represents an unconditional flow of execution from source to edge, with no modification of the initial `AnalysisState` (i.e., its `traverse` implementation returns its parameter unaltered). `TrueEdge` instead models a flow of execution conditional to the evaluation of the expression at its source: the execution proceeds by reaching the edge's destination only if it evaluates to `true`. Conversely, `FalseEdge` implements a conditional flow of execution that reaches the edge's destination only if the expression at its source evaluates to `false`. For both `TrueEdge` and `FalseEdge`, the implementation of `traverse` relies on methods defined in the `SemanticDomain` interface (Section 3.3.2) that is implemented by `AnalysisState`: first, the expression at its source is tested through the `satisfies` method, and if the test's result is `UNKNOWN` or corresponds to the edge's semantics (i.e., `SATISFIED` for `TrueEdge` and `NOT_SATISFIED` for `FalseEdge`), the `assume` method is used to filter the source state accordingly. Otherwise, the return value of `AnalysisState.bottom()` (that is defined by the `Lattice` interface, Section 3.3.1) is returned.

Finally, as a mean to model library functions in addition to the one presented in Section 3.6, `LiSA` offers *native CFGs*, that is, *CFGs* with a single `Statement` and no `Edges`. Whenever a call to one of these *CFGs* is found, the call's result can be evaluated by simply rewriting it into the only statement contained in the *native CFG*, and then executing its `semantics` method. Modeling complex or frequently used library functions through *native CFGs* can drastically reduce the complexity of the analysis, as less code needs to be analyzed, while still providing all the necessary information about the modeled functions.

3.2.2 Symbolic expressions

A static analyzer's main duty is to compute program properties by taking into account the semantic meaning of the instructions appearing in the program. In this context, an extensible set of syntactic constructs such as the one provided by `LiSA` through *CFGs* comes with an intrinsic problem: instructions (i.e., `Statements`) do not have well-defined semantics, as that is parametric to the source language. To recover well-definedness, `LiSA` adopts a two-phase rewriting: not only is the source program compiled to *CFGs*, but each of their `Statements` gets rewritten into *symbolic expressions* during the analysis.

The `SymbolicExpression` class, whose partial hierarchy is shown in Figure 3.4, is the base type for the expressions that `LiSA`'s analysis components understand and

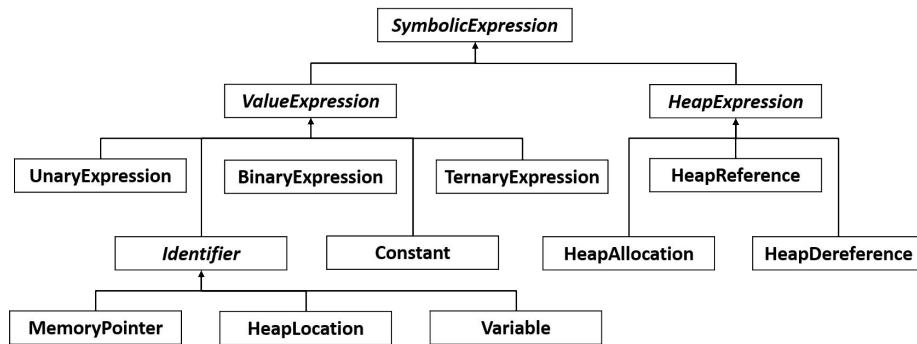


Figure 3.4: The SymbolicExpression hierarchy

analyze. Note that there is a clear distinction between expressions dealing with values of variables (i.e., ValueExpressions) and ones dealing with memory structures (i.e., HeapExpressions). This is a direct consequence of the architecture, introduced in Section 3.1 and thoroughly discussed in Section 3.3.5, that separates domains dealing with the two worlds, decoupling their implementations. ValueExpressions model what can be handled entirely by the *Value Domain* (Section 3.3.3): constants, variables, and operators between them. Specifically, Identifiers model program variables (either actual variables or ones representing memory locations) that can be *weak* or not, that is, if expressions assigned to them overwrite existing abstractions or should be joined with the previous ones (through \sqcup) to preserve soundness. Operators instead come in the form of Unary, Binary, or TernaryExpressions, depending on how many values they operate on. On the other side, HeapExpressions represent operations that change or navigate the structure of the dynamic memory of the program. For instance, HeapReference references an existing location in the memory, effectively creating a pointer to it, while HeapDereference dereferences one to get back to the actual location. As for CFGs, Statements, and Edges, *symbolic expressions* are also extensible following **P3**. Note that, as discussed in Section 3.1 and deepened in Section 3.4, no *symbolic expression* is defined for calls, as those are abstracted away by the *Interprocedural Analysis*.

Example 3.2.1. To better explain how the second rewriting is carried out, let us consider the expression `new B()` at line 5 of Figure 3.2. In JAVA, object instantiation consists of four operations: (i) allocation of a memory region, (ii) creation of a pointer to the region, (iii) invocation of the desired constructor using the fresh pointer as receiver, and (iv) storage of the pointer on the operand stack. Such behavior could be mimicked by a Statement instance with the following (simplified) semantics function:

```

1 AnalysisState semantics(
2     AnalysisState entryState,
3     InterproceduralAnalysis interproc) {

```

```

4 // create a synthetic receiver
5 VariableRef rec = new VariableRef("$receiver");
6 AnalysisState recState = rec.semantics(entryState, interproc);
7
8 // assign the fresh memory region to the receiver
9 HeapAllocation created = new HeapAllocation();
10 HeapReference ref = new HeapReference(created);
11 AnalysisState callState = entryState.bottom();
12 for (SymbolicExpression v : recState.getComputedExpressions())
13     callState = callState.lub(callState.assign(v, ref));
14
15 // call the constructor
16 String name = createdType.toString();
17 Expression[] params = ArrayUtils.insert(0, expressions, rec);
18 UnresolvedCall call = new UnresolvedCall(name, name, params);
19 AnalysisState sem = call.semantics(callState, interproc);
20
21 // leave a reference on the stack
22 return sem.smallStepSemantics(ref);
23 }

```

While the methods exploited in this snippet are the subject of the following sections, we can nonetheless capture the intuition behind them. First, a `VariableRef` (that is an instance of `Statement` and thus modeling the syntactic reference to a program's variable by its name) is created at line 5, mimicking the creation of the constructor call's receiver. Then its semantics is computed at line 6 starting from the pre-state `entryState`, obtaining a new instance of `AnalysisState` (Section 3.3.6) that will contain the `Variable` (an instance of `SymbolicExpression`) corresponding to it. The following five lines are responsible for making such variable point to a newly allocated memory region: line 13 assigns a pointer (line 10) to a region that is being allocated in-place (line 9) to the `Variable` corresponding to the receiver (line 12). Next, the call to the constructor is performed at line 19 by (i) extracting the name of the type created by the expression (line 16, where `createdType` is a field containing the `Type` being instantiated) as both the class and method name, and (ii) adding the instrumented receiver to the original parameters of the constructor call (line 17, where `expressions` is a field containing the original parameters and `ArrayUtils.insert` is a method that clones an array and adds a new element to it). The semantics of the `UnresolvedCall` created at line 18 will defer the resolution and evaluation to the *Interprocedural Analysis*. The post-state of the call is then used at line 22 to reprocess the reference to the memory region through `smallStepSemantics`, returning the result as the final post-state of the whole instruction.

3.3 The analysis state

The state of LiSA's analyses is modularly built (addressing **P4**) bottom up, ensuring that each component does not have visibility of its parents and siblings. This also helps achieving **P5**, as no additional knowledge is needed to implement a com-

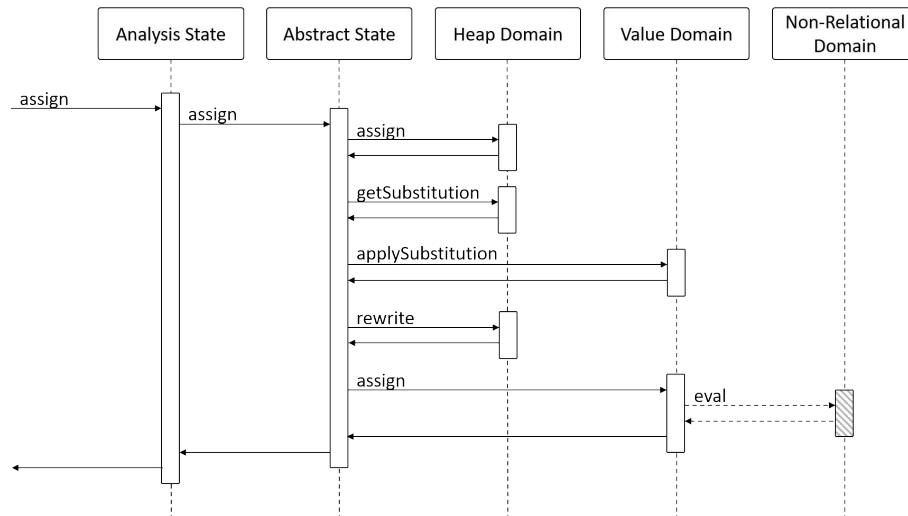


Figure 3.5: Sequence diagram *Analysis State*'s *assign*

ponent other than what is strictly required by it. As discussed in Section 3.8, this plays an important role in picking up LiSA quickly. We illustrate such structure starting from the `Lattice` (Section 3.3.1) and `SemanticDomain` (Section 3.3.2) interfaces, that define ordered structures and abstract operators, respectively. We then proceed bottom up, following the state's natural structure, introducing the *Value Domain* in Section 3.3.3, the *Heap Domain* in Section 3.3.4, and the *Abstract State* combining them in Section 3.3.5. Lastly, the *Analysis State* class in Section 3.3.6 wraps all previous components, providing the final state used by LiSA's analyses.

To grasp the intuition of how the *Analysis State* operates, consider the sequence diagram of Figure 3.5, depicting how the *assign* method (one of the semantics operations that will be introduced with `SemanticDomain`) behaves. Note that the pattern shown here is also valid for the other semantic operations, as it follows the overall communication scheme defined in [61]. When the *assign* method of the *Analysis State* is invoked, the call is immediately forwarded to the *Abstract State*. The latter will first compute the effects of the assignment on the dynamic memory through the *Heap Domain*'s own *assign* method. Then, since such an operation might have caused materialization or merge of heap identifiers (Section 3.3.4), the *Abstract State* retrieves a *substitution* (i.e., a replacement of variables in the pre-state with variables of the post-state) from the *Heap Domain*, and uses it to update the *Value Domain*. Then, *Heap Domain*'s *rewrite* replaces portions of the assigned expression dealing with dynamic memory (e.g., field accesses) with heap identifiers, rendering the right-hand of the assignment *memory-free* (that is, only dealing with variables, constants, and operations between them). The updated *Value Domain* instance is then used to evaluate the effects of the assignment on program variables, using *Value Domain*'s *assign* method on the rewritten expressions. The new *Heap* and *Value Domain* instances are then wrapped into a fresh *Abstract State* object, that is returned to the

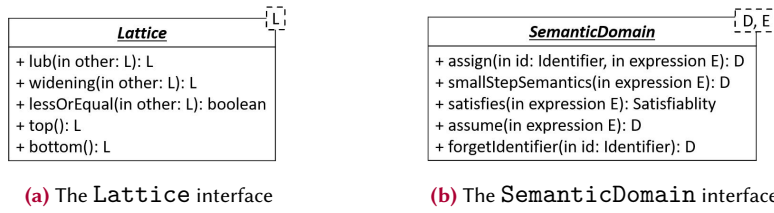


Figure 3.6: The core LISA interfaces

caller as part of the final *Analysis State* that is built as result of the original call. Note that, as *Abstract State*, *Heap Domain*, and *Value Domain* are defined modularly, each computational step might hide additional complexity (for instance, the *Value Domain* could be a Cartesian product of several instances, whose `assign` methods are recursively invoked). Moreover, the *Value Domain* can optionally rely on an inner *Non-Relational Domain* (Section 3.3.3, marked with diagonal stripes in Figure 3.5) instance to compute the semantics of an expression, exploiting its `eval` method.

A design choice common to most components illustrated in this section is to be parametric on a type that must be an instance of the component itself. This is a common technique used for implementing generic classes in JAVA: when a class is declared as `class C<T> extends C<>` and defines methods such as `void func(T param)`, implementers can bind the parameter to themselves with `class A extends C<A>` to ensure that inherited methods will only accept instances of themselves as parameters, such as `void func(A param)`. For the sake of conciseness, we refer to this kind of type parameters as *instance binders*.

3.3.1 Lattice

Ordered structures in LISA implement the `Lattice` interface (Figure 3.6a). The latter has an instance binder `L` and defines the following methods:

- `L lub(L other)`, that returns the least upper bound \sqcup between the receiver of the call and the element passed as parameter;
- `L widening(L other)`, that applies the widening operator ∇ to the receiver of the call and the element passed as parameter, returning its result;
- `boolean lessOrEqual(L other)`, implementing the partial order \sqsubseteq , and returning `true` if and only if `this` \sqsubseteq `other`;
- `L top()`, returning the top element \top of the lattice;
- `L bottom()`, yielding the bottom element \perp of the lattice.

For the sake of convenience, `widening` has a default implementation that delegates to `lub`. While this is generally correct only for ACC lattices, it minimizes the operators that one is asked to implement while letting non-ACC lattices redefine it.

Moreover, while requiring the existence of a top element might seem too restrictive (as abstract interpretation can work with `cpos`), an element to use as a worst-case over-approximation is usually required by static analyzers. Such an element is used whenever the analyzer cannot precisely compute the semantics of some program feature, or has to give approximations for unknown values: for instance, user inputs and results of reflective calls are usually abstracted with such an element. As \top is the less precise element of an ordered structure, we avoid asking users of LiSA for a custom worst-case approximation and use \top whenever a worst-case assumption is needed.

To fulfill **P4** and **P5** more closely, reoccurring features of implementations should have a unique shared implementation that is parametric to instance-specific logic. In fact, LiSA provides a base implementation for `Lattice` in the `BaseLattice` class. The latter implements `lub`, `widening`, and `lessOrEqual` by handling common cases (e.g., $\forall e. \perp \sqcup e = e$) and then delegating each non-trivial computation to the concrete implementation.

3.3.2 Semantic Domain

LiSA's `SemanticDomain` interface (Figure 3.6b) defines the minimum set of operations that an entity reasoning about a program's semantics must support. Similarly to `Lattice`, `SemanticDomain` has an instance binder `D`. Moreover, a second type parameter `E`, that must be a subtype of `SymbolicExpression`, exists to restrict the type of expressions that a domain is able to handle. Each instance of this interface represents some abstract information about program variables, and can thus be used as pre-state for semantic computations and can be obtained by one. The following methods are defined by the interface:

- `D assign(Identifier id, E expression)`, that computes the semantics of assignments, yielding the post-state of assigning `expression` to `id` using the receiver of the call as pre-state;
- `D smallStepSemantics(E expression)`, used to compute the semantics of a non-assigning expression, and returning the post-state of `expression`'s semantics starting from the pre-state of the receiver;
- `Satisfiability satisfies(E expression)`, determining if `expression` is always, never, or sometimes satisfied by the information contained in the current domain instance (`Satisfiability` is a predefined lattice used in LiSA to express the result of Boolean evaluations: possible values are `SATISFIED`, `NOT_SATISFIED`, `UNKNOWN`, and `BOTTOM`);
- `D assume(E expression)`, that (optionally) refines the receiver of the call by assuming that `expression` holds;
- `D forgetIdentifier(Identifier id)`, that forgets all abstract information regarding a specific identifier.

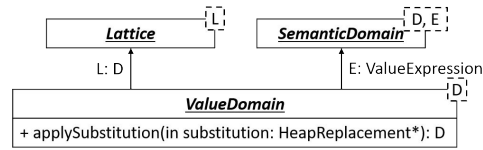


Figure 3.7: The ValueDomain hierarchy

Following **P2**, all methods of `SemanticDomain` must be parametric to the source language of the program point under semantic evaluation. While statements from different languages can use their own symbolic expressions, we offer an additional mechanism to rely on language-specific reasoning by means of the `ProgramPoint` interface. All methods of `SemanticDomain` accept an additional `ProgramPoint` parameter (left out of Figure 3.6b and from the earlier discussion for conciseness), enabling language-specific reasoning by querying its methods.

3.3.3 Value Domain

LiSA’s *Value Domain* is the analysis component responsible for tracking properties of program variables. As we are presenting LiSA’s infrastructure bottom-up, *Value Domain* is the *last* component taking action to compute a `SymbolicExpression`’s semantics. Interface `ValueDomain` (Figure 3.7), that is parametric only to its instance binder `D`, defines the operations that a *Value Domain* must support. It inherits both from `Lattice<D>` and `SemanticDomain<D, ValueExpression>`. Implementations of `ValueDomain` thus represent both elements in the ordered structure they belong to, and abstract transformers able to produce new elements. Moreover, as mentioned in Section 3.1, a *Value Domain* is only responsible for dealing with variables, constants, and operators between them. The type parameter `E` from `SemanticDomain` is thus bound to `ValueExpression`, forcing the domain to only accept call- and memory-free expressions. In this setup, `ValueDomain` implementations can closely resemble their formalization, that typically provides an abstract semantics only for the operations of interest. `ValueDomain` defines `applySubstitution` as a mean to interact with the *Heap Domain*. While such feature will be thoroughly discussed in Section 3.3.5, it can be intuitively described as a remapping of abstract information from a set of variables to another one. Examples *Value Domain* implementations are `Interval` [43], `Polyhedra` [50], and `Tarsis` (Chapter 6). Moreover, combinations of *Value Domains* are still *Value Domains*: products [38] and smashed sums [12] can thus be used to execute different domains concurrently, potentially enabling cooperation to compute properties.

As for the `Lattice` interface, reoccurring features of *Value Domains* such as the environment that is typical of Cartesian (non-relational) abstractions should have a unique shared implementation that modularly interacts with domain-specific logic to address **P4** and **P5**. We therefore identify two common types of *Value Domains* that

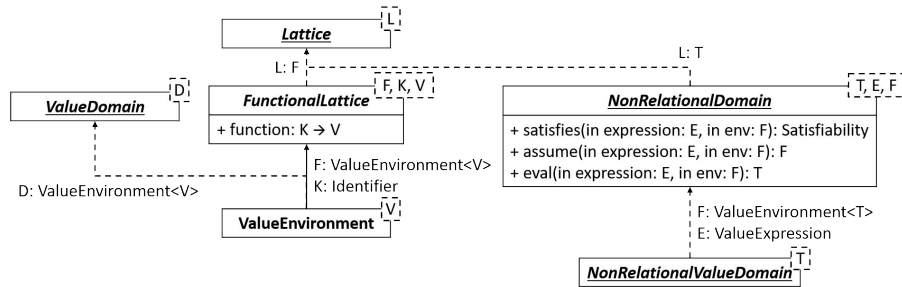


Figure 3.8: The NonRelationalDomain hierarchy

fit this situation, and accordingly provide separate infrastructures: *Non-Relational Domains* and *Dataflow Domains*.

Non-Relational Domains

Formalization of domains that reason on each variable independently without tracking relations among them usually follows the same pattern. First, an ordered structure is introduced to model values of individual variables that the domain will track. Then, an environment (i.e., a map from variables to instances of the ordered structure) is used to hold separate abstractions for each variable. The environment itself is an ordered structure: it is effectively a functional lift of domain instances to variables, and the operators are simply point-wise applications of the original ones from the ordered structure of the domain. Lastly, the domain is equipped with operators that, given an environment providing abstractions of each variable, can compute the abstract value of an expression.

LiSA defines the *Non-Relational Domain* infrastructure (Figure 3.8) as a mean to simplify implementations of Cartesian abstractions. At first, the point-wise logic is implemented into class `FunctionalLattice`, representing the functional lift of values of type `V` to keys of type `K`, where both are parametric. In addition, the class also has an instance binder `F`. `V` must also be a subtype of `Lattice<V>`, thus ensuring that the point-wise implementations can delegate to values of the co-domain. `ValueEnvironment` is then defined to obtain a `ValueDomain` whose ordered structure is composed of functions. It is parametric to the type `V` of its values, that must be an instance of `NonRelationalValueDomain<V>`, an interface that inherits from `Lattice` that will be introduced shortly. `ValueEnvironment` implements the `ValueDomain` interface binding the type parameter `D` to `ValueEnvironment<V>` and extends `FunctionalLattice<ValueEnvironment<V>, Identifier, V>` (hence being a function mapping program variables to abstract values).

LiSA then defines the `NonRelationalDomain` interface to model the domain-specific logic of Cartesian abstractions. It has three type parameters: an instance binder `T`, the type `E` of `SymbolicExpressions` that the domain can handle, and the kind `F` of `FunctionalLattice<F, Identifier, T>` that the domain is designed

to work with. `NonRelationalDomain` inherits from `Lattice<T>`, and defines three additional methods:

- `Satisfiability satisfies(E expression, F env)`, determining if the given expression is always, never, or sometimes satisfied when the variables appearing in it assume the values contained in `env`;
- `F assume(E expression, F env)`, that (optionally) refines `env` by assuming that `expression` holds with the variables appearing in it take the values defined in `env`;
- `T eval(E expression, F env)`, that is responsible for computing an abstract value for the given expression assuming that values of program variables are the ones in `env`.

`NonRelationalValueDomain` specializes `NonRelationalDomain` by binding `F` to `ValueEnvironment<T>` and `E` to `ValueExpression`, while maintaining its instance binder `T`. Such interface represents an ordered structure that can compute abstractions for single expressions given the ones for each program variable, thus modeling a Cartesian abstraction. In fact, `ValueEnvironment` implements the semantic operations defined in `SemanticDomain` by delegating the evaluation of each expression to the `NonRelationalValueDomain` that it contains. Similarly to `Lattice`, `LISA` provides a base implementation for `NonRelationalValueDomain`, factoring out common features (**P4** and **P5**), in the `BaseNonRelationalValueDomain` class. The latter implements `eval` and `satisfies` providing automatic recursive evaluation of sub-expressions, while also extending `BaseLattice` (Section 3.3.1) to provide base implementations of lattice operations.

As `LISA` provides a full implementation for the `ValueEnvironment` class, non-relational domains such as `Interval` can be implemented by just (i) providing lattice operations for single intervals, and (ii) defining the logic for expression evaluation. `LISA` then takes care of wrapping the domain inside a `ValueEnvironment`, providing a unique functional lifting to all Cartesian abstractions. An example implementation is presented at the end of Section 3.8 with the `Sign` domain.

Type Inference. A special case of Cartesian abstractions are type systems, that compute the possible runtime types of expressions and variables. In fact, `LISA` runs type inference exploiting the *Non-Relational Domain* infrastructure, with few modifications. The `NonRelationalTypeDomain` interface is defined as a base for domain-specific reasoning about types. Similarly to `NonRelationalValueDomain`, it has an instance binder `T` and it inherits from `NonRelationalDomain`. Type parameter `F` is bound to `TypeEnvironment<T>` (a special instance of `ValueEnvironment`), while the parameter `E` is bound to `ValueExpression`. It also introduces a new method `Set<Type> getInferredRuntimeTypes()` that yields what has been inferred as possible types of the last expression evaluated. This method is used by

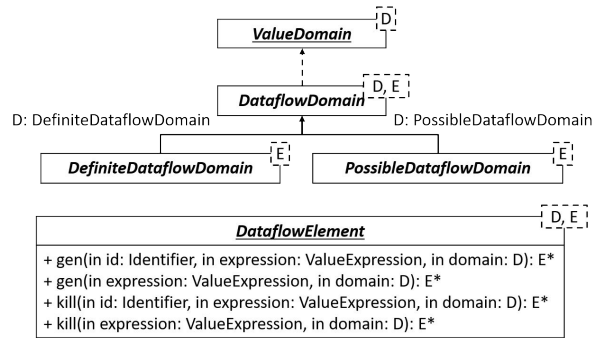


Figure 3.9: The DataflowDomain hierarchy

TypeEnvironment to store the set of types computed by the domain every time a new expression is evaluated.

Dataflow Domains

Similarly to Cartesian abstractions, dataflow analyses [70] also have fundamental features that follow a common pattern coupled with implementation-specific logic. Their formalization usually starts with a set of elements that represent abstract information. In terms of abstract interpretation, if the set of all possible dataflow elements is D , the ordered structure backing the domain is the complete lattice $\langle \wp(D), \subseteq, \cup, \cap, \emptyset, D \rangle$ if the domain is *possible*, that is, if it tracks information that holds on at least a program execution. On the other hand, the complete lattice is $\langle \wp(D), \supseteq, \cap, \cup, D, \emptyset \rangle$ if the domain is *definite*, that is, if it tracks information holding along all program executions. Then, the semantics of each instruction i is defined as the dataflow equation $\text{out} = \text{in} \setminus \text{kill}_i(\text{in}) \cup \text{gen}_i(\text{in})$, where:

- in is the pre-state;
- out is the post-state;
- kill_i is a function that, given a pre-state, yields all of its elements that no longer hold after i is executed;
- gen_i is a function that, given a pre-state, yields all new elements that start holding after i is executed.

The *Dataflow Domain* infrastructure in Figure 3.9 factors common components of dataflow domains, simplifying the implementation process. Class `DataflowDomain` implements the base *Value Domain* to be used when executing dataflow analyses. It has an instance binder D , matching the one of `ValueDomain` that it implements, and it is parametric to the type E of `DataflowElements` (that will be introduced briefly) that it contains. Lattice operations are delegated to its two concrete subclasses: `DefiniteDataflowDomain` and `PossibleDataflowDomain`, implementing

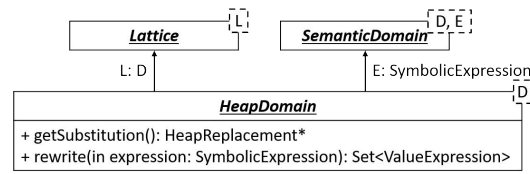


Figure 3.10: The HeapDomain hierarchy

definite and *possible* lattices respectively. Operations defined in `SemanticDomain` are uniquely implemented in `DataflowDomain` through the dataflow equation, whose domain- and language-specific *gen* and *kill* functions are delegated to implementations of the `DataflowElement` interface. The latter has an instance binder `E`, and a type parameter `D` subtype of `DataflowDomain` that expresses the kind of domain the implementation is meant to work with. `DataflowElement` defines two variations of the *gen* and *kill* functions, one for assignments and one for normal expressions, where the analysis-specific reasoning can be implemented.

In this infrastructure, dataflow domains (e.g., `Reaching Definitions`) can be implemented by simply defining an implementation of `DataflowElement` with the *gen* and *kill* logic, specifying the type of `DataflowDomain` that must be used for the analysis. `LISA` will then instantiate such domain and use it to wrap the provided implementation.

3.3.4 Heap Domain

`LISA`'s *Heap Domain* is the analysis component responsible for tracking how the dynamic memory of the program evolves during its execution. As the sole component having full knowledge of how expressions are resolved to memory locations, the *Heap Domain* operates before the *Value Domain* as it must simplify memory-dealing expressions that the latter cannot handle. From the implementation point of view, the two components have similar characteristics: the `HeapDomain` interface (Figure 3.10) models the component, and it is parametric to its instance binder `D`. Moreover, just as `ValueDomain`, its instances are both elements in the ordered structure they belong to (i.e., `HeapDomain` inherits from `Lattice<D>`), and transformers that produce new elements (i.e., it inherits from `SemanticDomain<D, SymbolicExpression>`). Note that the type parameter `E` of `SemanticDomain` is bound to `SymbolicExpression`: in fact, this component must be able to track if program variables refer to memory regions, and thus needs to handle both `ValueExpressions` and `HeapExpressions`. Moreover, as Section 3.3.5 will depict, one of *Heap Domains*' duties is to dynamically rewrite `SymbolicExpressions` by replacing all portions that manipulate memory (i.e., that are instances of `HeapExpression`) with one or more `Identifiers` that represent the regions of dynamic memory the replaced expression could resolve to. The rewriting happens through the `rewrite` method offered by `HeapDomain`, that recursively visits an expression and its sub-expressions returning the corresponding

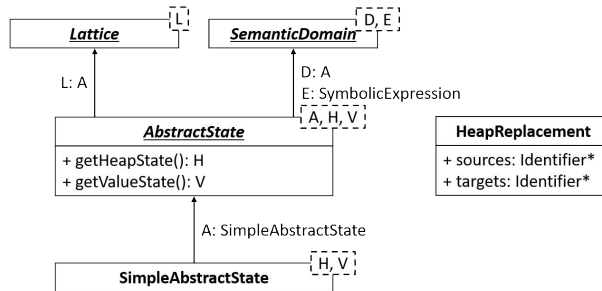


Figure 3.11: The AbstractState hierarchy

ValueExpressions. Its discussion, together with the one of `getSubstitution()`, is postponed to Section 3.3.5. Examples of *Heap Domain* implementations are Andersen’s *Pointer Analysis* [8] and *Shape analysis* [119]. Moreover, as for *Value Domains*, combinations of *Heap Domains* are still *Heap Domains*.

3.3.5 Abstract State

LiSA’s *Abstract State* wraps both *Value* and *Heap Domains*, coordinating their communication. It is designed after the framework presented in [61], where the two communicate by means of expression rewriting and variable renaming. Roughly, the semantics of such framework lets the two domains compute properties independently whenever an expression only deals with values or memory. When one instead requires knowledge about both worlds, the expression is first evaluated by the *Heap Domain* that tracks its effects on the memory. Then, abstract locations called *heap identifiers* are used to replace memory-dealing sub-expressions, and the *Value Domain* can process this rewritten expression to track properties about those identifiers. Furthermore, as the semantics of the *Heap Domain* might materialize or merge heap identifiers, a *substitution* can be applied to the *Value Domain* when necessary, before computing its semantics. A substitution can be described as a sequence of multi-variable replacements between heap identifiers. Algorithm 1 generates a new domain instance by applying a substitution $[I_1 \rightarrow I'_1, \dots, I_n \rightarrow I'_n]$, where each $I \rightarrow I'$ represents a single replacement from the variables of I to the ones of I' . The intuition behind this algorithm, where $result[i' = i]$ is the evaluation of the assignment $i' = i$ in the state $result$, is that each individual replacement assigns all the variables in I' to all the variables in I , thus replacing the latter with the former in the original domain. A substitution is then just a sequence of replacements.

The *AbstractState* interface in Figure 3.11 is modeled after the framework presented in [61]. It can be seen as the product between the *HeapDomain* H and the *ValueDomain* V (both left generic as type parameters). *AbstractState* has an instance binder A and inherits from `SemanticDomain<A, SymbolicExpression>` and `Lattice<A>`. The framework’s implementation is split among all involved components, with the *Abstract State* orchestrating their interaction. Specifically, three

Algorithm 1: Application of a substitution of heap identifiers

Data: V domain, $[l_1 \rightarrow l'_1, \dots, l_n \rightarrow l'_n]$ sub
Result: V result

```

1 result  $\leftarrow$  domain;
2 foreach  $l \rightarrow l' \in$  sub do
3    $v \leftarrow \sqcup \{v' \mid v' = \text{result}[i' = i], i \in l, i' \in l'\}$ ;
4   result  $\leftarrow v$ ;
5 return result;

```

methods are defined between the *Heap* and *Value Domains*:

- `rewrite` from `HeapDomain` replaces memory-dealing expressions with synthetic variables such as `HeapIdentifiers` and `MemoryPointers` (both subtypes of `Identifier`);
- `getSubstitution` from `HeapDomain` gets the substitution generated with the creation of the domain's instance, that is, with the evaluation of the last expression;
- `applySubstitution` from `ValueDomain` yields a copy of the domain where the substitution has been applied (the method has a default implementation that encodes Algorithm 1, exploiting both the `assign` and `forgetIdentifier` methods from `SemanticDomain`).

The concrete implementation of the communication between the *Heap Domain* and *Value Domain* is provided by the `SimpleAbstractState` class. *Abstract State* is left modular and extensible (**P4**) as further layers of abstraction can be applied to the entire state. For instance, `Trace Partitioning` [116] must be applied on the state *as a whole*, and can thus be defined as an `AbstractState` implementing a function from execution traces to `AbstractStates`. Communication between *Heap Domain* and *Value Domain* in `SimpleAbstractState` abides to the scheme of [61], as shown in the following snippet (here, we only report the implementation of `smallStepSemantics`, as the one of the other `SemanticDomain`'s methods can be derived from it):

```

1 SimpleAbstractState smallStepSemantics(SymbolicExpression expression) {
2   // let the heap process and rewrite the expression
3   HeapDomain heap = getHeapState().smallStepSemantics(expression);
4   Set<ValueExpression> exprs = heap.rewrite(expression);
5   // apply substitutions
6   List<HeapReplacement> sub = heap.getSubstitution();
7   ValueDomain value = getValueState().applySubstitution(sub);
8   // let the value process the expressions
9   ValueDomain lub = value.bottom();
10  for (ValueExpression expr : exprs)
11    lub = lub.lub(value.smallStepSemantics(expr));
12  // return the updated state
13  return new SimpleAbstractState(heap, lub);
14 }

```

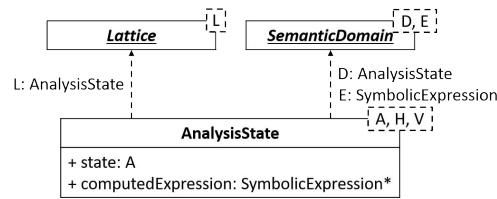


Figure 3.12: The AnalysisState hierarchy

Note that the choice of employing the framework of [61] is a key aspect for both **P4** and **P5**, as all three components involved in it are only responsible for an independent part of the overall computation. Furthermore, type inference is also run during the analysis, as a `TypeEnvironment` is also embedded into the *Abstract State*, enabling components that need runtime type information to access it. As such environment is handled exactly as `ValueDomain`, it has been left implicit in this section.

3.3.6 Analysis State

The *Analysis State* is the outer-most implementation of `SemanticDomain`, and it is thus the one explicitly visible to the rest of the analysis. A direct implication of this is that other components are agnostic w.r.t. how LiSA abstracts memory structures and values of the program, as advocated by **P4**. Class `AnalysisState` (Figure 3.12) implements both the `SemanticDomain<AnalysisState, SymbolicExpression>` and the `Lattice<AnalysisState>` interfaces, and is parametric to the concrete types `A`, `H`, and `V` of `AbstractState`, `HeapDomain`, and `ValueDomain` it contains, respectively. Its main duty is to wrap the *Abstract State* together with additional semantic information over which the analysis must reach a fixpoint. Here, we identify as *mandatory* information only the set of `SymbolicExpressions` that are left on the stack after evaluating an arbitrary expression, but more can be added. As a wrapper, operators from `SemanticDomain` delegate to the inner *Abstract State*.

3.4 Interprocedural Analysis

LiSA's *Interprocedural Analysis* is responsible for computing both a program-wide fixpoint and the result of calls, as the two features are strictly related. In fact, the computation of the overall program fixpoint is a direct consequence of call evaluation. If a pre-computed result is to be returned when a call is encountered, call-chains should be analyzed bottom-up starting from the target of the last call, thus ensuring that results are already available when needed. Instead, if results are to be freshly generated, call-chains should be analyzed top-down, starting from the *CFG* containing the first call.

`InterproceduralAnalysis` (Figure 3.13a) is the interface defining the operations that an *Interprocedural Analysis* must support. LiSA's analysis begins by invoking

<i>InterproceduralAnalysis</i>
+ fixpoint(in program: Program, in entryState: AnalysisState)
+ getResultOf(in CFGCall call, in entryState: AnalysisState, in params: SymbolicExpression*)
+ getResultOf(in OpenCall call, in entryState: AnalysisState, in params: SymbolicExpression*)
+ resolve(in call: UnresolvedCall, in types: Type*): Call

(a) The `InterproceduralAnalysis` interface

<i>CallGraph</i>
+ resolve(in call: UnresolvedCall, in types: Type*): Call

(b) The `CallGraph` interface**Figure 3.13:** `InterproceduralAnalysis` and `CallGraph`

ing `fixpoint`, passing the program to analyze and the `AnalysisState` that must be used as entry state for the analysis of *CFGs* that are reachable from outside the program (e.g., a `main` function). Such method must implement a strategy for analyzing the program, reaching a fixpoint on each of its *CFGs*. Individual *CFG* fixpoints are evaluated using Algorithm 2, uniquely implemented in `LISA`, that implements the classical worklist-based fixpoint over graphs. The algorithm takes as input the target *CFG*, an `AnalysisState` instance to use as pre-state for the starting nodes of the *CFG*, and a reference to the `InterproceduralAnalysis`, and yields the post-state computed for each `Statement` of the *CFG*. The algorithm exploits few supporting functions:

- `entries(cfg)`, returning the roots of the *CFG*;
- `preds(cfg, st)`, yielding the predecessors of a given node in the given *CFG*;
- `succ(cfg, st)`, that returns all the successors of a given node in the given *CFG*;
- `semantics(st, in, iproc)`, that computes the post-state of a node given its pre-state.

Specifically, `semantics(st, in, iproc)` corresponds to the language-specific `semantics` defined in `Statements`. As discussed in Section 3.2.2, non-calling `Statements` will rewrite themselves as a sequence of `SymbolicExpressions` and will feed them to the *Analysis State* to compute the post-state. However, when a call must be evaluated, the rewriting process is not enough. In fact, the `semantics` function can resort to its `InterproceduralAnalysis` parameter.

As shown in Figure 3.3a, four concrete instances of `Call` are included in `LISA`: `NativeCall`, `CFGCall`, `OpenCall`, and `UnresolvedCall`. `NativeCalls` only targets *native CFGs*: as such, the `semantics` of this class exploits the latter’s rewriting functionality to transform them into a `Statement` instance, and then delegates the computation to its `semantics`. `CFGCalls` instead invoke *CFGs* and thus have to access their targets’ fixpoint results. `InterproceduralAnalysis` provides this capability through the `getResultOf` overload accepting a `CFGCall`. The reasoning applied by this method depends on implementation-specific logic, and specifically

Algorithm 2: Fixpoint algorithm defined over a *CFG*

Data: *CFG* *cfg*, *AnalysisState* *entry*, *InterproceduralAnalysis* *iproced*
Result: *Statement* \rightarrow *AnalysisState* *result*

```

1 result  $\leftarrow$  { (st, entry) | st  $\in$  entries(cfg) };
2 ws  $\leftarrow$  entries(cfg);
3 while ws  $\neq$   $\emptyset$  do
4   st  $\leftarrow$  pop(ws);
5   in  $\leftarrow$   $\sqcup$ { result(m) | m  $\in$  preds(cfg, st) };
6   out  $\leftarrow$  semantics(st, in, iproced);
7   if not out  $\sqsubseteq$  result(st) then
8     if widening threshold reached then
9       result(st)  $\leftarrow$  result(st)  $\nabla$  out;
10    else
11      result(st)  $\leftarrow$  result(st)  $\sqcup$  out;
12    ws  $\leftarrow$  ws  $\cup$  succ(cfg, st);
13 return result;
```

on how call chains are analyzed. Instead, the `getResultOf`' overload accepting an `OpenCall` is used in the latter's semantics, letting the analysis compute an abstraction of an unknown result (for instance, one could conservatively assume that open calls can natively manipulate memory, thus always return \top as post-state). Lastly, `UnresolvedCalls` are calls that only carry signature information, and are not yet bound to their targets. Target resolution is performed by the *Call Graph*, that will be introduced in the following section, but a further degree of modularity (**P4**) is added by having the call interact with *InterproceduralAnalysis*' `resolve` method, as some implementations (i.e., intraprocedural ones) might return fixed over-approximations when calls are evaluated, bypassing the *Call Graph* invocation that otherwise happens here. Regardless, every instance of `UnresolvedCall` is converted to one of the other `Call` instances⁴, and their semantics can be normally applied.

The *Interprocedural Analysis* can be implemented, among other possibilities, as a context-sensitive [120, 84] analysis, following call-chains top-down evaluating each *CFG* fixpoint as they are called. An alternative approach is to adopt a modular [48] (also called *summary-based*) analysis, where call-chains are analyzed bottom-up accessing pre-computed results.

3.4.1 Call Graph

The *Call Graph* is tasked with resolving `UnresolvedCalls` to their targets. As such calls only come with signature information (i.e., the name of the target *CFG*) and their parameters, the whole program needs to be searched for possible targets. The search is a complex operation that relies on several features of programming languages, and

⁴`UnresolvedCalls` might resolve to both *CFGs* and *native CFGs*: here, LiSA instantiates a `MultiCall`, whose semantics yields the lub of the internal `CFGCall` and `NativeCalls`.

it can be logically split into two phases: scanning for possible targets along the program and matching the actual parameters to each candidate's signature.

Candidate scanning depends on the call type. If the call is known to have a receiver (that is, if it is an *instance* call), only the receiver's type hierarchy is to be searched for targets. Hierarchy traversal is language-specific, as it is influenced by how the language enables inheritance (e.g., it might be single or multiple, or it might provide explicit and implicit interface implementations). Instead, choosing the starting point for hierarchy traversal is a feature depending on the implementation of *Call Graph*: for instance, a graph implementing *Class Hierarchy analysis* [54] would consider all possible subtypes of the receiver's static type, while one implementing *Rapid Type analysis* [17] would restrict such set to the subtypes instantiated by the program. Regardless, the hierarchy of candidate types must then be searched for *CFGs* with a matching name. If the call instead does not have a receiver (i.e., if it is *static*), the whole program needs to be searched for *CFGs* with a matching name.

Once candidates have been selected, the actual parameters of the call must be matched with the formal ones defined by each target. Once more, this feature is language-specific: capabilities like optional parameters and named ones, as well as how types of each parameter are evaluated complicate the matching process to a point where no unique algorithm can be applied.

To make resolution parametric (**P2**), LiSA delegates language-specific call resolution features to each *UnresolvedCall* instance: when created, users need to specify both a strategy for traversing a type hierarchy given a starting point and a strategy for matching a list of *Expressions* to an arbitrary signature. Note that, once a call has been resolved, an entry state for the targets has to be prepared by assigning actual parameters to formal ones. As this process also follows parameter matching, the *Interprocedural Analysis* will defer this preparation to the same algorithm.

LiSA's *CallGraph* interface defines the entity responsible for providing the call resolution algorithm through its *resolve* method. Following **P4** and **P5**, the resolution process is uniquely implemented in an abstract class named *BaseCallGraph*, that takes care of invoking *UnresolvedCall*'s algorithms. Implementers then only have to specify which types should be considered as receiver types, thus implementing strategies such as *Class Hierarchy analysis* or *Rapid Type analysis*.

3.5 Frontends

Frontends are tasked with performing the first rewriting phase, translating a (possibly partial) original program into one made of *CFGs* that can be analyzed by LiSA. As mentioned at the beginning of Section 3.2, a component performing a translation is included in most static analyzers, as moving to a more convenient representation makes writing analyses simpler. LiSA's *frontends*, however, are more than just raw compilers: as the sole component with deep knowledge about the language they tar-

get, they must define `Statements` with their semantics, types, and language-specific algorithms that implement the execution model of the language.

Even if they might seem less relevant to the whole analysis process, writing and maintaining a complete *frontend* for a language is no easy endeavor. In fact, mature and widespread programming languages have very complex semantics to model, with features that might be ambiguous or not formally defined⁵. Moreover, each language has its own evolution, leading to different versions needing support. This not only translates to a higher number of instructions to model, but also to a variety of runtime environments (containing different libraries and software frameworks) whose semantics has to be taken into account for precise static analysis.

Writing a *frontend* usually begins with the code parsing logic. Whenever it is not possible to use official tools (e.g., by plugging into the compilation process), parser generators such as ANTLR⁶ can be used to create custom abstract syntax tree visitors. `Statement` instances for each instruction must be included as part of the *frontend* (potentially using common implementations provided by LiSA), each one providing its own semantics and bringing language-specific algorithms that are exploited during the analysis. Type inference is optional, as the one run by LiSA during the analysis can be exploited inside semantics functions. This means that constructs such as `+`, that in most languages have different semantics depending on the type of its operands, can be modeled by a single `Statement` instance. Modeling runtimes and libraries is achieved through *native CFGs* or SARL files, as will be shown in Section 3.6.

3.6 Modeling library behavior: SARL

In this section, we define a domain-specific language called SARL, that can be used to express the semantics of library code and how the execution model of a program may change when executed through a framework. This section is based on the published paper [65]. SARL has been developed before LiSA, and has thus been experimented using the Julia [123] static analyzer, targeting object-oriented languages (specifically JAVA and C#). The formalization is however independent from the analyzer, as all algorithms involved are defined as simple iterators over the program's syntax. SARL's extension to LiSA is thus straightforward, as (i) the *CFG* model presented in Section 3.2 is an extension of the object-oriented definitions used in SARL's formalization, and (ii) SARL can be applied to the program's structure before the analysis begins.

SARL (**S**tatic **A**nalysis **R**efining **L**anguage) is a domain-specific language that allows one to easily instruct a static analyzer about the execution model of a framework to *improve* the results in terms of both precision and soundness. SARL targets statically type-safe, object-oriented programming languages (C# and JAVA in particular).

⁵For instance, PYTHON does not have a formal specification of its semantics, while C admits syntactic constructs whose behavior is undefined.

⁶<https://www.antlr.org/>, with several well-tested grammars available at <https://github.com/antlr/grammars-v4>.

These languages offer a construct that allows to add metadata to object-oriented components, describing their characteristics. We will generically refer to it as *annotation*. SARL adopts annotations as the key mean to instruct a static analyzer about both the structure and the execution model of the application under analysis. The goal of SARL is to produce a set of rules, called *framework specification*, that describes the behavior of a framework or library. Such description can then be automatically applied to programs to produce a collection of annotations on them that the analyzer can interpret and exploit during the analysis (e.g., when building the call graph of the program or approximating the heap structure).

Since we want to apply SARL specifications and extract the annotations on the target application *before* the analysis starts, framework specifications must be evaluated syntactically on the analyzed application without any semantic knowledge of the program. At this stage, no information about the runtime types of the program's values is known, as well as the effective targets of method calls. Hence, the content of framework specifications needs to be designed using static types and call targets.

SARL has been interfaced with Julia, an abstract interpretation-based static analyzer of JAVA bytecode whose analyses consider a wide range of annotations. Born as a JAVA analyzer, Julia has been extended to analyze .NET (CIL) bytecode as well [62]. We applied SARL to model two popular .NET frameworks (Windows Forms⁷ and ASP.NET⁸), and we present, for each of these frameworks, the results of Julia's analyses on the 3 most popular GitHub repositories of projects relying on them. In particular, we study how the analysis improved in terms of precision and soundness when using the SARL specification w.r.t. the original Julia analysis. The experimental results show that for programs widely relying on a framework (Windows Forms) the improvement is dramatic (SARL specification removed between 35.3% and 74.5% of false alarms), while when only a small portion of a program exploits a framework (ASP.NET) the benefit is restricted to that portion (between 1.6% to 4.3%).

This section is structured as follows. Section 3.6 shows a first example of a SARL specification for ASP.NET. Section 3.6.1 describes Julia, and Section 3.6.2 introduces a second specification, targeting Windows Forms, and uses it to introduce SARL and its constructs. Finally, Section 3.6.3 reports the benefits obtained when applying SARL specifications to six applications using Windows Forms or ASP.NET.

Example SARL specification

ASP.NET is a Microsoft framework used to build web applications written in C#. It comes in various flavors, like WebForms and MVC. As usually happens with most web frameworks, applications written with ASP.NET have an execution model fairly different from the one of a standard application. In fact, a wide variety of methods are invoked from the external environment, such as page and graphical event han-

⁷<https://docs.microsoft.com/it-it/dotnet/framework/winforms/>.

⁸<https://www.asp.net/>.

```

1 rule: rte ".net"
2 rule: superclass HttpApplication
3 predicate: isControl = cls -> subtypeOf: "Control"
4 predicate: isNestedComponent = and(fld -> type satisfies isControl, fld -> definingClass satisfies
  isControl)
5 predicate: isEventHandler = and(mtd -> basicReturnType:: "void", and(mtd ->
  numberOfParameters:: 2, and(mtd -> hasParameter and(par -> index:: 0, par -> type::
  "Object")), mtd -> parameter and(par -> index:: 1, par -> type.subtypeOf: "EventArgs"))))
6 predicate: isWebViewExecute = and(mtd -> basicReturnType:: "void", mtd -> name:: "Execute")
7 predicate: isGetAppInstance = and(mtd -> returnType.subtypeOf: "HttpApplication", mtd ->
  name:: "get_ApplicationInstance")
8 specification: annotate mtd with EntryPoint if and(mtd -> definingClass satisfies isControl,
  satisfies isEventHandler)
9 specification: annotate fld with ExternallyRead, Injected if satisfies isNestedComponent
10 specification: annotate mtd with EntryPoint if and(mtd -> definingClass.subtypeOf:
  "WebPageExecutingBase", and(mtd -> numberOfParameters:: 0, or(satisfies
  isWebViewExecute, satisfies isGetAppInstance)))
11 specification: annotate mtd with EntryPoint if and(mtd -> definingClass::startsWith
  "__ASP.FastObjectFactory", and(mtd -> name::startsWith "Create_ASP_", and(mtd ->
  returnType:: "System.Object", mtd -> numberOfParameters:: 0)))

```

Figure 3.14: ASP.NET specification

dlers. These are usually not public, and since no explicit calls to these methods are found in the code, a static analyzer would consider them (as well as all other methods invoked directly or indirectly) not reachable, potentially leading to both false positives (e.g., warnings about unreachable event handlers) and negatives (e.g., missing warnings on the code implementing of the event handler). Moreover, graphical objects are usually stored in fields, enabling the runtime environment to access them for initialization. As these are usually never explicitly assigned within the application, a semantic static analyzer would consider them as always null, potentially producing both false positives (e.g., nullness alarms when the object stored in the field is dereferenced) and negatives (e.g., on the code of a branch of an if-then-else statement that is guarded by a nullness check on the field, thus considered as deadcode).

The SARL specification of Figure 3.14, where namespaces have been omitted for the sake of compactness, describes both these features (as well as marking few other methods as reachable), providing metadata (as annotations supported by Julia), in a concise and self-explanatory manner: it is applied when a .NET program contains at least one `HttpApplication` (lines 1 and 2), which is the base class for ASP.NET applications. Then, fields representing runtime-managed objects (subtypes of the `Control` class) are identified and marked (line 9) as externally read and written (that is, injected), while event handlers, web page creation factories, and other standard framework methods are considered as entry points (lines 10-11) for the analysis. By annotating methods as entry points, SARL instructs the analyzer (and the call graph constructor in particular) to consider them as externally called with arbitrary parameters. In this way, the analysis will produce alarms on the code directly or indirectly executed by the method (removing false negatives) and will remove warnings about unreachable event handlers (removing false positives). Similarly, once these fields are

annotated as externally written (injected), the analyzer will consider that they might be assigned with arbitrary values, thus removing the aforementioned false positives.

As one can see from this brief example, SARL allows one to easily specify to which programs the specification should be applied, a set of predicates (improving the readability and reusability of the specification), and a set of rules to annotate program components as well as libraries. This specification will later be used in Section 3.6.3 to refine the results of Julia on applications that use ASP.NET.

3.6.1 Julia

Julia's analyses are interprocedural (that is, they consider the flow of control and information from callers to callees and vice-versa) and abstract the heap (that is, they consider the flow of data through heap writes and reads). This is essential to perform semantic static analyses, such as information flow or sound nullness analysis.

In Julia, the model of the program under analysis is built by a so-called class analysis, that infers the possible runtime types of the variables and stack elements. Julia uses the one defined in [107], which has been shown to be a reasonable trade-off between precision and cost. The construction of such model of the program is called *extraction* in Julia, since methods are extracted and then analyzed only if they are actually called in the program. The extraction starts from a set of entry points, that, by default, are all the public methods of the analyzed application. However, users can specify other entry point modes as an input of the analysis, and in particular (i) only standard entry point methods (e.g., `main` and servlet methods), (ii) only explicit entries (that is, methods annotated as `@EntryPoint`), or (iii) all public and protected methods. For the sake of simplicity, in the rest of the section we consider only the default mode. Julia includes various static analyses (e.g., sound nullness analysis [122], taint analysis [60], and data-size analysis [126]), that are logically split into two groups: Basic, performing simpler and mostly local reasoning, and Advanced, offering deeper semantic analyses.

Being born as a JAVA analyzer, Julia acquired knowledge on widespread JAVA frameworks over the years. However, behaviors have been hardcoded throughout various analysis components, making it hard to understand and document which aspects of each framework have been covered and how. Instead, Julia has no hardcoded model for C# frameworks. Thus, our initial effort targets this area.

Annotations. As of version 2.7.0.3, Julia defines more than 70 annotations with various meanings. Some of them are used to provide context about how the application interacts with the external environment (e.g., `@EntryPoint` states that a method could be called from outside the program, while `@Injected` states that a field or a parameter could be written by an external source), while the majority of them are used to provide information to a specific checker (e.g., `@SqlTrusted` is used to instruct the Injection checker that untrusted data should not flow into that location

since it will end up in a database, while `@NonNull` states that a field or a method's return value are never `null`). Finally, `@SuppressJuliaWarnings` instructs Julia that a certain kind of warning should not be reported on the annotated component (either a field, method, constructor, class, method parameter, or local variable).

Each annotation has a different scope, and thus, a different impact on the analysis: `@EntryPoint` will be exploited during the construction of the call graph, but its effect will be propagated throughout the whole analysis (as additional reachable code will be considered); `@SqlTrusted` instead will only be used during the execution of the taint analysis-based checkers (the Injection checker is the most popular, but other ones exist). Thus, a *framework specification* could be logically split into sections, each one having effects on a different set of checkers.

3.6.2 The SARL Language

The goal of SARL is to allow a user to specify a set of rules (called *framework specification*) representing how the framework affects the runtime behavior of a program. Such specification will then be exploited by a static analyzer to improve its precision and soundness. In particular, we rely on annotations to pass this information. Therefore, these should be expressive enough to represent these runtime behaviors. Throughout this section, we need annotations to specify: (i) when a method might be called by the framework runtime (`@EntryPoint` in Julia), (ii) when a field is read or written by the framework runtime (`@ExternallyRead` and `@Injected` in Julia, respectively), (iii) when the warnings on a specific component (e.g., method or field) should be suppressed (`@SuppressJuliaWarnings` in Julia, `@SuppressWarnings` in JAVA), and (iv) properties related to specific analyses (e.g., `@AutoClosedResource` of the `CloseResource` analysis in Julia).

Building a *framework specification* can be achieved with two different approaches. One can acquire knowledge about the framework itself, understanding its model of execution and how it interacts with the application code. Then, the acquired knowledge needs to be converted into a SARL specification, by understanding how each framework feature may impact the various analysis modules. While this approach ensures that every peculiarity of the framework is taken into account, the number and heterogeneity of software frameworks makes it hard to achieve, since one should possess knowledge on both the analyzer *and* the framework. Another approach consists in iteratively analyzing software that relies on the target framework, inspecting the analysis results searching for evidence of the lack of framework knowledge by the analyzer (e.g., unreachable methods that are instead invoked by the framework, or no injection-related warning on a web application) and fixing them in the specification. This approach is highly dependent on how representative the software is in exploiting the framework's functionalities, but can nevertheless be a good starting point. Both SARL instances presented in this section were built with the latter approach.

Figure 3.15 depicts the overall architecture of our approach, and how SARL in-

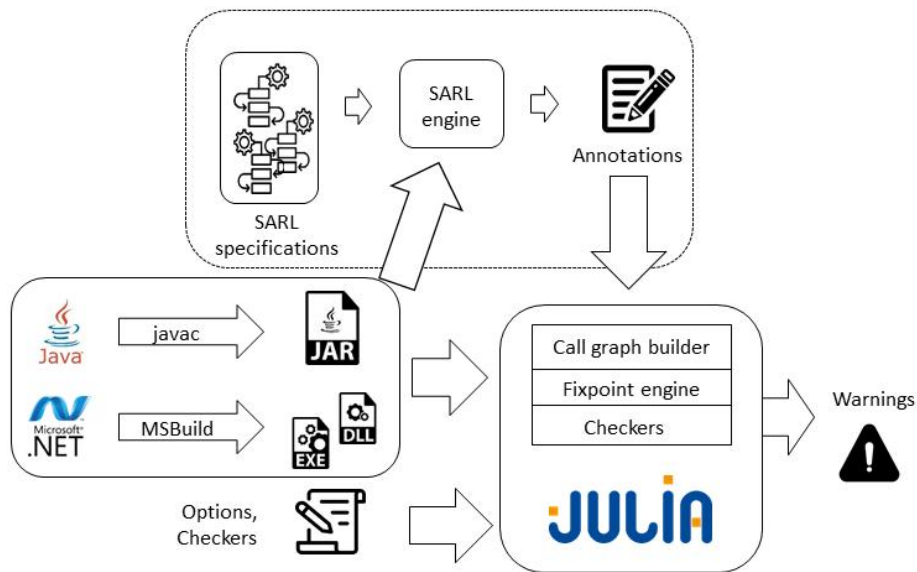


Figure 3.15: Schema of Julia's architecture with SARL

interfaces with the Julia static analyzer. Given a framework specification and an application to be analyzed, the SARL engine (represented inside the dotted rectangle) produces a set of annotations. These are then serialized to an XML file and passed together with all the other inputs of the analysis (analyzed code, analysis options, and checkers to run) to Julia. Further developments will bring the SARL engine inside the analyzer itself, making the identification of frameworks and the generation of extra annotations a fixed step of the analysis pipeline.

Example 3.6.1. Before discussing SARL formally, we introduce another SARL specification targeting Windows Forms, a framework to build GUIs of C# desktop applications. Usually, these are developed through the UI designer of Visual Studio, that places a pointer to each graphical component in private fields initialized by the generated code, causing the analyzer to raise a high number of warnings about field usage (stating that a field can be replaced by a local variable, or that the value written inside a field is never read later). Moreover, each graphical component in Windows Forms implements the *IDisposable* interface (which represents objects that should be disposed when no longer needed, since they could hold handles to non-managed resources that need to be manually released) and it is disposed by the framework runtime. Figure 3.16 reports the specification of Windows Forms (as in Figure 3.14, namespaces have been omitted for compactness), that will be used to explain SARL constructs.

Language Definition. SARL is built over five basic components: *rules*, *implications*, *specifications*, *predicates*, and *library specifications*. Rules embed information to

```

1 rule: rte ".net"
2 rule: superclass Form
3 predicate: isComponent = cls -> subtypeOf: "IComponent"
4 predicate: isDisposable = fld -> type.subtypeOf: "IDisposable"
5 predicate: isNestedComponent = and(fld -> type satisfies isComponent, fld -> definingClass
6 satisfies isComponent)
7 predicate: isGeneratedFormField = and(fld -> definingClass.subtypeOf: "ContainerControl",
8 and(fld -> hasAccessor: "private", and(fld -> name: "components", fld ->
9 type.subtypeOf: "IContainer")))
10 specification: annotate fld with ExternallyRead, Injected if satisfies isNestedComponent
11 specification: annotate fld with AutoClosedResource if or(fld -> type.subtypeOf:
12 "ContainerControl", and(satisfies isDisposable, fld -> definingClass satisfies
isComponent))
13 specification: annotate fld with NonNull, ExternallyRead, Injected if satisfies
"isGeneratedFormField";
14 library: annotate mtd with ResourceThatDoesNotNeedToBeClosed if cls Brushes mtd matches
15 "get_*(()LSystem/Drawing/Brush;"
16 library: annotate mtd with ResourceThatDoesNotNeedToBeClosed if cls Pens mtd matches
17 "get_*(()LSystem/Drawing/Pen;"
18 library: annotate mtd with ResourceThatDoesNotNeedToBeClosed if cls Process mtd
19 "GetCurrentProcess()LSystem/Diagnostics/Process;"

```

Figure 3.16: Windows Forms specification

detect if a given application relies on the framework. The specification is applied if and only if the conditions specified in the rules hold. Core components (implications, specifications, and predicates) allow one to define the conditions required to apply an annotation to a program member. Finally, library specifications allow the generation of annotations also on non-application classes (that is, classes that come from a supporting library or the system runtime).

Figure 3.17, where $n \in \mathbb{N}$ is a natural number and $\sigma \in \Sigma^*$ is an arbitrary string, defines the complete syntax of SARL, while each construct's semantics is formalized step-by-step when they are presented. During the formalization, we consider a program p as a set of classes; each class is a tuple $\langle n, A, F, C \rangle$ where n is the name of the class, while A , M , and F are the set of annotations, fields, and methods of the class, respectively. An annotation is a tuple $\langle n, \wp(n \times str) \rangle$, with n being a fully qualified name and $\wp(n \times str)$ being the set of members, represented as a pair of name and string value. A field is a tuple $\langle n, t, A, \wp(str) \rangle$, where n is the name, t is the type, A is the set of annotations, and $\wp(str)$ is the set of accessors. A method is a tuple $\langle n, t, A, \wp(str), P, V \rangle$, where n is the name, t is the return type, A is the set of annotations, $\wp(str)$ is the set of accessors, P is the set of parameters, and V is the set of local variables. A parameter is a tuple $\langle n, t, A, i \rangle$, with n being the name, t being the type, A being the set of annotations, and i being the index of the parameter. A variable instead is a pair $\langle n, t \rangle$ with n being the name and t being the type. Each element of the formalization could be subscripted with a letter stating if it refers to a class c , a field f , a method m , or a parameter p . The semantics relies on a set of standard operators over the different object-oriented program components. For the sake of simplicity, from now on we denote with X^* sequences or sets of X components, and with i_k

$v \in \text{VAL} ::= n \mid \sigma$	$c^* \in \text{COND}^* ::= (\text{op}^*)^? v$
$\text{id} \in \text{ID} ::= [a-zA-Z] \mid [_a-zA-Z]([_a-zA-Z0-9])^*$	$c^+ \in \text{COND}^+ ::= \text{op}^+ . c^+ \mid t^+ [\text{id}] :: c^+ \mid (t^+)^? :: c^+ \mid t^+ c^l$
$\text{qn} \in \text{QNAME} ::= \text{id} . \text{id}^*$	$c^l \in \text{COND}^l ::= \text{not}(c^>) \mid \text{and}(c^>, c^>) \mid \text{or}(c^>, c^>) \mid \text{satisfies id}$
$m \in \text{MEM} ::= \text{id} = \text{str}$	$c^> \in \text{COND}^> ::= t^+ \rightarrow \text{op}^+(\text{id})^?.c^+ \mid t^+ \rightarrow \text{op}^* v \mid c^l$
$a \in \text{ANN} ::= \text{qn} ((m, m)^*)^?$	$p \in \text{PRED} ::= \text{predicate: id} = c^>$
$\text{op}^* \in \text{OP}^* ::= \text{equals} \mid \text{contains} \mid \text{matches} \mid \text{startsWith} \mid \text{endsWith}$	$s \in \text{SPEC} ::= \text{specification: annotate tg with } a(a)^* \text{ if } c^>$
$\text{op}^+ \in \text{OP}^+ ::= \text{name} \mid \text{index} \mid \text{type} \mid \text{basicType} \mid \text{hasAnnotation} \mid \text{definingClass} \mid \text{returnType} \mid \text{basicReturnType} \mid \text{hasVariable} \mid \text{hasAccessor} \mid \text{hasOptionValue} \mid \text{hasParameter} \mid \text{subtypeOf} \mid \text{containsMethod} \mid \text{containsField} \mid \text{numberOfParameters} \mid \text{definingMethod}$	$l \in \text{LIB} ::= \text{library: annotate tg with } a(a)^* \text{ sig}$
$\text{op}^\triangleleft \in \text{OP}^\triangleleft ::= \text{equals} \mid \text{contains} \mid \text{startsWith} \mid \text{endsWith}$	$i \in \text{IMPL} ::= \text{implication: } a \text{ implies } a(a)^*$
$t^* \in \text{TYPE}^* ::= \text{str} \mid \text{int}$	$r^\circ \in \text{RULE}^\circ ::= \text{rte } (\text{op}^\triangleleft)^? \varepsilon \sigma \varepsilon$
$t^+ \in \text{TYPE}^+ ::= \text{cls} \mid \text{ann} \mid \text{par} \mid \text{var} \mid \text{mtd} \mid \text{fld}$	$r^\dagger \in \text{RULE}^\dagger ::= \text{annotation } (\text{op}^\triangleleft)^? \text{qn}$
$\text{tg} \in \text{TARGET} ::= \text{cls} \mid \text{fld} \mid \text{mtd} \mid \text{par}$	$r^\vee \in \text{RULE}^\vee ::= \text{superclass } (\text{op}^\triangleleft)^? \text{qn}$
$\text{sig} \in \text{SIG} ::= \text{cls qn (fld op}^* \varepsilon \sigma \varepsilon \mid \text{mtd op}^* \varepsilon \sigma \varepsilon (\text{par } n)^?)^?$	$r^\wedge \in \text{RULE}^\wedge ::= \text{uses type } (\text{op}^\triangleleft)^? \text{qn}$
	$r^\circ \in \text{RULE}^\circ ::= r^\circ$
	$r^\circ \in \text{RULE}^\circ ::= r^\dagger \mid r^\vee \mid r^\wedge$
	$r \in \text{RULE} ::= \text{rule: } (r^\circ \mid r^\circ)$
	$\text{spec} \in \text{SARL} ::= r^* (i \mid p \mid s \mid l)^*$

Figure 3.17: SARL syntax

element i of tuple k (e.g., n_m represents the name of method $m \in M$).

Rules. A rule r defines a condition to be satisfied to apply a specification. Rules express conditions on either the analysis r° , or the code r° . Analysis rule r° defines the runtime environment (e.g., .NET or JAVA) of the framework. Instead, code rules define what should be found inside the application to apply a specification, in particular identifying some specific types (either as supertype - r^\vee , or as type in a member signature - r^\wedge), or annotations from the library (r^\dagger). Rules semantics is defined though function hold as:

$$\text{hold}((r^{\circ*}, r^{\circ*}), p) \Leftrightarrow \begin{cases} r^{\circ*} = \emptyset \vee \exists r^\circ \in r^{\circ*} : \text{hold}(r^\circ, p) \\ \wedge \\ r^{\circ*} = \emptyset \vee \exists r^\circ \in r^{\circ*} : \text{hold}(r^\circ, p) \end{cases}$$

$$\text{hold}(r^\circ, p) \Leftrightarrow \text{holdString}(\text{extractRTE}(p) \text{ op}^\triangleleft \text{str})$$

$$\text{hold}(r^\dagger, p) \Leftrightarrow \exists n \in \text{extractAnn}(p) : \text{holdString}(n \text{ op}^\triangleleft \text{qn})$$

$$\begin{aligned} \text{hold}(r^\vee, p) &\Leftrightarrow \exists n \in p : n' \in \text{extOrImpl}(n) : \text{holdString}(n' \text{ op}^\triangleleft \text{qn}) \\ \text{hold}(r^\wedge, p) &\Leftrightarrow \exists n \in \text{extractType}(p) : \text{holdString}(n \text{ op}^\triangleleft \text{qn}) \end{aligned}$$

where `holdString` checks a condition over strings, `extractRTE` returns the runtime environment of the program, and `extractAnn`, `extOrImpl`, `extractType` extract the names of all annotations, inherited types, and used types, respectively.

Example 3.6.2. The first two lines of the specification of Windows Forms in Figure 3.16 define multiple `r` to be applied. In particular, this specifies to apply the framework to applications whose (i) runtime environment is set to `.net` (line 1), and (ii) at least one class inherits from (or implements) `Form` class (line 2).

Implications. An implication `i` specifies that an annotation `a` implies a set of other annotations. Then, if a program member is annotated with the former, it is automatically annotated with all of the latter. This can be useful when developers have used annotations from the libraries to get some functionalities in their code, and these annotations semantically imply some other annotations supported by the analyzer, or when a framework searches a program member through reflection by searching all annotated members. The semantics is defined through functions `impl` and `ann`:

$$\begin{aligned} \text{impl}(i, p) &= \bigcup_{(n_c, A_c, F, M) \in p} \left\{ \begin{array}{l} (n_c, A_c \cup \text{ann}(A_c, i), F', M') : \\ F' = \bigcup_{f \in F} (n_f, t_f, A_f \cup \text{ann}(A_f, i), C_f), \\ M' = \bigcup_{m \in M} \left\{ \begin{array}{l} (n_m, t_m, A_m \cup \text{ann}(A_m, i), C_m, P'_m, V_m) : \\ P'_m = \bigcup_{p \in P_m} (n_p, t_p, A_p \cup \text{ann}(A_p, i), i_p) \end{array} \right\} \end{array} \right\} \\ \text{ann}(A, a \text{ implies } a^*) &= A \cup \left\{ \begin{array}{l} \emptyset \quad \text{if } a \notin A \\ a^* \quad \text{if } a \in A \end{array} \right\} \end{aligned}$$

Example 3.6.3. A JAVA method annotated with JAX-RS's `@javax.ws.rs.GET` will eventually be called from the external environment to handle an HTTP GET request. Hence, such a method has to be considered an entry point of the analysis. Thus, relative to Julia, an implication between `@GET` and `@EntryPoint` is needed.

Predicates. A predicate `p` lets one assign an arbitrary name `id` to a condition `c` (later defined in this section), to avoid rewriting it multiple times. For example, one might define predicate `isGetter` whose condition identifies a getter method. Once it is defined, its name can be used instead of rewriting the actual condition.

Example 3.6.4. Line 5 of the Windows Forms specification in Figure 3.16 defines the `isNestedComponent` predicate. This holds if and only if the type of the given

field satisfies predicate `isComponent` (that is, it is a subtype of `IComponent` as defined at line 3), and the class defining the field satisfies `isComponent` as well (that is, it is a subtype of `IComponent`).

Specifications. This is the core component of SARL. A specification lets one specify a condition on a program member that, when satisfied, causes a set of annotations to be generated on that program member. Thus, all such members have to be iterated when evaluating a specification. This construct enables one to identify members depending on their structure, as well as that of their related members. This goes beyond the simple reflective access (e.g., the one offered by library specifications described below), allowing one to identify members in a very precise manner. A specification s consists of the type tg of program member we want to annotate, one or more annotations a , and a condition c^\wedge that states when these have to be applied. For example, when analyzing a Unity⁹ application, each `Start` method of classes that extend `UnityEngine.MonoBehaviour` should be considered as an entry point for the analysis, since such method will be called by the Unity engine to perform the setup of the component. This can be achieved using a specification that has `mtd` as target, contains `@EntryPoint` as annotation, and as condition the *and* of the two aforementioned conditions (method's name and parent class). The semantics is captured by function `spec`, where `typeOf` returns the type - class, field, method, or parameter - of a program component, and `chain` is the function that checks if a condition holds:

$$\text{spec}(s, p) = \bigcup_{(n_c, A_c, F, M) \in p} \left\{ \begin{array}{l} (n_c, A_c \cup \text{cond}(s, (n_c, A_c, F, M)), F', M') : \\ F' = \bigcup_{f \in F} (n_f, t_f, A_f \cup \text{cond}(s, f), C_f), \\ M' = \bigcup_{m \in M} \left\{ \begin{array}{l} (n_m, t_m, A_m \cup \text{cond}(s, m), C_m, P'_m, V_m) : \\ P'_m = \bigcup_{p \in P_m} (n_p, t_p, A_p \cup \text{cond}(s, p), i_p) \end{array} \right\} \end{array} \right\}$$

$$\text{cond}(s, pm) = \begin{cases} a^* & \text{if } \text{typeOf}(pm) = tg \wedge \text{chain}(pm, c^\wedge) \\ \emptyset & \text{otherwise} \end{cases}$$

Example 3.6.5. Line 9 of Windows Forms specification (Figure 3.16) specifies that all fields satisfying the `isNestedComponent` predicate should be annotated with `@ExternallyRead` and `@Injected`.

Conditions. c^\wedge may be (i) the application of a predicate (via its name - first case of c^l), (ii) a logical operator (*and*, *or*, *not* - remaining cases of c^l) applied to other conditions, (iii) a non-terminal operator followed by a further condition ($t^+ \rightarrow \text{op}^+(\text{[id]}^2.c^+)$), or (iv) a terminal operator followed by a constant value ($t^* \rightarrow \text{op}^* v$) where

⁹<https://unity3d.com/>.

`str` and `int` represent strings and integers, respectively. Conditions are grouped in chains, where operators are applied to navigate among the properties of program members (e.g., starting from a class, one could navigate to a parameter of one of its supertype's methods). The ability to navigate through program members enables the definition of syntactic conditions that correspond to how a framework might search for a program member to interact with, both by searching instances of particular types or by retrieving members annotated with a given framework annotation. The formalization of the check of these conditions is represented by function `chain` in specifications' semantics and left implicit for the sake of simplicity (mostly standard checks of standard OO properties). Notice that, if one omits an operator, the default one is applied depending on the program member that is currently under evaluation.

Library specifications. In object-oriented software, most of the code is contained in libraries providing standard features to the application. However, libraries contain code that could need SARL-generated annotations, since their methods or fields could require additional knowledge. However, library code is usually much bigger than the application code, and iterating over it would lead to a huge overhead. In this context, SARL does not provide complex conditions, but it simply allows one to check the signature of a program member and annotate it. Therefore, `l` consists of the type `tg` of program member we want to annotate, one or more annotations `a`, together with the signature `sig` of the target program member. When applied, this leads to adding the given annotations to all the program members whose signature fulfills the specified signature. Notice that, even if this component was specifically designed to operate on library code, it can be nonetheless used to annotate application code, avoiding the iteration on all program members by loading them through reflective calls. Functions `lib` and `addLA` capture the semantics of library annotations, where `checkSignature` checks if two signatures represent the same element, and `typeOf` returns the type - class, field, method, or parameter - of a program component:

$$\text{lib}(l, p) = \bigcup_{(n_c, A_c, F, M) \in p} \left\{ \begin{array}{l} (n_c, A_c \cup \text{addLA}(l, n_c), F', M') : \\ F' = \bigcup_{f \in F} (n_f, t_f, A_f \cup \text{addLA}(l, f), C_f), \\ M' = \bigcup_{m \in M} \left\{ \begin{array}{l} (n_m, t_m, A_m \cup \text{addLA}(l, m), C_m, P'_m, V_m) : \\ P'_m = \bigcup_{p \in P_m} (n_p, t_p, A_p \cup \text{addLA}(l, p), i_p) \end{array} \right\} \end{array} \right\}$$

$$\text{addLA}(l, pm) = \begin{cases} a^* & \text{if } \text{typeOf}(pm) = \text{tg} \wedge \text{checkSignature}(pm, \text{sig}) \\ \emptyset & \text{otherwise} \end{cases}$$

Example 3.6.6. Line 10 of Windows Forms specification (Figure 3.16) specifies to annotate with `@ResourceThatDoesNotNeedToBeClosed` all the getter methods

Algorithm 3: Application of a SARL specification to a program

```

Data: (app, lib), spec
Result: (app, lib)
1   $rte \leftarrow \{ r^\circ \mid r^\circ \in spec \};$ 
2   $code \leftarrow \{ r^\circ \mid r^\circ \in spec \};$ 
3  if hold((rte, code), app) then
4    for  $s \in spec$  do
5      |  $app \leftarrow spec(s, app);$ 
6    for  $l \in spec$  do
7      |  $app \leftarrow lib(l, app);$ 
8      |  $lib \leftarrow lib(l, lib);$ 
9    for  $i \in spec$  do
10   |  $app \leftarrow impl(i, app);$ 
11   |  $lib \leftarrow impl(i, lib);$ 
12 return (app, lib);

```

of class `Brushes` that return a `Brush` instance, since these are system-wide objects handled by the runtime, and therefore should not be manually closed by the program.

SARL application. Algorithm 3 reports the algorithm for applying a SARL specification to a program. In particular, given a specification `spec`, and a program composed of an application `app` and a library `lib` (both represented as a set of classes), it applies the specification if and only if the rules set r^* is satisfied on the application `app` (line 4). If this is the case, it then sequentially applies all the specifications `s` (lines 5-6), libraries `l` (lines 7-9), and implications `i` (lines 10-12) contained in the SARL specification `spec`. Note that, while specifications `s` are applied only to the application, library specifications `l` and implications `i` are applied to both the application and the library part of the program. In this way, SARL allows adding information about the libraries of the framework, and not only to model its effects on the application.

3.6.3 Experimental Results

SARL has been interfaced with the Julia static analyzer, version 2.7.0.3, as specified in Figure 3.15. The SARL specification parser relies on JavaCC¹⁰, while the semantics has been natively implemented in JAVA and passed to Julia through external (i.e., specified in an XML file rather than the application code) annotations.

In this section, we analyze, for both Windows Forms and ASP.NET, the 3 most popular applications publicly available in GitHub that rely on these frameworks. We adopt as a metric of the popularity of a repository its number of stars. For each application, we took the last stable release in the repository. All the statistics refer to the status of GitHub on June 28th, 2020. Each application has been analyzed with and

¹⁰<https://javacc.org/>.

Framework	Application	Version	LOCs	TW	TW/O
WinForms	Shadowsocks ¹²	4.1.6	12788	2'23"	2'14"
WinForms	ShareX ¹³	12.4.1	99191	5'24"	5'24"
WinForms	CefSharp ¹⁴	73.1.130	17863	2'01"	1'59"
ASP.NET	SignalR ¹⁵	2.4.1	49182	4'24"	4'36"
ASP.NET	AspnetBoilerplate ¹⁶	4.5.0	87288	6'15"	6'08"
ASP.NET	Umbraco ¹⁷	8.0.2	130384	11'09"	11'03"

Table 3.1: Analyzed applications

without the framework specification. We report as lines of code (LOC) the number of physical lines of code reported by Locmetrics¹¹ on the C# source files (with *cs* file extension) of the different applications in GitHub. Therefore, we consider only the code of the application and not the libraries. For each application, we compared the results of the analyses with and without SARL, investigating the number of warnings added or removed by the latter analysis. Each of such warnings has been manually investigated to ensure that no true positives were lost with our approach, and that new warnings can be accounted on the introduction of new entry points causing the analysis of previously unreachable code.

Table 3.1 reports the applications we selected, where column **Framework** reports the framework it uses (here, WinForms is a shorthand for Windows Forms), **Application** is the name of the analyzed application, **Version** is the analyzed version (taken from GitHub, thus directly associated with a commit that can be used for reproducibility), **TW** and **TW/O** the analysis time with and without the application of the framework specification (considering also the time needed for applying the specification to the application), respectively¹⁸. All the analyses were executed on an *r5.xlarge* Amazon Web Services machine. These instances featured a Xeon Platinum 8000 series (Skylake-SP) processor with a sustained all-core Turbo CPU clock speed of up to 3.1 GHz and 32 GB of RAM.

Windows Forms

Table 3.2 reports, for each application, the number of warnings with Basic checkers without the SARL specification, and the number of common, added, and removed warnings when applying the specifications presented in Figure 3.16 of Section 3.6.2. The results highlight that even small specifications can have a major impact on the results of the analyses, removing a huge portion of false alarms issued due to the

¹¹<https://www.cheonghyun.com/blog/120>.

¹²<https://github.com/shadowsocks/shadowsocks-windows>.

¹³<https://github.com/ShareX/ShareX>.

¹⁴<https://github.com/cefsharp/CefSharp>.

¹⁵<https://github.com/SignalR/SignalR>.

¹⁶<https://github.com/aspnetboilerplate/aspnetboilerplate>.

¹⁷<https://github.com/umbraco/Umbraco-CMS>.

¹⁸Stars and ranking of each project have been omitted here, but can be found in the original paper.

	Shadow.	ShareX	CefSharp
Warn. w/o spec.	730	5471	465
Common (%)	473 (64.8%)	1397 (25.5%)	241 (51.8%)
Added (%)	0 (0%)	6 (0.1%)	0 (0%)
Removed (%)	257 (35.2%)	4074 (74.5%)	224 (48.2%)

Table 3.2: Difference in warnings on Windows Forms analyses

Warning	SS		SX		CS	
	A	R	A	R	A	R
ResourceNotClosedAtEndOfMethod	0	8	0	98	0	32
CloseableNotStoredIntoLocal	0	202	0	2802	0	124
FieldShouldBeReplacedByLocals	0	32	0	958	0	50
FieldsOnlyUsedInConstructors	0	0	0	1	0	0
UselessAssignmentToDefaultValue	0	7	0	83	0	6
TestIsPredetermined	0	4	0	66	0	6
UnreachableInstruction	0	4	0	66	0	6
SetStaticInNonStaticWarning	0	0	6	0	0	0

Table 3.3: Warnings removed on Windows Forms applications

lack of framework knowledge by the analyzer. We will focus on the removed warnings only since the 6 ones added in ShareX analysis are all real alarms residing in methods that were previously considered deadcode and thus not analyzed.

Table 3.3 reports the warnings removed (**R**) and added (**A**) when applying the Windows Forms specification to Shadowsocks (**SS**), ShareX (**SX**), and CefSharp (**CS**), grouped by type. The specification targeted mostly disposable objects stored into fields: the majority of the warnings (4403 out of 4556, that is, 96%) refer to them.

Warnings about closable resources, such as *CloseableNotStoredIntoLocal* and *ResourceNotClosedAtEndOfMethod* are issued whenever objects inheriting from *Closeable* (Java) or *IDisposable* (C#) might not get closed/disposed: such an object should be stored in (i) a `final/readonly` field on which a call to *close()/Dispose()* happens in reachable code, or (ii) a local variable on which a call to *close()/Dispose()* happens before the end of the method. Figure 3.18 shows a snippet of code from ShareX. In method `UpdateTrayMenu`, a new `ToolStripMenuItem` is created and added to another `ToolStripMenuItem` retrieved from a field of `MainForm`. Since (i) class `ToolStripMenuItem` implements `IComponent`, (ii) `MainForm` inherits from `Form` which implements `IComponent`, and (iii) the newly created `ToolStripMenuItem` will be reachable from `MainForm` after the execution of the if-else block, such object will be automatically disposed from the runtime environment when the instance of `MainForm` will be disposed by the Windows Forms runtime. When analyzing ShareX without the Windows Forms specification, Julia raises the following warning:

CloseResource: This instance of class `ToolStripMenuItem` does not seem to be always closed by the end of this method. It seems leaked at line 156 [*at Recent-*

```

1 class MainForm : Form {
2     ToolStripMenuItem tsmiTrayRecentItems;
3 }
4 class RecentTaskManager {
5     void UpdateTrayMenu() {
6         ToolStripMenuItem tsmi = MainForm.tsmiTrayRecentItems;
7         ToolStripMenuItem tsmiLink = new ToolStripMenuItem();
8         if (...) tsmi.DropDownItems.Insert(2, tsmiLink);
9         else tsmi.DropDownItems.Add(tsmiLink);
10    }
11 }

```

Figure 3.18: Disposable objects stored in fields of classes

```

1 class StatisticsStrategyConfigurationForm : Form {
2     Button OKButton;
3     void InitializeComponent() {
4         OKButton = Button();
5         // init code
6         splitContainer1.Panel2.Controls.Add(OKButton);
7     }
8 }

```

Figure 3.19: UI fields generated by Visual Studio

TaskManager.cs:156]

When we apply the Windows Forms specification in Figure 3.16 to this code, line 8 annotates field `tsmiTrayRecentItems` as `@AutoClosedResource`, and this informs Julia that every resource reachable from that field will be automatically disposed. Hence, the above warning is no longer issued, as the newly created `ToolStripMenuItem` `tsmiLink` will end up being reachable from such field.

Warnings `FieldShouldBeReplacedByLocals` and `FieldsOnlyUsedInConstructors` are instead issued when a field could be replaced by a local variable inside the *only* method (`FieldShouldBeReplacedByLocals`) or constructor (`FieldsOnlyUsedInConstructors`) that references them. Figure 3.19 shows a simplification of code generated by Windows Forms to represent a UI component in Shadowsocks. Field `OKButton` is initialized and added to a container inside `InitializeComponent()`, but the field is never used later in the code. The initialization is inside the Visual Studio designer-generated file, and the user has no control over this. Julia raises the following warning on this code:

ImproperField: Field `OKButton` should be replaced by local variables [*at StatisticsStrategyConfigurationForm.cs*]

When we apply the Windows Forms specification in Figure 3.16 to this snippet of code, line 7 annotates field `OKButton` as `@ExternallyRead` and `@Injected`, and Julia does not produce the above warning anymore.

Finally, Julia's analysis is able to detect when a test always evaluates to true or false. In this situation, two warnings are issued: a `TestIsPredetermined` warning stating that the test is useless since it always evaluates to the same Boolean value, and a `UnreachableInstruction` warning to explicitly mark the unreachable branch (if this

```

1 class BrowserTabUserControl {
2     IContainer components = null;
3     override void Dispose(bool disposing) {
4         if (disposing)
5             if (components != null) {
6                 components.Dispose();
7                 components = null;
8             }
9         base.Dispose(disposing);
10    }
11 }

```

Figure 3.20: Dispose() pattern of Form classes

	SignalR	ANB	Umbraco
Warn. w/o spec.	681	552	1729
Common (%)	658 (96.6%)	544 (98.6%)	1658 (95.9%)
Added (%)	1 (0.1%)	0 (0%)	0 (0%)
Removed (%)	23 (3.4%)	8 (1.4%)	71 (4.1%)

Table 3.4: Difference in warnings on ASP.NET analyses

contains some code). In addition, when a field or a local variable gets initialized with its default value Julia raises an *UselessAssignmentToDefaultValue* warning. Figure 3.20 shows a pattern generated by Visual Studio for handling the disposal of resources of Form classes in CefSharp. The `components` field is non-null only if the form contains some resources that are not UI objects but needs to be disposed (e.g., a `Timer` instance). However, the field is initialized and managed by the framework runtime, and the code of the application never assigns it. The code of the designer declares the field, initializes it to `null`, and then disposes it if it is not `null`. On this piece of code, Julia raises the following three warnings:

UselessAssignment: Useless assignment of field `components` to its default value
[at *BrowserTabUserControl.cs*:8]
UselessTest: The result of this test is fixed: you are comparing `null` against `null`
[at *BrowserTabUserControl.cs*:103]
Deadcode: This instruction seems unreachable [at *BrowserTabUserControl.cs*:105]

When we apply the Windows Forms specification in Figure 3.16 to this snippet of code, line 9 annotates field `components` as externally injected (as it effectively happens in the framework runtime), and therefore Julia analyses do not produce anymore these warnings.

ASP.NET

Table 3.4 reports, for each analyzed application, the number of warnings with a standard analysis (without the SARL specification) executing Basic checkers altogether, and the number of common, added, and removed warnings when performing the same analyses with the specifications presented in Figure 3.14 of Section 3.6 (ANB

Warning	SR		AB		UM	
	A	R	A	R	A	R
FieldNeverUsed	0	6	0	0	0	1
FieldReadWritten	0	0	2	0	0	0
Uncalled	0	17	0	10	0	70
PossibleInsecureCookieCreation	1	0	0	0	0	0

Table 3.5: Warnings removed on ASP.NET applications

```

1 abstract class AbpWebApplication : HttpApplication {
2     void Application_Start(object sender, EventArgs evargs) {
3         // startup code
4     }
5 }
6 class MvcApplication : UmbracoApplicationBase { }

```

Figure 3.21: Application_Start method

is a shortcut for AspnetBoilerplate). It is noticeable that the results on ASP.NET applications are less pervasive than the ones on desktop applications: this is due to the nature of those projects that are libraries (SignalR and AspnetBoilerplate) or content providers (Umbraco), and they contain very few web pages based on ASP.NET.

Table 3.5 reports the warnings removed (**R**) and added (**A**) when applying the ASP.NET specification to SignalR (**SR**), AspnetBoilerplate (**AB**), and Umbraco (**UM**) grouped by warning type. As for Windows Forms results, we will not discuss the added warnings since they are true alarms on methods that were previously considered dead code.

Julia issues an *Uncalled* warning on each method that is not reachable from the entry points of the application, and the code from these methods is never analyzed. Figure 3.21 shows an `Application_Start` method in AspnetBoilerplate. While this is never actually used within the application code, it is indeed invoked by the framework runtime at the first startup of `MvcApplication`. Hence, even if its access is restricted (i.e., it is not public), it must be considered an entry point. Besides, the compilation of the web views of the application results in an assembly containing one class file per view (named `ASP._Page_namespace_.viewName`) with only a constructor, a getter for the current application instance and an `Execute` method, and an object factory (named `__ASP.FastObjectFactory_.applicationName`) that is used by ASP.NET to instantiate the web views. These methods are both (i) generated code, and (ii) invoked by the runtime. When analyzing Umbraco, Julia raises the below warnings:

Deadcode: Method `Application_Start` is unreachable [at `UmbracoApplicationBase.cs:72`]

Deadcode: Method `Create_ASP__Page_Umbraco_Install_Views_Index_cshtml` is unreachable [at `__ASP.FastObjectFactory_umbraco.cs`]

Deadcode: Method `get_ApplicationInstance` is unreachable [at `ASP._Page-`

```

1 class _Default : Page {
2     TextBox userName;
3     TextBox roles;
4     Button login;
5     void Login(object sender, EventArgs e) {
6         var identity = new GenericIdentity(userName.Text);
7         var principal = new GenericPrincipal(identity,
8             SplitString(roles.Text));
9     }
10 }

```

Figure 3.22: UI fields generated by Visual Studio

`[_Umbraco_Install_Views_Index_cshtml.cs]`

When we apply the ASP.NET specification in Figure 3.14 to this snippet of code, line 8 annotates `Application_Start` as an entry point, while lines 10 and 11 do the same for the latter methods, removing all three warnings from the results.

A *FieldNeverUsed* warning is issued whenever a field is never read or written inside the whole reachable application code. Figure 3.22 shows a snippet of ASP.NET code from SignalR. Method `Login` is an event handler, and hence never explicitly called in the reachable code. In addition, fields `userName` and `roles` are never explicitly initialized: they represent an alias for the web view component declared in the `cshtml` file, and their values will be injected from the runtime environment. When analyzing SignalR without the SARL specification, Julia raises the following warnings:

```

FieldAccess: Field userName is never used [at _Default.cs]
FieldAccess: Field roles is never used [at _Default.cs]
FieldAccess: Field login is never used [at _Default.cs]
Deadcode: Method Login is unreachable [at _Default.cs]

```

When we apply the ASP.NET specification in Figure 3.14 to this code, line 8 annotates method `Login` as an entry point, while line 9 annotates fields `userName`, `roles`, and `login` as externally read and injected, thus removing the four warnings reported above.

3.7 Multilanguage analysis

We now instantiate LISA and its components to showcase how multilanguage analyses can be easily performed. We demonstrate the effectiveness of our approach on the JoyCar IoT system of Figure 1.1 (Section 1.2.1). Code snippets reported in this section are available on GitHub¹⁹, where the full implementation of this analysis is published.

As the codebase is composed of two languages, a *frontend* for each has to be built. These have been developed using ANTLR for parser generation, and mostly

¹⁹<https://github.com/lisa-analyzer/lisa-joycar-example>.

exploit `Statements` and `Edges` provided out-of-the-box from LiSA. The key aspect w.r.t. multilanguage analysis is the handling of constructs that enable inter-language communication, offered here by the *Java Native Interface* (JNI). At runtime, the JAVA VM tries to resolve calls to native methods using the name-mangling scheme reported in the JNI specification²⁰. We thus proceeded by providing an implementation for native methods found in JAVA code using the following (simplified) snippet:

```
1 void parseAsNative(CFG cfg, String className, String name,
2   Parameter[] formals, Type returnType) {
3   String mangled = nameMangling(className, name, formals);
4   Expression[] args = buildArguments(formals);
5   UnresolvedCall call = new UnresolvedCall(mangled, args);
6   if (!returnType.isVoidType())
7     cfg.addNode(new Return(call));
8   else {
9     Ret ret = new Ret();
10    cfg.addNode(call);
11    cfg.addNode(ret);
12    cfg.addEdge(new SequentialEdge(call, ret));
13  }
14 }
```

The code above bridges the two codebases by creating an `UnresolvedCall`, where (i) the target's name is built with the mangling scheme from the specification, (ii) the arguments for the call correspond to the ones passed to the native method preceded by the pointer to an instance of `JNIEnv` (an object required by JNI to hold pointers to native functions), and (iii) the value returned by the call is also returned by the native method, if any. With this setup, not only can the C++ code be parsed regularly, but the analysis components are also agnostic to the presence of JNI, as the call to the native method is treated exactly as any other call. Note that, while this specific example did not require it, the generated call can be preceded by arbitrary instrumentations (e.g., the state conversion typical of *boundary functions*).

The next step is to select the analysis components. We mostly rely on analyses natively provided by LiSA:

- the *Interprocedural Analysis* is set to a context-sensitive implementation that follows call-chains top-down, thus starting from the `main` method and traversing them until a recursion is encountered (and thus enabling LiSA to follow *every* call in our target application);
- the *Call Graph* implementation uses inferred runtime types of variables and expressions;
- the *Abstract State* used is `SimpleAbstractState`;
- as the program properties do not rely on dynamic memory, we use a fast but imprecise *Heap Domain* called `MonolithicHeap`, that abstracts each memory location to a unique synthetic one.

²⁰<https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/design.html>, paragraph "Resolving Native Method Names".

```

1 public class Taint extends BaseNonRelationalValueDomain<Taint> {
2     static final Annotation TAIN_T_ANNOT = new Annotation("lisa.taint.Tainted");
3     static final Annotation CLEAN_ANNOT = new Annotation("lisa.taint.Clean");
4
5     static final Taint TAIN_TED = new Taint(true);
6     static final Taint CLEAN = new Taint(false);
7     static final Taint BOTIOM = new Taint(null);
8
9     final Boolean taint;
10    Taint(Boolean taint) {
11        this.taint = taint;
12    }
13
14    Taint evalIdentifier(Identifier id, ValueEnvironment<Taint> env) {
15        Annotations annots = id.getAnnotations();
16        if (annots.contains(TAIN_T_ANNOT))
17            return TAIN_TED;
18        if (annots.contains(CLEAN_ANNOT))
19            return CLEAN;
20        return env.getState(id);
21    }
22
23    Taint evalConstant() {
24        return CLEAN;
25    }
26
27    Taint evalUnaryExpression(UnaryOperator operator, Taint arg) {
28        return arg;
29    }
30
31    Taint evalBinaryExpression(BinaryOperator operator,
32        Taint left, Taint right) {
33        return left.lub(right);
34    }
35
36    Taint evalTernaryExpression(TernaryOperator operator,
37        Taint left, Taint middle, Taint right) {
38        return left.lub(middle).lub(right);
39    }
40 }

```

Figure 3.23: A simple Taint Analysis implementation

As *Value Domain*, we implemented a Taint analysis [133, 60] as a *Non-Relational Domain*, whose simplified source code is reported in Figure 3.23. The domain is based on the poset $\{\{\perp, \text{clean}, \text{tainted}\}, \{\perp, \text{clean}\}, (\text{clean}, \text{tainted})\}$, that forms a finite (and thus complete) lattice using trivial \sqcup and \sqcap operators that, given a pair of elements, return the greater and smaller of the two, respectively. Implementation-wise, the superclass `BaseNonRelationalValueDomain` handles base cases of lattice operations, that is, when one of the operands involved is either \top (*tainted*) or \perp , or when the two operands are the same element. Hence, no additional logic needs to be implemented for \sqcup , \sqcap and \sqsubseteq . Recursive expression evaluation is also provided out-of-the-box by `BaseNonRelationalValueDomain`, and the concrete implementation only has to provide evaluation of individual expressions. Specifically, we consider all constants as *clean*, while evaluation of unary, binary, and ternary expressions is carried out by computing the \sqcup of their arguments. Tainted values are generated only when evaluating variables, relying on their annotations: as LISA assigns the result of `calls` to temporary variables, transferring all annotations from the call's targets, this enables uniform identification of both sources (i.e., annotated with `TAIN_T_ANNOT`) and

sanitizers (i.e., annotated with `CLEAN_ANNOT`). Variables are thus considered always *tainted* or always *clean* relying on such annotations, defaulting otherwise to the abstraction stored inside the environment.

To exploit our analysis results, we defined a *Check* instance that iterates over the application under analysis to scan for method parameters that are annotated with `@lisa.taint.Sink`, a third kind of annotation that identifies places where tainted information should not flow. When one such parameter is found, the check inspects all call sites where the corresponding method is invoked, and checks if the post-state of the `Expression` passed for the annotated parameter contains a tainted expression on the stack, according to our `Taint` domain. If it is, a warning is issued. We then proceed by annotating as source (i.e., with `@lisa.taint.Tainted`) the value returned by `readAnalog`, and as sink (i.e., with `@lisa.taint.Sink`) the second parameter of `softPwmWrite`. The analysis can then be executed, obtaining the following warning on the `softPwmWrite` call:

```
The value passed for the 2nd parameter of this call is tainted, and it reaches the sink at parameter 'value' of softPwmWrite [at JoyCar.cpp:124:55]
```

thus showcasing that cross-language vulnerabilities can be discovered in a single analysis run. Also note that, with the same setup, domains computing complex structures (e.g., automata) can still operate cross-language without incurring in expensive serializations and deserializations needed to communicate information across analyzers.

3.8 LiSA for teaching

In this section, we report our experience in using LiSA to let master level students experiment with static analysis. Several techniques are often taught in bachelor or master courses, in response to the increasing need for software verification skills. However, they require a relevant theoretical background and several preliminary notions that must be taught to students before they can move on to the implementation aspects that are equally challenging. Based on our teaching experience²¹, the theoretical background and notions needed to teach static analysis by abstract interpretation take most of the available time, allowing professors to only show few popular abstractions (e.g., `Sign` and `Interval` domains). Hence, issues related to the actual design of new analyses and the experimental evaluation of the trade-off between accuracy and computational cost of the analysis on different domains risk being neglected. This also limits the involvement of brilliant students in this research area. We present here the implementation of some simple value analyses that are usually formalized and taught in static analysis courses. In particular, we present the

²¹Prof. Cortesi and the Software and System Verification group at Ca' Foscari University (<https://ssv.dais.unive.it/>) taught a course on static analysis in the master in Computer Science during the last 2 decades. Full details at <https://www.unive.it/data/course/332756>.

Available Expressions dataflow analysis and the Sign abstract domain. These are just two possible instances of the LiSA infrastructure, while slightly different analyses (e.g., Constant Propagation, Intervals, ...) might easily be implemented in a similar way. The reported implementations can be easily employed as guiding examples to enable students to experiment with practical static analysis. This section is based on the published paper [66].

How to Run LiSA. The following fragment shows how to run LiSA using the IMP frontend (a frontend for a toy imperative language that is used in LiSA for testing purposes) and how to obtain the analysis results that we will show in this section.

```

1 Program program = IMPFrontend.processFile(filePath);
2 LiSAConfiguration conf = new LiSAConfiguration();
3 conf.setAbstractState(getDefaultFor(AbstractState.class,
4   new MonolithicHeap(),
5   new Sign()));
6 conf.setDumpAnalysis(GraphType.DOT);
7 conf.setWorkdir(outDir);
8 LiSA lisa = new LiSA(conf);
9 lisa.run(program);

```

The first line invokes the IMP front-end in order to process the IMP program located at `filePath`, returning a LiSA program that contains the CFGs corresponding to the functions contained in the source program. Line 2 creates a configuration for LiSA and the following lines proceed to initialize it: lines 3-5 set the *Abstract State*, and in turn the *Heap Domain* and *Value Domain*. In this case, we use the default implementation for the abstract state but we set the `MonolithicHeap` used also in Section 3.7 and the `Sign` abstractions for *Heap* and *Value Domains*, respectively. Then, line 6 tells to LiSA to dump the analysis results in dot format inside the output directory `outDir` specified at line 7. Finally, a LiSA instance is created at line 8 starting from the configuration, and the analysis is executed at line 9 on the program.

Dataflow Analyses. As discussed in Section 3.3.3, LiSA provides suitable interfaces for dataflow analyses, both possible and definite. In the following, we show the LiSA interface for forward and definite dataflow analyses. Dually, LiSA provides the interface for possible and forward dataflow analyses. These are parametric to an implementation of interface `DataflowElement` that must provide the classical `kill` and `gen` methods of the dataflow analyses. Let us focus on the `assign` method:

```

1 class DataflowDomain<
2   D extends DataflowDomain<D, E>,
3   E extends DataflowElement<D, E>>
4   extends BaseLattice<D> implements ValueDomain<D> {
5
6   Set<E> elements;
7   E domain;
8   D assign(Identifier id, ValueExpression exp) {
9     Set<E> updated = new HashSet<>(elements);
10    for (E killed : domain.kill(id, exp, (D) this))
11      updated.remove(killed);

```

```

12     for (E generated : domain.gen(id, exp, (D) this))
13         updated.add(generated);
14     return mk(domain, updated);
15 }
16 }

```

The implementation provided by LiSA implements the generic dataflow equation $out = in \setminus kill_i(in) \cup gen_i(in)$: first, the `kill` method is applied to the current instance to remove the killed dataflow elements (lines 10-11), followed by the application of the `gen` method to add the dataflow elements that are generated by the assignment (lines 12-13). In this way, the effects of the `gen` and `kill` methods are internally handled by the above class and they do not need to be implemented by the specific instance of dataflow analysis. Indeed, it is enough for the dataflow element concrete implementation `E` to implement the `kill` and `gen` methods. For instance, the class `AvailableExps` is just a few lines of code, as reported below²² (`getIdentifiers` is a helper function that returns all variables appearing in the given expression).

```

1 class AvailableExps implements DataflowElement<
2     DefiniteDataflowDomain<AvailableExpressions>,
3     AvailableExpressions> {
4
5     SymbolicExpression exp;
6     Collection<AvailableExps> gen(Identifier id, ValueExpression exp,
7         DefiniteDataflowDomain<AvailableExps> domain) {
8         Collection<AvailableExps> result = new HashSet<>();
9         result.add(new AvailableExps(exp));
10        return result;
11    }
12
13    Collection<Identifier> kill(Identifier id, ValueExpression exp,
14        DefiniteDataflowDomain<AvailableExps> domain) {
15        Collection<AvailableExps> result = new HashSet<>();
16        for (AvailableExpressions ae : domain.elements)
17            if (getIdentifiers(ae.exp).contains(id))
18                result.add(ae);
19        return result;
20    }
21 }

```

To perform an analysis, it is enough to feed `DefiniteDataflowDomain` with `AvailableExps`, set it as *Value Domain*, and run LiSA as we have shown at the beginning of this section. Figure 3.24b depicts the analysis results for a minimal example (Figure 3.24a), where each node reports the computed available expressions²³.

Non-Relational Abstract Domains. One of the first numerical analyses that are usually introduced in static analysis courses is the `Sign` analysis, tracking if each variable is zero, positive or negative. LiSA offers the `BaseNonRelationalValueDomain` to easily develop *Non-Relational Domains* (Section 3.3.3), requiring the concrete class to only implement *symbolic expressions* evaluation methods on non-bottom elements,

²²Here, we report only the `kill/gen` overloads for assignments, as the others can be simply derived by these.

²³The LiSA implementation has an internal optimization that avoids generating elements for constants.

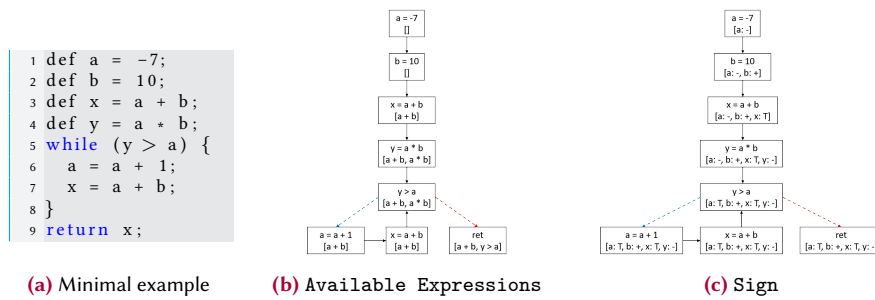


Figure 3.24: Analysis results on a minimal example

together with Lattice operations. In the following, we report the `Sign` class with an excerpt of its `evalBinaryExpression` method, a callback that is invoked by `BaseNonRelationalValueDomain` to compute an abstract value for a binary expression once its arguments have been evaluated. Similarly, it implements the evaluation methods for the other *symbolic expressions* types (e.g., unary, ternary, constant, ...) that are similar to the one presented here.

```

1 class Sign extends BaseNonRelationalValueDomain<Sign> {
2   static final Sign POS = new Sign();
3   static final Sign NEG = new Sign();
4   static final Sign ZERO = new Sign();
5   static final Sign TOP = new Sign();
6   static final Sign BOTTOM = new Sign();
7   ...
8   Sign evalBinaryExpression(BinaryOp op, Sign left, Sign right) {
9     switch (op) {
10      ...
11      case NUMERIC_ADD:
12        if (left == ZERO) return right;
13        else if (right == ZERO) return left;
14        else if (left == right) return left;
15        else return TOP;
16      ...
17    }
18  }
19 }

```

Lines 2-6 define the five abstract points that compose the `Sign` lattice. Then, at lines 10-17, method `evalBinaryExpression` switches on the binary operator `op`, defining the corresponding sign abstract semantics. We report only the case for the binary operator `NUMERIC_ADD`, implementing the classical signs rules.

At this point, we can feed `ValueEnvironment` with `Sign` and run `LISA`. The analysis results for the minimal example of Figure 3.24a are reported in Figure 3.24c.

3.9 Conclusion

In this chapter, we thoroughly described `LISA`, a modular framework for multilanguage static analysis with an open-source `JAVA` implementation. `LISA` operates by analyzing an extensible language of *CFGs*, whose nodes contain user-defined language-

specific semantics that translates them into *symbolic expressions*. These are atomic constructs with precise semantics, that abstract domains can analyze. LiSA's infrastructure modularly decomposes semantics evaluation into separate tasks, each carried out by a different analysis component. Each such component performs agnostically w.r.t. the concrete implementations of other ones, and is responsible for abstracting specific program features. The *Interprocedural Analysis*, in cooperation with the *Call Graph*, abstracts calls and call-chains, leaving the rest of the analysis with call-free programs. Then, the modular *Analysis State* orchestrates memory and value abstraction. The former is performed by the *Heap Domain*, that abstracts all heap operations by rewriting them with synthetic variables representing heap locations they resolve to, leaving the *Value Domain* with call- and memory-free programs. Individual library functions can be modeled as *native CFGs*, that is, as special CFGs with a unique node that expresses the semantics of the whole function. We also reported our teaching experience with LiSA, which emphasizes how the modular structure enables experimenting with LiSA achievable by Master level students.

Furthermore, we presented SARL as a mean to structurally model software frameworks. Through SARL, one can write concise *framework specifications* expressing how a given framework modifies a program's execution model. Specifications are composed of syntactic rules identifying program members, with each specifying one or more annotations. Then, the program can be syntactically visited to apply such rules, and every member matched by one of these is annotated with the respective annotations. Analysis components can then react to the presence of annotations, thus taking into account the framework's non-standard semantics.

Finally, we demonstrated LiSA's capability of analyzing software written in multiple programming languages, identifying a vulnerability that spanned JAVA and C++ in a proof-of-concept case study. Instead, in the following chapters, we present two LiSA applications to individual programming languages having widely different semantics: Go (Chapter 4), a statically typed and compiled language, and PYTHON (Chapter 5), a dynamically typed and interpreted language. These are studied in the context of blockchain and data science, respectively, providing static analysis techniques to prove programs correct.

4 Smart contracts analysis

Chapter Contents

4.1	Related Work	82
4.2	Blockchain frameworks	83
4.3	Sources and sinks of non-determinism	84
4.3.1	Sources of non-determinism	85
4.3.2	Sinks of non-determinism	86
4.4	Flow analysis for non-determinism detection	88
4.4.1	An Overview on Information Flow	89
4.4.2	GoLiSA for non-determinism detection	91
4.4.3	Detection of Sources and Sinks in GoLiSA	93
4.5	Experimental Evaluation	95
4.5.1	Quantitative evaluation	95
4.5.2	Qualitative evaluation	97
4.5.3	Limits	98
4.6	Commercio.network: an industrial case study	98
4.6.1	Commercio.network	99
4.6.2	Detecting non-determinism on Commercio.network	99
4.7	Conclusion	101

In this chapter, we define a novel technique based on information flow to detect usages of non-deterministic constructs in software running on a blockchain. This chapter is based on [106] and a paper under second revision for ECOOP¹.

In the last decade, blockchain software has undergone a notable evolution. In 2008, Bitcoin [101] introduced a Turing-incomplete low-level language to specify locking conditions that must hold for a transaction to be accepted by the network [11]. In 2013, Ethereum [29, 10] provided a Turing-complete bytecode where smart contract rules are enforced by the blockchain consensus. Code execution happens on the Ethereum Virtual Machine (EVM), resulting in software identified as *decentralized applications* (DApps). EVM bytecode is supported by high-level domain-specific languages (DSLs), such as Solidity and Vyper, that have been designed from scratch for the purpose of being executed in the restricted environment of blockchain. Subsequently, thanks to frameworks such as Hyperledger Fabric [9], Tendermint [24, 86], and Cosmos SDK [87], general-purpose programming languages (GPLs) such as Go, JAVA, and JAVASCRIPT began being used to develop smart contracts and DApps, with Go being the most popular in industrial blockchains.

The popularity of GPLs for writing smart contracts and DApps is steadily increasing. Their success is mostly due to the maturity of the languages themselves, directly resulting in wide communities, consolidated tools (such as IDEs and debuggers),

¹<https://conf.researchr.org/home/ecoop-2023>.

and most importantly a pool of expert and knowledgeable developers that can write highly efficient smart contracts. Yet, GPLs were not conceived solely for blockchain ecosystems: code that is harmless and bug-free in other contexts may result in vulnerabilities and errors. Among these, one of the most insidious is non-determinism. When the result of an operation on a blockchain is non-deterministic, there is no guarantee that a common state can be reached by the network's nodes, possibly preventing it from reaching consensus. This can manifest, among other possibilities, as transaction failures or denial of service. Nevertheless, not all instances of non-determinism are intrinsically dangerous: logging the time of a transaction can result in different timestamps appearing in each node's logs, but it does not endanger consensus as it is not *observable* by other nodes. In fact, non-deterministic instructions are problematic only if they can affect the shared blockchain state. GPLs also offer constructs and features that drive their adoption in specific contexts, and these add to the existing challenges that program analysis already faces. In Go, the language tackled in this chapter, examples of such peculiar features are the composition-driven type embedding², innate concurrency constructs³, and block-based variable scoping⁴.

As an example, consider the code in Figure 4.1, reporting an excerpt of method `ValidateBasic` from the module `x/authz` (part of the Cosmos SDK versions 0.43.x and 0.44.{0,1} and affected by CVE-2021-41135⁵). The code is meant to fail the validation of expired grants. Note that the guard at line 2 involves the local clock of nodes (`time.Now()`) rather than leveraging the timestamp included in the Block header provided by the Byzantine Fault Tolerant clock, that is agreed upon by the consensus. As reported in the official Cosmos forum [58]:

Local clock times are subjective and thus non-deterministic. An attacker could craft many Grants, with different but close expiration times (e.g., separated by a few seconds), and try to exercise the granted functionality for all of them close to their expiration time. It is likely in such a scenario that some nodes would consider a grant to have expired while others would not, leading to a consensus halt.

The code was then fixed in version 0.44.2, but is still a clear example of a vulnerability arising from non-deterministic constructs. The problem tackled of blockchain non-determinism is clearly felt by the communities of the blockchain frameworks treated in this chapter. As a representative example, the Tendermint Core documentation [81], while discussing non-determinism, reports:

While programmers can avoid non-determinism by being careful, it is also possible to create a special linter or static analyzer for each language

²https://go.dev/doc/effective_go#embedding.

³https://go.dev/doc/effective_go#concurrency.

⁴https://go.dev/ref/spec#Declarations_and_scope.

⁵<https://nvd.nist.gov/vuln/detail/CVE-2021-41135>.

```
1 func (g Grant) ValidateBasic() error {
2   if g.Expiration.Unix() < time.Now().Unix() {
3     return sdkerrors.Wrap(ErrInvalidExpirationTime, "Time can't be in the
4       past")
5   }
6   // [...]
}
```

Figure 4.1: Cosmos SDK code affected by CVE-2021-41135

to check for determinism. In the future we may work with partners to create such tools.

This chapter presents a software verification approach based on static analysis for the detection of non-deterministic vulnerabilities in blockchain ecosystems, covering the most popular frameworks for developing this kind of software, such as Hyperledger Fabric, Tendermint Core, and Cosmos SDK. We shift the classical focus that has been applied in this context beyond the mere syntactic absence of non-deterministic constructs. In fact, we aim at distinguishing *harmful* usages of non-determinism, that is, constructs affecting the blockchain state and response, from *harmless* ones. As a consequence, the set of alarms issued to the user sensibly shrinks, as shifting from a syntactic approach towards a semantic one leads to a sensible reduction in false positives. We propose a semantic flow-based static analysis for detecting flows from non-deterministic constructs to blockchain state modifiers and response builders. The choice of a flow-based analysis seems natural when the problem is phrased as “*is there an execution where a non-deterministic value affects the blockchain state or the contract’s response?*”. We thus exploit the well-consolidated literature in this area to adopt scalable solutions that soundly over-approximate all program executions.

We provide a static analyzer implementing our approach: GoLiSA, a sound static analyzer based on abstract interpretation for Go applications. Intuitively, we use our analyzer’s fixpoint engine to mark *all* program variables (local variables, objects’ fields, ...) that can contain values affected, directly or indirectly, by a non-deterministic construct or computation. Specifically, we can perform a shallower analysis detecting only explicit flows using *Taint* [133, 60] analysis, where non-deterministic constructs and blockchain state modifiers are modeled as sources and sinks, respectively. Alternatively, we can perform a deeper analysis able to also detect implicit flows by means of the *Non-interference* [73, 74] analysis, where problematic constructs and blockchain state modifiers are instead modeled as low and high variables, respectively. Both solutions are implemented in GoLiSA, whose analysis starts by syntactically visiting the input application to annotate all sources and sinks. The annotations are dynamically generated depending on the kind of application of interest (i.e., targeting Hyperledger Fabric, Cosmos SDK, or Tendermint Core). Since there is no predefined set of sources in the target program, both *Taint* analysis and *Non-interference* are parametric: they consider as *harmful* (i.e., tainted or low integrity, depending on the analysis that is to be executed) only variables that are annotated

as sources. The fixpoint engine then takes care of propagating values coming from sources on the entirety of the program, exploiting our analyses implementations. After the fixpoint converges, a mapping stating if each program variable is the result of a non-deterministic computation is available at each program point. These are then used by our non-deterministic *Checks*, that visit the whole application searching for statements annotated with the sink annotation. Whenever one is found, the mappings are used to determine if the values used as parameters of the call are critical or, in the case of *Non-interference*, if the call happens in a critical state.

Our approach, as highlighted by our evaluation, shows a significant decrease in false positives on real-world blockchain applications compared to other analyzers for blockchain non-determinism. The solution has been experimented on a benchmark of more than 600 real-world blockchain programs written in Go. These show that GoLiSA is able to analyze almost the totality of the benchmark, detecting all the reported non-determinism vulnerabilities. The analyses are then evaluated in terms of precision of the results (true positive, false positive, and true negative alarms), showing that GoLiSA outperforms existing open-source static analyzers for Go blockchain software. Moreover, the evaluation shows that the execution time of the analyses is not impractical for real use cases. To the best of our knowledge, GoLiSA is the first sound semantic-based static analyzer for blockchain software able to precisely detect critical non-determinism behaviors while scaling to real-world programs.

4.1 Related Work

The non-determinism of smart contracts written in GPLs is a well-known issue [92, 143]. Takamaka [124, 125] enforces determinism by limiting the set of instructions and APIs of the target language, avoiding unsafe statements that might lead to non-deterministic behaviors through white-listing fully deterministic APIs. This approach ensures safe development while preventing API extensions coming with new language versions to bypass the check. However, it also severely limits the exploitable features of the GPL. On the other hand, black-listing undesired APIs is a much harder approach to maintain, but it seems the most widespread technique in Go analyzers. For instance, ChainCode Analyzer⁶ and Revive^{CC}⁷ detect mainly black-listed imports related non-deterministic APIs using a syntactical approach. Besides, they can detect non-deterministic map iterations by AST traversal with minimal syntactic reasoning. Signature of invoked functions can also be black-listed instead of imports [92]. These tools and frameworks inherently limit API usage, sensibly reducing the benefits of adopting a GPL even when the code poses no harm to the blockchain. Non-determinism detection has also been covered for concurrent applications, suggesting that it is “*most often the result of a mistake on the part of the programmer*” [59].

⁶<https://github.com/hyperledger-labs/chaincode-analyzerc>.

⁷<https://github.com/sivachokkapu/revive-cc>.

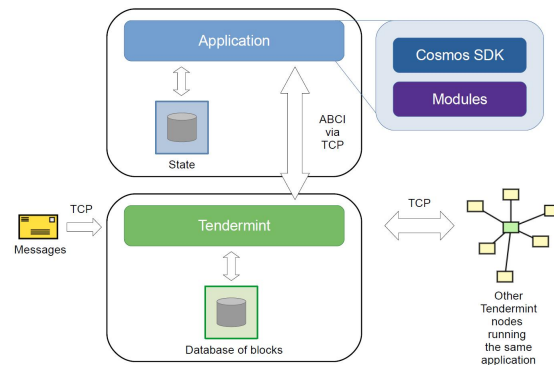


Figure 4.2: Cosmos SDK architecture

4.2 Blockchain frameworks

Here, we briefly describe the three most popular blockchain frameworks for Go: Hyperledger Fabric⁸, Tendermint⁹, and Cosmos SDK¹⁰.

Hyperledger Fabric. Hyperledger Fabric (HF) is a permissioned blockchain framework designed to be adopted in enterprise contexts, supported by the Linux Foundation and other contributors such as IBM, Cisco, and Intel. In HF, smart contracts and DApps are written in *chaincode* that can be implemented in several GPLs such as Go, JAVASCRIPT, and JAVA. In most cases, the chaincode interacts only with the world state database component of the ledger, and not with the transaction log [80]. Go is currently the most popular language on GitHub related to *chaincode*¹¹, as Go smart contracts are the best performing ones [69].

Tendermint Core and Cosmos SDK. Tendermint Core, recently rebranded as Ignite, is a platform for building blockchain nodes, supporting both public and permissioned *proof-of-stake* (PoS) networks. It is a Byzantine Fault Tolerant (BFT) middleware that separates the application logic from the consensus and networking layers, allowing one to develop blockchain applications written in any programming language, including Go, and replicate them on many machines [25].

Cosmos SDK is an open-source Go framework that eases the development of blockchain applications while optimizing their execution by running them on Tendermint Core. As shown in Figure 4.2, Cosmos SDK abstracts all the boilerplate code needed to set up a Tendermint Core node, allowing for customized protocol configurations. The programming style follows the object-capability model, where the security of subcomponents is imperative, especially those belonging to the core library.

⁸<https://www.hyperledger.org/use/fabric>.

⁹<https://tendermint.com/>.

¹⁰<https://v1.cosmos.network/sdk>.

¹¹Querying the keyword `chaincode` on GitHub (<https://github.com/search?q=chaincode>) results in more than 1900 repositories, about half of which are written in Go. Accessed 02/2022.

```

1 func transfer(from, to Address, value int64, stub *shim.ChaincodeStub) {
2   start := time.Now()
3   // operations that take few milliseconds
4   elapsed := time.Now().Sub(start)
5   log.Println("Time elapsed for the transfer operations: ", elapsed)
6 }

```

(a) Safe usage of the `time` API

```

1 func transfer(from, to Address, value int64, stub *shim.ChaincodeStub) {
2   t := time.Now()
3   // operations that take few milliseconds
4   err := shim.PutState("transaction-time", t)
5   // rest of the chaincode
6 }

```

(b) Unsafe usage of the `time` API**Figure 4.3:** Examples of harmless and harmful non-determinism in blockchain

Cosmos SDK is a framework for DApps, supporting different functionalities through highly customizable modules, that can also manage smart contracts.

Consensus in blockchain frameworks

Consensus protocols ensure the validity and authenticity of transactions performed in the blockchain, as they check results of smart contracts or DApps computations through the state of the network's nodes. If a given number of nodes agree on the final state, consensus is reached and the transaction is validated. Otherwise, it is discarded and the nodes proposing spurious states are excluded from the network. When consensus cannot be reached, the blockchain either forks or halts. Deterministic execution is thus required for software that runs in a blockchain, as it guarantees that, when starting from a common state, the same result is reached in any distinct blockchain node, avoiding inconsistencies among peers and consensus failures. Nevertheless, GPLs provide several components that can explicitly lead to non-determinism, such as (pseudo-)random values generators or external computations. Furthermore, even methods that are explicitly sequential and deterministic pose a threat when executed on different nodes, such as the `time.Now()` call from Figure 4.1. Despite these threats, popular blockchain frameworks such as HF and Cosmos SDK do not enforce particular restrictions on the usage of non-deterministic methods and components.

4.3 Sources and sinks of non-determinism

Example 4.3.1. When trying to prevent non-deterministic vulnerabilities, a first solution is to limit the expressiveness of the GPL by either black- or white-listing APIs and constructs. Consider the Go snippets reported in Figure 4.3. Both frag-

ments rely on the `time` API to retrieve a timestamp from the host system. In general, the results of calls to the `time` API are subjective to the node executing them, and they might lead to blockchain non-determinism due to different system settings (e.g., time, date, time zones, ...) or due to nodes executing the code at slightly different times. Specifically, Figure 4.3a shows a safe usage of the `time` API: the timestamp is only used for logging with no observable consequences on the blockchain state or the execution result. Instead, Figure 4.3b reports a problematic usage of the API, as the timestamp is stored in the blockchain using `PutState`, an HF-specific function that updates the shared network state. Since timestamps could differ on each node, this potentially leads to inconsistent executions (i.e., different blockchain states or execution results), causing transaction failure¹².

It should thus be evident that identifying sources of non-determinism and preventing their usage is not enough when we aim at discerning between harmful and harmless non-deterministic constructs. In fact, one should also recognize how these are used, determining if they can influence the shared blockchain state. In the rest of this section we discuss, for each blockchain framework presented in Section 4.2, (i) the constructs that generate potentially harmful non-determinism (that is, *sources* of non-deterministic values), and (ii) the blockchain state modifiers and response builders (i.e., statements that make a transaction succeed or fail), namely *sinks* that are sensitive to non-determinism¹³. This will prepare the ground for the core contribution of this chapter: a static approach to detect critical usages of non-determinism in blockchain software, reported in Section 4.4.

4.3.1 Sources of non-determinism

The sources of non-determinism can be logically split into two families, the first being related to the combination of framework and GPL adopted to develop the software. This family comprises a set of constructs and APIs allowed by the framework that may break the consensus during the execution of smart contracts or DApps. In Go, these are:

- *iteration over maps* that, being the iteration order unspecified¹⁴, is not guaranteed to be deterministic;
- *parallelization and concurrency*, that can lead to race conditions on shared resources, thus creating non-determinism on the computed values;
- *global variables*, that may change innately and cause inconsistencies in the results, since they depend on the application state of a peer and not on that of the blockchain [92, 23].

¹²In this case, the `GetTxTimestamp` method from the HF API should have been used instead of `time.Now`.

¹³The complete list of sources and sinks of non-determinism is available at <https://github.com/lisa-analyzer/go-lisa/blob/master/go-lisa/sources-sinks.md>.

¹⁴https://golang.org/ref/spec#For_statements

Level	Category	Package	Statements/Methods
Framework/Language	Map iteration	-	range on map
	Parallelization/concurrency	-	go (Go routine), <- (channel)
	Random value generation APIs	math/rand, crypto/rand	*
	Global variables	-	-
Environment	File system APIs	io, embed, archive, compress	*
	OS APIs	os, syscall, internal, time	*
	Database APIs	database	*
	Internet APIs	net	*

Table 4.1: Potential non-deterministic behaviors related to Go

- *random value generators*, that can potentially be allowed in smart contracts [31] to employ custom logic while being non-deterministic by definition.

The second family instead involves statements related to the underlying environment. While these are not intrinsically non-deterministic, their result cannot be expected to remain consistent on different nodes. These comprise APIs handling:

- *file systems*, as the program might rely on files that are not present on all nodes, as they might have been deleted, edited, moved, or there might be insufficient disk space causing any operation to fail;
- *operating systems* (OS), since the blockchain might operate on various hosts and language APIs could return different results on each OS (e.g., time and date methods could return different values if nodes are not synchronized);
- *databases*, where records might be deleted, edited, or contain different data;
- *Internet connections*, as networking setup or errors could cause some addresses to be unreachable on few nodes of the network.

Table 4.1 summarizes the instructions and libraries of Go¹⁵ that we consider as possible causes of non-determinism, where * represents the entirety of the package. For the sake of simplicity, the table reports instructions and packages omitting the full signatures of each method. Note that only few methods within those packages lead to non-deterministic behaviors: for instance, most methods from package `time` handling dates and times do not pose a threat in smart contracts and DApps, and are in fact quite common. However, operations such as retrieving the current time of the OS (i.e., methods `Since`, `Now`, `Until`) are potentially dangerous.

4.3.2 Sinks of non-determinism

Sinks of non-determinism comprise constructs and APIs with the ability to both modify the common state of the blockchain, or have an impact on the response of

¹⁵The full list of Go APIs sources considered in our analyses is available at https://github.com/lisa-analyzer/go-lisa/blob/master/go-lisa/src/main/resources/for-analysis/nondeterm_sources.txt.

```

1 func (s *SmartContract) transaction(APIStub shim.ChaincodeStubInterface)
  sc.Response {
2   if rand.Int() % 2 == 0 {
3     return shim.Error("Fail")
4   } else {
5     return shim.Success(nil)
6   }
7 }

```

Figure 4.4: Non-determinism related to the blockchain response

Framework	Package	Type/Interface	Statements/Methods	Sink
HyperLedger Fabric	shim	ChaincodeStubInterface	PutState, DelState PutPrivateData, DelPrivateData	parameters
			Success, Error	statement
Tendermint Core	abci/types	Application	ResponseBeginBlock, ResponseDeliverTx, ResponseEndBlock, ResponseCommit, ResponseCheckTx	instance returned
Cosmos SDK	types	KVStore	Set, Delete	parameters
	kv, dbadapter, gaskv, listenkv, prefix, iavl, tracekv	Store	Set, Delete	parameters
	types/errors		ABCIError, Redact ResponseDeliverTx, ResponseCheckTx, WithType, Wrap, Wrapf	statement

Table 4.2: Main sinks for blockchain software written in Go

blockchain networks. While the former is inherently involved in consensus protocols, the execution of code within the blockchain does not necessarily change the shared state (e.g., functions that simply read a value). However, the execution may lead to non-deterministic responses, compromising the consensus of the network, as in the trivial example of Figure 4.4. Table 4.2, where the **Sink** column identifies what part of the API should not receive non-deterministic values, summarizes the main instructions and components that we consider as sinks for non-determinism.

Hyperledger Fabric APIs for Go. In HF, chaincode executes transaction proposals against world state data that may change its state. Programmatically, interface `ChaincodeStubInterface` from the HF Go APIs enables access and modification of the blockchain state. Table 4.2 reports the current components (as of version 2.4) involved in the data-write proposal. Their semantics does not affect the blockchain state until the transaction is validated and successfully committed. Hence, if these components lead to different results due to non-determinism, consensus will not validate the transaction and no new state will be committed. HF provides `Success` and `Error` to yield successful and failed transaction responses, respectively.

Tendermint Core APIs for Go. Tendermint Core is a middleware with no explicit access to the application state by design, enabling communication through the *Ap-*

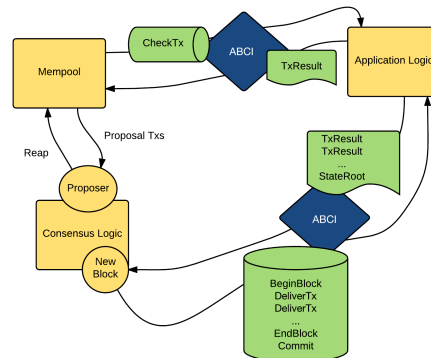


Figure 4.5: ABCI methods and consensus flow

plication *Blockchain Interface* (ABCI¹⁶). Figure 4.5 depicts the consensus process used to validate and store a transaction using the ABCI methods. As reported in the official documentation of Tendermint Core v0.35.1, only `BeginBlock`, `DeliverTx`, `EndBlock`, and `Commit` must be strictly deterministic to ensure consensus. Although the logic of these methods is different, they possess similar structure: they all accept a request and return a response (`ResponseBeginBlock`, `ResponseDeliverTx`, `ResponseEndBlock`, `ResponseCommit`), with the latter that must be deterministic.

Cosmos SDK APIs. Cosmos SDK handles both the application and the blockchain state through the *store*¹⁷. At a high level, the store is a set of key-value pairs used to store and retrieve data, implemented by default as a *multistore* (i.e., a store of stores), as shown in Figure 4.6. The multistore encapsulation enables modularity of the Cosmos SDK, as each module declares and manages its own subset of the state using specific keys. Keys are typically held by *keepers*, a Cosmos SDK abstraction with the role of managing access to the multistore’s subset defined by each module. All Store definitions implement the `KVStore` interface. The latter provides common APIs to access and modify the state of the blockchain using methods such as `Set` and `Del`. As for responses, Cosmos provides several methods (such as `ABCIError`, `Wrap`, `ResponseDeliverTx`) in package `types/errors` to fail transactions.

4.4 Flow analysis for non-determinism detection

In this section, we introduce and discuss our approach for detecting non-deterministic behaviors in blockchain software. In particular, we consider non-determinism as *critical* only if a non-deterministic value can affect the blockchain state, either directly (i.e., being stored inside the state) or indirectly (e.g., guarding the execution of state updates). Any other usage of non-determinism is considered safe, as it does not affect

¹⁶<https://github.com/tendermint/spec/blob/master/spec/abci/abci.md>.

¹⁷<https://docs.cosmos.network/master/core/store.html>.

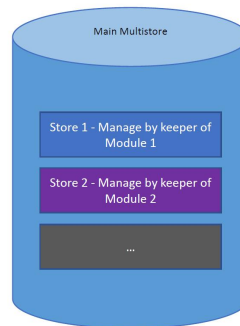


Figure 4.6: Main store of Cosmos SDK

the blockchain state or response. As such, when mentioning non-determinism in the remainder of the chapter, we implicitly refer to its critical version. We rely on information flow analysis for detecting values originating from sources of non-determinism that can affect the state of the blockchain. We only focus on static analyses, since they soundly over-approximate all possible behaviors of target programs and can thus give guarantees about the absence of such behaviors. We instantiate two types of analyses: a *Taint* analysis, able to capture the so-called *explicit flows*, and a *Non-interference* analysis, that can also detect *implicit flows*.

4.4.1 An Overview on Information Flow

Information flow analyses [55, 118] address the problem of understanding how information *flows* from one variable to another during a program’s execution. These analyses usually partition the space of program variables into *private* (or secret) and *public*, with the latter being accessible to — and in some cases also modifiable by — an external attacker. The goal of these analyses is then to find program executions where information *flows* from one partition to the other, that is, where values of variables from one partition can affect the values of variables from the other one. Figure 4.7 reports examples¹⁸ of the three main types of flows, namely:

- *explicit flow*: a secret variable is assigned to a value obtained using public ones;
- *implicit flow*: an assignment to a secret variable is conditionally executed depending on values of public ones;
- *side channel*: observable properties of the execution, e.g., the amount of computational resources used, depends on the values of some secret variables.

In general, the term *source* is traditionally used for variables holding values that one wants to track along program executions, while *sink* is used to describe locations where values coming from sources should not flow. Using this terminology, when

¹⁸[https://en.wikipedia.org/wiki/Information_flow_\(information_theory\)](https://en.wikipedia.org/wiki/Information_flow_(information_theory)).

```

1 var l1, l2, l3 (* public variables *)
2 var h1, h2, h3 (* private variables *)
3 h1 := l1 (* explicit flow from l1 to h1 *)
4 if l2 = true then
5   h2 := 3 (* implicit flow from l2 to h2 *)
6 if l3 = 1 then
7   (* expensive long-running work *)
8 h3 := 0 (* side channel from l3 to h3 *)

```

Figure 4.7: Example of explicit, implicit, and side channel flows

the property of interest ensures the *integrity* of secret variables, information flow analyses can be instantiated using public variables as sources and private ones as sinks, exactly as in Figure 4.7 and the list above. These can detect situations where (i) a possibly corrupted value provided by a malicious attacker could be stored into variables whose content is supposed to be reliable, or (ii) such a value governs the update to private variables. If, however, one wants to ensure the *confidentiality* of secret variables, the same analyses can be recasted with private variables acting as sources and public ones as sinks, thus searching for flows in the opposite direction. The target of the analysis is then to find disclosures of private data to external entities.

In the context of non-deterministic behaviors in blockchain environments, information flow analyses can be used to detect when non-deterministic values end up or affect the blockchain's state, thus checking the *integrity* of that state w.r.t. non-deterministic values. As such, we are interested in information flowing from public to private variables, and we will use *sources* to identify ones that are initialized to non-deterministic values and *sinks* to identify all variables that have an effect on the blockchain's state. Moreover, we will focus on explicit and implicit flows. In fact, side channels are typically studied to detect secret information leaking through, for instance, execution time, thus violating the confidentiality of that information instead of its integrity. On the other hand, explicit and implicit flows identify non-deterministic values that are either used to update the blockchain's state or a transaction's result, or that govern their execution. As a concrete example, recall the code from Figure 4.1: the vulnerability presented there is an implicit flow since the blockchain's state is not directly updated with non-deterministic values, but the execution of the update (i.e., the return statement) is conditional to some non-deterministic value (i.e., `g.Expiration.Unix() < time.Now().Unix()`).

In the following, we introduce two well-established information flow analyses that we will use for non-determinism detection.

Non-interference

Non-interference [73, 74] is a notion of security capturing the idea that if computations over private information are independent from public information, then no leakage of the former can happen. In simple terms, after partitioning the space of inputs of a program P into *low* (private or secret, denoted by \mathbb{L}), and *high* (public

or available to anyone, denoted by \mathbb{H}), *Non-interference* is satisfied if changes in the high input do not affect the observable (i.e., low) output of the program:

$$\forall i_{\mathbb{L}} \in \mathbb{L}, \forall i_{\mathbb{H}}, i'_{\mathbb{H}} \in \mathbb{H} . P(i_{\mathbb{L}}, i_{\mathbb{H}})_{\mathbb{L}} = P(i_{\mathbb{L}}, i'_{\mathbb{H}})_{\mathbb{L}}$$

This notion is often instantiated in language-based security by partitioning the space of program variables between \mathbb{L} and \mathbb{H} , and finding instances of explicit or implicit flows between these partitions. Such analysis computes, for each program point, a mapping from variables to the information level they hold (low or high), while also keeping track of an execution state depending on the information level of the Boolean conditions that guard the program point. Violations of *Non-interference* for integrity can then be detected whenever an assignment to a variable in \mathbb{H} either (i) assigns a low value (that is, an expression involving variables in \mathbb{L}), or (ii) happens with a low execution state (that is, guarded by at least a Boolean condition that involves variables in \mathbb{L}), thus identifying both explicit and implicit flows. This can be formalized as a type system for security [118].

Taint Analysis

Taint analysis [133, 60] is an instance of information flow analysis that can be seen as a simplification of *Non-interference* considering only explicit flows. In this context, variables are partitioned into *tainted* and *untainted* (or *clean*), with the former representing variables that can be tampered with by an attacker and the latter representing variables that should not contain tainted values across all possible program executions. Roughly, *Taint* analysis corresponds to the language-based *Non-interference* instantiation without the execution state, thus unable to detect implicit flows. *Taint* has been instantiated to detect many defects in real-world software, such as web-application vulnerabilities [26], privacy issues [68] (also related to GDPR compliance [67]), and vulnerabilities of IoT software [64].

4.4.2 GoLiSA for non-determinism detection

At this point, we are in position to instantiate GoLiSA¹⁹, a Go frontend for LiSA, for the static detection of non-deterministic behaviors in blockchain software. The core idea of our solution is to track the values generated by the hotspots (that is, the sources) identified in Section 4.3.1 during the execution of a program using either *Taint* analysis or *Non-interference*. Similarly, after the analysis completes, we can use a *Check* to exploit the abstract information provided by the domain of choice, checking if any of the sinks specified in Section 4.3.2 receives one such non-deterministic value as parameter or, in the case of *Non-interference*, if the sink is found in a *low* execution state.

GoLiSA's analysis is instantiated as follows:

¹⁹<https://github.com/lisa-analyzer/go-lisa>.

- *Taint* analysis and *Non-interference* are implemented as *Value Domains*, both of them being *Non-Relational Domains* (i.e., mapping from variables to abstract values — taintedness and integrity level respectively — with no relations between different variables), with *Non-interference* exploiting a special instance of `ValueEnvironment` that also keeps track of abstractions for each guard, evaluating them through the domain's own `eval` method;
- field-insensitive program point-based *Heap Domain* (Section 8.3.4 of [117]), where any concrete heap location allocated at a specific program point is abstracted to a single abstract heap identifier;
- context-sensitive *Interprocedural Analysis*, abstracting full call-chain results until a recursion is found;
- runtime types-based *Call Graph*, using the runtime types of call receivers to determine their targets;
- two *Checks*, for *Taint* analysis and *Non-interference*, that scan the code in search for sinks, checking the taintedness or integrity level of each sink.

The analysis begins by visiting the input program to detect the statements annotated as sources and propagating the information from them. The analyses produce, for each program point, a mapping stating if each variable is the result of a non-deterministic computation. These mappings are then used by our *Checks*, that visit the program in search of statements annotated as sinks. When one is found, the mappings are used to determine if values used as parameters of the call are critical or, in the case of *Non-interference*, if the call happens on a critical state. The choice of the analysis to run (and thus of the *Check* to execute) is left to the user.

Example 4.4.1. For instance, let us consider the fragment reported in Figure 4.3a. At line 5, despite variable `elapsed` being marked as tainted, no warning is raised by GoLiSA regardless of the chosen analysis, as it does not reach any sensitive sink. Instead, analyzing the fragment from Figure 4.3b results in the following alarm:

The value passed for the 2nd parameter of this call is tainted, and it reaches the sink at parameter 'value'

The warning is issued with both analyses, since variable `t` is marked as tainted and reaches a blockchain state modifier through an explicit flow.

Consider now the example reported in Figure 4.1. Here, no explicit flow happens at line 3, that contains the blockchain state modifier `Wrap`, but its execution depends on the non-deterministic value used in the condition at line 2, that is, `time.Now().Unix()`. As this is an implicit flow, the *Taint* analysis is not able to detect it. GoLiSA will however discover it with *Non-interference*, raising the following alarm:

The execution of this call is guarded by a tainted condition, resulting in an implicit flow

4.4.3 Detection of Sources and Sinks in GoLiSA

To exploit information flow analyses, the analyzer must know which are the sources and sinks of the program. GoLiSA provides a solution based on annotations, marking the corresponding statements as sources and sinks. In the following, we describe how GoLiSA annotates sources (Table 4.1) and sinks (Table 4.2).

Methods and functions. As shown in Tables 4.1 and 4.2, all sinks and several sources correspond to functions and methods of APIs from either the Go runtime or the blockchain frameworks. GoLiSA contains a list of the signature of these functions and methods and it automatically annotates the corresponding calls in the program by syntactically matching them. This process does not rely on SARL (Section 3.6) yet, and we plan to transition to using it as future work.

Example 4.4.2. For instance, when GoLiSA iterates over the following snippet, it is able to discover the call to `time.Now`, that gets annotated as source, and the one to `PutState`, whose parameters get annotated as sinks:

```
1 key := "key"
2 tm := time.Now()
3 stub.PutState(key, []byte(tm))
```

Then, the analysis propagates taintedness from the return value of `time.Now()` to the second parameter of `PutState`, thus issuing an alarm at line 3.

Map Iterations. To detect iterations over maps, one needs to reason about typing. GoLiSA exploits runtime types inferred by LiSA to identify `range` statements happening over maps. If a map iteration occurs, that is, if the object in a `range` statement is inferred to be a map, then GoLiSA marks as sources the variables used to store keys and values of the map.

Example 4.4.3. Consider the following code snippet:

```
1 s := ""
2 kvs := map[string]string{"a": "hello", "b": "world!"}
3 for k, v := range kvs {
4     s += v
5 }
6 stub.PutState("key", []byte(s))
```

While analyzing the code, `range` statements are checked for the types of their parameter. GoLiSA annotates as sources both `k` and `v`, as `kvs` is inferred to be a map, while the sink at line 6 is detected through already discussed annotations.

Information flow analyses can then propagate the taintedness from `v` to `s`, that in turn flows to the second parameter of `PutState`, issuing an alarm at line 6.

Global variables. GoLiSA syntactically annotates all global variables as a source of non-determinism, as their value could be modified independently on each peer.

Example 4.4.4. In the following code, the value of global variable `glob` could differ from peer to peer depending on the number of times function `inc` has been executed. This can happen as not all peers simulate the same transaction, for instance due to differences in the endorsement policy of each peer [92].

```

1 var glob string
2 func inc() {
3     glob += "a"
4 }
5 func (s *SmartContract) transaction(stub shim.ChaincodeStubInterface)
   sc.Response {
6     stub.PutState("key", []byte(glob))
7 }

```

Before the analysis, GoLiSA iterates over all program components, annotating `glob` as a source. The sink at line 6 is annotated as sink as previously discussed. Then, the information flow analysis propagates taintedness from `glob` to the second parameter of the call to `PutState`, raising an alarm at line 6.

Go routines. GoLiSA inspects the code of Go routines, checking the scope of variables they use. If these are defined outside the routine using them, they are effectively shared among threads, potentially leading to race conditions or non-deterministic behaviors. Hence, GoLiSA annotates such variables as sources.

Example 4.4.5. The following snippet defines and invokes a simple Go routine that modifies a variable defined in an enclosing scope:

```

1 s := ""
2 go func(){
3     for i := 1; i <= 10000; i++ {
4         s += "0"
5     }
6 }
7 stub.PutState("key", []byte(s))

```

When GoLiSA finds the Go routine, it checks the scopes of each variable, inferring that `s` is declared outside the routine itself. Hence, GoLiSA annotates `s` at line 1 as source, while the sink at line 7 is annotated as previously discussed. Then, the information flow analysis propagates taintedness from `s` to the second parameter of `PutState`, issuing an alarm at line 7 since the value of `s` depends on how many times the Go routine has executed the loop body.

Go channels. Channels are pipes that connect concurrent Go routines. Operator `<-` allows interaction with channels to retrieve a value from them, blocking until

one is available. GoLISA annotates as sources the instructions reading values from channels, as the order in which these are written to is intrinsically non-deterministic.

Example 4.4.6. Consider the following example:

```
1 c := make(chan int)
2 go myroutine1(c)
3 go myroutine2(c)
4 x, y := <- c, <- c
5 stub.PutState("key", []byte(x))
```

GoLISA iterates over the program searching for occurrences of operator `<-`. It then annotates variables `x` and `y` as sources, as they receive a value from `c`. The sink at line 5 is detected as previously discussed. The analysis then propagates taintedness from `x` to the second parameter of `PutState`, resulting in an alarm at line 5.

4.5 Experimental Evaluation

In this section, we experimentally evaluate the analyses implemented in GoLISA to detect non-determinism issues in blockchain software. First, we study them quantitatively, on a set of 651 real-world HF smart contracts retrieved from public GitHub repositories. The HF framework was chosen for the quantitative evaluation as, to the best of our knowledge, it is the only framework supported by several static analyzers detecting non-determinism issues, and in particular by the ones involved in our comparison with GoLISA. Furthermore, HF is currently the most popular and widespread blockchain framework among public GitHub repositories, with most chaincodes written in Go. We then evaluate the quality of our results on two applications, to show how the analyses work and how the information is propagated in programs. In particular, we selected the first application from the HF benchmark, while the second one is the Cosmos SDK code reported in Figure 4.1.

All the experiments have been performed on an HP EliteBook 850 G4 equipped with an Intel Core i7-7500U at 2,70/2,90 GHz and 16 GB of RAM memory running Windows 10 Pro 64bit, Oracle JDK version 13, and Go version 1.17.

4.5.1 Quantitative evaluation

The experimental artifact set has been retrieved from 954 GitHub repositories, by querying for the `chaincode` keyword, as smart contracts are called in HF, and selecting ones from unforked Go repositories only²⁰, and that include the `Invoke` and `Init` methods: these are the transaction requests' entry points for chaincodes²¹. Then, we filtered out files unrelated to smart contracts and removed chaincodes not analyzable due to failures of either GoLISA or the tools discussed in Section 4.5.1. Failures

²⁰<https://api.github.com/search/repositories?q=chaincode+fork:false+language:Go+archived:false&sort=stars&order=desc>. Accessed: 17-10-2022.

²¹See <https://pkg.go.dev/github.com/hyperledger/fabric-chaincode-go/shim>.

Analysis	#A	#U	ET	AT	#W	#TP	#FP	#FN
<i>Taint</i>	59	592	2h:14m:21s	12.38s	155	117	38	7
<i>Non-interference</i>	60	591	2h:24m:24s	13.29s	175	124	51	0

Table 4.3: Analysis evaluation

of GoLiSA on such chaincodes are due to missing support of high-order functions, recursion, and C code invocation via the built-in Go `cmd/go` package²². This resulted in a benchmark consisting of 651 files (~167391 LoCs), that, from here on, we refer to as IHF.

Table 4.3 reports the results of the experimental evaluation of GoLiSA over IHF, where **#A** is the number of affected chaincodes (i.e., chaincodes where at least a warning was issued), **#U** is the number of unaffected chaincodes (i.e., chaincodes where no warning was raised), **ET** is the total execution time, **AT** is the average execution time, **#W** is the total number of warnings issued, **#TP** is the number of true positives among the raised warnings, **#FP** is the number of false positives among the raised warnings, and **#FN** is the number of false negatives, namely warnings that were not issued. In terms of execution time, the analyses performed on average in around 15 seconds per chaincode. The experiments show that *Non-interference* performs better than *Taint* in terms of precision, being able to detect all the true positives contained in IHF, with a low false positives rate (29.14%). This was expected since, as we have already discussed in Section 4.4 and unlike *Non-interference*, *Taint* is only able to track explicit information flows. In fact, the 7 false negatives (column **#FN** of Table 4.3) produced by *Taint* correspond to implicit non-deterministic behaviors. To confirm our experimental results, all chaincodes contained in IHF were manually checked to ensure that no critical non-determinism behavior was missed by GoLiSA.

Comparison

We compared GoLiSA with the open-source static analyzers for Go chaincode described in Section 4.1, namely ChainCode Analyzer and Revive^{CC}. Table 4.4 reports the comparison between GoLiSA and these tools over IHF.

The comparison shows that GoLiSA - *Non-interference* finds all the true issues contained in the benchmark, achieving the best and most accurate result in terms of precision with a 29.14% false positives ratio. Instead, although it has some false negatives, GoLiSA - *Taint* is the analysis with the lowest percentage of false positives with a ratio of 24.52%.

Revive^{CC} triggers 351 warnings out of which 77.49% are false positives. The only non-deterministic behavior missed by Revive^{CC} (last column of Table 4.3) is due to it considering the `io/ioutil.ReadFile` API as safe, although reading a file should be considered non-deterministic in blockchains. Finally, ChainCode Analyzer is more pre-

²²We decided not to implement those standard features since this would have required a relevant effort to support only a few more chaincodes.

Tools	# Warning	# TP	# FP	# TN
GoLiSA - <i>Taint</i>	155	117	38	7
GoLiSA - <i>Non-interference</i>	175	124	51	0
ChainCode Analyzer	203	68	135	53
Revive ^{CC}	351	79	272	1

Table 4.4: Warnings triggered by the analyzers on HF

cise w.r.t. Revive^{CC}, with 66.50% of false positives, but it also has the highest number of false negatives, failing to detect a huge number of critical non-deterministic behaviors. This can be attributed to the fact that ChainCode Analyzer does not consider several APIs leading to non-determinism as critical and it fails to soundly detect iteration over maps.

Note that the amount of true positives discovered by GoLiSA analyses differs from the ones of other tools. In fact, GoLiSA is the only tool involved in our comparison that issues warnings on sinks rather than sources. This translates to fewer alarms being issued whenever values of multiple sources flow to the same sink (here, GoLiSA issues a single warning, while other tools issue one for each source), and to more alarms being raised whenever the value of a single source flows to multiple sinks (here instead, other tools issue a single warning, while GoLiSA issues one for each sink).

4.5.2 Qualitative evaluation

Explicit Flow: the Boleto contract. The *boleto*²³ contract, taken from IHF, comes with a real non-determinism issue that can be found with explicit flows, and that was also detected by other tools during the comparison of Section 4.5.1. The *boleto* contract seems to be a proof of concept application handling tickets in an e-commerce store, with the method `registrarBoleto` used to register a ticket:

```

1 func (s *SmartContract) registrarBoleto(APIStub shim.ChaincodeStubInterface,
   args []string) sc.Response {
2 // [...]
3 objBoleto.CodigoBarra = strconv.Itoa((rand.Intn(5) + 10000000 + // [...]
4 var notExpiredDate = time.Now()
5 objBoleto.DataVencimento = notExpiredDate.Format("02/01/2006")
6 // [...]
7 boletoAsBytes, _ := json.Marshal(objBoleto)
8 APIStub.PutState(args[0], boletoAsBytes)
9 // [...]
10 }

```

Analyzing *boleto*, GoLiSA detects the explicit flow leading to a non-deterministic behavior with both *Taint* and *Non-interference*. Method `registrarBoleto` contains two different sources of non-determinism that directly flow into the same sink: the *Random API* used to generate a barcode at line 3, and the *OS API* that retrieves the

²³<https://github.com/arthurmsouza/boleto/blob/master/boleto-chaincode/boleto.go>

local machine's time to set a date at line 4. As values from both sources are used to update fields of `objBoleto`, the latter is marked as tainted by the analysis, resulting in `boletoAsBytes` being tainted as well. `PutState`'s parameters are considered as sinks by GoLiSA's analyses (Table 4.2). According to the official documentation of HF²⁴, the `PutState` method does not affect the ledger until the transaction is validated and successfully committed. However, a transaction needs to produce the same results among different peers to be validated: as passing non-deterministic values to `PutState` will cause the transaction to fail, GoLiSA raises a warning on line 8.

Implicit Flow: Cosmos SDK v.43. Analyzing the code in Figure 4.1, GoLiSA is able to detect an implicit flow that leads to a non-deterministic behavior, that can only be detected using *Non-interference*. The `ValidateBasic` method of Cosmos SDK v. 0.43.x and v. 0.44.{0,1} was designed to validate a grant to ensure it has not yet expired. In this case, the source detected by GoLiSA is the *OS API* used to retrieve the local machine time involved in the expiration check of the grant time at line 2 of Figure 4.1. By propagating the information, GoLiSA detects that the expiration check governs the execution of the return statement. Since the `Wrap` method is annotated as a sink, GoLiSA triggers an alarm at line 3 of Figure 4.1 as the sink is contained in a block whose guard depends on non-deterministic values.

4.5.3 Limits

Unlike some frameworks and GPLs used in other blockchains, the frameworks targeted in this chapter are used to develop *permissioned*, and often *private*, blockchains, meaning that the related software is not publicly available. This is the reason why the benchmark HF crawled from GitHub consists of 651 chaincodes, a number that is not comparable with smart contract benchmarks obtained investigating other (public and permissioned) blockchains. For instance, [141] collects 3075 distinct smart contracts from the Ethereum blockchain, resulting in a wider benchmark.

The proposed solution for detecting non-deterministic behaviors is fully static. It is well known that static analysis is intrinsically conservative and may produce false positives. Even if none have been raised by GoLiSA on the selected benchmark, one should expect false positives when applying our approach to arbitrary DApps.

4.6 Comercio.network: an industrial case study

The approach presented in this chapter has been applied, in a joint work with Comercio.network²⁵, to analyze their homonymous blockchain on version 2.2.0²⁶, com-

²⁴<https://github.com/hyperledger/fabric-chaincode-go/blob/1476cf1d3206f620db7eea12312c98669d39fa22/shim/interfaces.go>.

²⁵<https://comercio.network/>.

²⁶<https://github.com/comercionetwork/comercionetwork/tree/v2.2.0>.

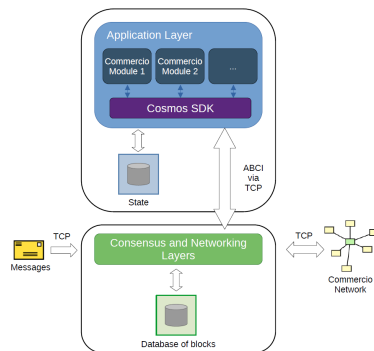


Figure 4.8: Commercio.network architecture

posed by 248 Go files (14961 LoCs). This section is based on the published paper [106].

4.6.1 Commercio.network

Commercio.network [37] is an open-source decentralized application framework provided by the homonymous company. As a blockchain, it can be described as a *permissioned Proof-Of-Stake* network, where a validator must join a consortium for being able to participate in the consensus. It can be also described as *public*, since anyone can set up a node and synchronize it with the Commercio.network main-net. The main purpose of this blockchain is to exchange electronic documents in a legally binding way thanks to the eIDAS Compliance²⁷, while following the principles of Self-Sovereign Identity²⁸. As shown in Figure 4.8, Commercio.network is based on Cosmos SDK. We recall here that Cosmos SDK is a framework for DApps built over Tendermint Core. In this context, the architecture of a module conventionally revolves around the *keeper*, a package and entity implementing its core functionalities. For example, the Commercio.network module `commerciokyc` uses the keeper of another custom module, `commerciomint`, along with other modules coming from the library of Cosmos SDK.

4.6.2 Detecting non-determinism on Commercio.network

We applied GoLiSA and our information flow approach to investigate possible bugs in the Commercio.network blockchain. The result of the analyses performed by GoLiSA highlighted two problems related to a single issue, that follows the same pattern as the Cosmos SDK bug discussed at the beginning of this chapter, and shown in Figure 4.1.

²⁷<https://digital-strategy.ec.europa.eu/en/policies/eidas-regulation>.

²⁸<http://www.lifewithalacrity.com/2016/04/the-path-to-self-sovereign-identity.html>.

```

1 func (k Keeper) AssignMembership(ctx sdk.Context, /* [...] */ , expited_at
  time.Time) error {
2   /* [...] */
3   if expited_at.Before(time.Now()) {
4     return sdkErr.Wrap(sdkErr.ErrUnknownRequest, fmt.Sprintf("Invalid expiry
      date: %s", expited_at))
5   }
6   /* [...] */
7 }

```

Figure 4.9: AssignMembership snippet from the commerciokeyc module

```

1 func (k Keeper) BurnCCC(ctx sdk.Context, user sdk.AccAddress, id string,
  burnAmount sdk.Coin) error {
2   pos, found := k.GetPosition(ctx, user, id)
3   if !found { /* [...] */ }
4   // Control if position is almost in freezing period
5   freezePeriod := k.GetFreezePeriod(ctx)
6   if time.Now().Sub(pos.CreatedAt) <= freezePeriod {
7     return sdkErr.Wrap(sdkErr.ErrInvalidRequest, "cannot burn position yet in
      the freeze period")
8   }
9   /* [...] */
10 }

```

Figure 4.10: BurnCCC snippet from the commerciomint module

Bug #1

The bug appears in the keeper package of module `commerciokyc`, in method `Membership`. It is located at line 89 of `keeper.go`²⁹. In a nutshell, the method enables the assignment of a `Commercio.network membership` of a given type to the specified user. As shown in Figure 4.9, the issue involves two main components: method `time.Now()` and the return of a wrapped error. The latter returns an error (wrapped using the `Wrap()` method of Cosmos SDK) to the caller, leading to a transaction failure. As discussed in previous sections, the time provided by `time.Now()` could differ on each node of the network, resulting in a consensus break.

Bug #2

The bug appears in the keeper package of the `commerciomint` module, inside method `BurnCCC`. It is located at line 174 of file `keeper.go`³⁰. In a nutshell, this method allows burning (i.e., removing) an amount of currency at the conversion rate stored in a `position`, retrievable from the keeper's store with a user account address and an id. If successful, `BurnCCC` gives back to the user the collateral amount, then updates or deletes the considered position but only if enough time, called `freeze period`, has passed since its creation. Similarly to Bug #1, this issue also involves two main components: the non-deterministic method `time.Now()` and the return of a wrapped error. The relevant code introducing the vulnerability is shown in Fig-

²⁹Source code available at: <https://github.com/commercionetwork/commercionetwork/blob/3e02d5e761eab3729ccf6f874d3c929342e4230c/x/commerciokyc/keeper/keeper.go#L89>.

³⁰Source code available at: <https://github.com/commercionetwork/commercionetwork/blob/3e02d5e761eab3729ccf6f874d3c929342e4230c/x/commerciomint/keeper/keeper.go#L174>.

ure 4.10. `BurnCCC` may return an error depending on the value assumed by `guard time.Now().Sub(pos.CreatedAt) <= freezePeriod`, that once more may evaluate differently on each node of the network, thus breaking consensus.

Patching and testing considerations

After a deep investigation, the company reports that no incidents or transaction failures happened because of these bugs during the live period of the release V2.2.0. Both bugs were patched in the major release v3.0.0. A repeated analysis of the latter did not find the aforementioned problems. The issue has been resolved by getting the time directly from the current Tendermint block header, a source that is both deterministic and supported by consensus. More precisely, the Cosmos SDK context method `ctx.BlockTime()` has been used instead of `time.Now()`.

Packages containing the bugs were tested with the standard Go testing framework and the libraries supported by Cosmos SDK, obtaining a satisfying level of code coverage. In particular, the keeper packages of `commerciomint` and `commerciokyc` in version v2.2.0 had a test coverage of 83.9% and 91.9%, respectively. However, both defects could not be detected by the test cases, due to incorrect initialization of testware. First, the Cosmos SDK blockchain Context passed to the keeper methods is set to the current time, invoking `WithBlockTime(time.Now())`. Furthermore, a different usage of `time.Now()` has been leveraged for the initialization of testing variables and struct fields regarding time. These are definitely some testing anti-patterns, since not only blockchain code involved in consensus but also tests should be deterministic. Therefore, it is recommended to use a fixed timestamp in tests, wherever some logic depends on the use of time.

4.7 Conclusion

In this chapter, we proposed a flow-based approach for detecting non-deterministic behaviors that are harmful for blockchains, as they might break consensus. Our proposal has been implemented in `GoLiSA`, a `LiSA` frontend for Go applications. To the best of our knowledge, `GoLiSA` is the first semantic-based static analyzer for blockchain software that is able to detect non-deterministic behaviors, with a low false positive rate. In the context of smart contracts, the proposed approach is placed in an off-chain architecture, i.e., the analysis is done before smart contracts are deployed in the blockchain, and it is not mandatory.

This work thus constitutes a first in-depth application of `LiSA`, targeting a statically typed and compiled language. In the next chapter, we present a second application of `LiSA`, targeting a dynamically typed and interpreted language, namely `Python`, that demonstrates the applicability of `LiSA`'s framework to a widely different programming language.

5 Analysis of data science programs

Chapter Contents

5.1	Related work	105
5.2	A concrete semantics for transformations	105
5.2.1	Obtaining the semantics of Python code	107
5.3	The dataframe graph domain	108
5.3.1	Abstract semantics	112
5.4	A first application: inferring dataframes shape	118
5.5	An early experiment using PyLiSA	121
5.6	Conclusion	122

A second application of LiSA is presented in this chapter, where we employ the PyLiSA frontend to analyze JuPyter¹ notebooks used in data science. The ever-increasing usage of data-driven decision processes led to *data science* (DS) and *machine learning* (ML) permeating several areas of everyday life, reaching outside the boundaries of computer science and software engineering. Ensuring correctness of these processes is particularly important when they are employed in critical areas like medicine, public policy, or finance. The work presented in this chapter is still ongoing; the following is thus meant as an early report.

In contrast to robustness verification of trained ML models, a widespread and appreciated research topic in recent years [135], the data pre-processing stage of the ML pipeline has received little attention. As raw data is often inconsistent or incomplete, DS and ML pre-processing programs apply transformations to polish it to the point where it can be visualized or used for training. This is a tedious and often trial-by-error procedure, and as a result, such code is often seen as a rarely tested one-off script, weakening the ML pipeline. Errors and inconsistencies at this stage can silently propagate downwards in the ML chain, leading to incorrect conclusions and below-par models [32]. As an example, “Growth in a Time of Debt” [114] by Reinhart and Rogoff has received notable attention in political discussions regarding the effectiveness of austerity policies w.r.t. the public debt of the country applying them. The paper was then criticized as the “selective exclusion of available data, coding errors and inappropriate weighting of summary statistics led to serious miscalculations that inaccurately represent the relationship between public debt and GDP growth among 20 advanced economies” [78].

Verification of DS and ML pre-processing notebooks can take different directions. A first axis is the inference of input data usage, that is deducing what portions of the

¹<https://jupyter.org/>.

input data are used by the program to produce a specific result. This is crucial to ensure that no bias is introduced in the analysis of the input data. Equivalently, such information can be used to check if unwanted data was used while producing a result. For instance, some studies might need to be independent of features such as gender or ethnicity. Furthermore, ML data leakages (i.e., sharing of information between the training and test datasets, one of the problems tackled in [128]) could be derived from the same results. Lastly, input data usage could also be used to automate data provenance reports [102]. A second line of work aims at highlighting data transformations introducing bias or skews [144]. Transformations like filter, concatenation, or value replacement can introduce bias and skew within the data. Concretely, consider the simple example of the dataset containing age and salary demographics. If a correlation exists between the two, filtering the data by salary introduces a bias w.r.t. the age. A third objective is inferring the shape of input data. DS notebooks are modeled after the structure of input data, and each transformation implicitly assumes that such structure is valid (e.g., that a column with a given name exists). As some of these notebooks are executed periodically to gain insights from aggregated data sources with ever-changing formats, preemptive analyses can run on the fetched data to ensure that the execution will not fail due to a format change.

Regardless of the verification direction, JuPyter notebooks are challenging to analyze. For starters, code comes in blocks that can be executed in any order with repetitions. Furthermore, transformations are performed through calls to library functions: a large number of them exist, each with complex and possibly overlapping semantics, and whose code is not always available. These add to the already challenging features of PYTHON, that shares most of the characteristics of dynamic languages: dynamic typing and evaluation, mutable and runtime-modifiable program structure, and high-level syntax with complex semantics that introduces innate data structures and operators over them. This chapter proposes an abstract interpretation approach to simplify the implementation of verification techniques for JuPyter notebooks containing DS programs. We propose an abstract domain that tracks transformations made to *dataframes*, that is, the in-memory tables containing the input data, in a unique graph. The latter contains atomic transformations, such as column access or filtering, as nodes that are linked by edges encoding the order in which they are applied. The final graph produced as post-state of the last instruction of the notebook is effectively a control flow graph containing only dataframe transformations. As a consequence, analyses that aim at determining properties of such transformations, such as the ones discussed earlier in this chapter, can be implemented as fixpoint algorithms over this control flow graph. As this is an ongoing work, we focus on a subset of the transformations offered by *pandas*², the de-facto standard for processing raw data in DS and ML, providing concrete and abstract semantics for them.

This chapter is structured as follows. Section 5.1 discusses related work, and is fol-

²<https://pandas.pydata.org/>.

lowed by the definition of the concrete domain we aim to abstract in Section 5.2. Section 5.3 defines the abstraction for dataframe values, presenting the domain with concretization, and abstraction functions, with a focus on its semantics in Section 5.3.1. We then explore a first use-case in Section 5.4, where we use our abstraction for inferring the shape of the dataframes used by a program. We then conclude with a preliminary experiment on a real DS notebook in Section 5.5.

5.1 Related work

Obtaining formal guarantees on the safety and fairness of ML models has been a subject of recent widespread interest [135]. Our work builds upon this large ecosystem and proposes a verification framework for the data pre-processing stage of the ML pipeline. The closest body of work is [128] which also analyzes DS notebooks along with their peculiar execution semantics and proposes an abstraction to detect data leakage. Our approach towards the shape inference of input data follows the work of [134] and extends it to support inputs to programs which contain datasets. Similarly, our objective of inferring input data usage directly derives from the compound data structure usage analysis presented in [136] and adds the ability to track the usage of selections of datasets. The objective of detecting bias/skew introduction is inspired by the `minspect` tool proposed in [75]. This tool builds a directed acyclic graph (DAG) of operations (like filters or projections) applied to the data by analyzing the code and using framework-specific backends (like `scikit-learn`). After analyzing the DAG, it suggests potential sources of bias/skew. Although this is promising, it only places syntactic checks and cannot concretely detect which operations cause these problems. Lastly, [102] is an automated data provenance tracking system for PYTHON scripts. However, this tool requires executing the code which might not always be feasible when large datasets are involved, or if those are not statically available.

5.2 A concrete semantics for transformations

Transformations of dataframes are performed by invoking library functions. Hence, no PYTHON constructs inherently require special semantics. Still, taking into account the effects of these functions on the program state is challenging: their code (when available) can be complex and confusing for a static analyzer. We thus indirectly model their semantics through auxiliary constructs. These represent atomic transformations that can be composed to obtain the semantics of several functions, providing support for libraries beyond `pandas`.

We thus extend the PYTHON syntax with expressions corresponding to atomic transformations. A list of the added expressions is visible in Figure 5.1, where op is a comparison operator (`==`, `!=`, `>`, `>=`, `<`, or `<=`), and f is a function that trans-

$$e \in E ::= \dots \mid df_read(e) \mid df_access(e, e) \mid df_concat(e) \mid df_transform(e, f) \mid df_filter(e, e \text{ op}, e) \mid df_assign(e, e, e)$$

Figure 5.1: Expressions added to the PYTHON language

forms the values contained in the dataframe. Concrete semantics for PYTHON have already been studied [99, 110, 77, 85]. We assume such semantics to be defined as a collecting semantics of the form:

$$\mathbb{S}[\text{st}] : \wp(\mathcal{D} \times \mathcal{S}) \rightarrow \wp(\mathcal{D} \times \mathcal{S})$$

where the state is, for the sake of convenience, split into an environment \mathcal{D} mapping identifiers³ to dataframes and the remaining state \mathcal{S} . Such function relies on the semantics of expressions:

$$\mathbb{E}[e] : \wp(\mathcal{D} \times \mathcal{S}) \rightarrow \wp(\mathcal{D} \times \mathcal{S} \times \mathbb{V})$$

We thus assume expressions to possibly lead to side effects, and to evaluate to any possible value in \mathbb{V} (that is, either a memory address or a primitive value). As the atomic transformations are (i) side effect-free, and (ii) evaluated on one state at a time, we define their semantics as:

$$\mathbb{E}[e](\{(D_1, S_1), \dots, (D_n, S_n)\}) = \bigcup_i \{ (D_i, S_i, \bullet) \mid \bar{\mathbb{E}}[e](D_i, S_i) = \bullet, 1 \leq i \leq n \}, n \in \mathbb{N}^\infty$$

where $\bar{\mathbb{E}}[e] : \mathcal{D} \times \mathcal{S} \rightarrow \mathbb{V}$ evaluates the side effect-free semantics of the expression e .

We then define a concrete dataframe as the tuple:

$$d \in \mathbb{D} = (S = \sigma, C = \langle \sigma_1, \dots, \sigma_n \rangle, R = \langle \langle v_1^1, \dots, v_n^1 \rangle, \dots, \langle v_1^m, \dots, v_n^m \rangle \rangle) \quad n, m \in \mathbb{N}, \sigma, \sigma_i \in \Sigma^*$$

C is thus the sequence of column names and R is the sequence of rows, with a value in \mathbb{V} for each column. Individual values on the i -th row can be indexed using both column names (v_σ^i) or column indexes (v_j^i). Moreover, S represents the source (file, url, ...) of the data (for dataframes constructed programmatically, $S = \epsilon$). The concrete semantics of the atomic transformations is captured in Figure 5.2, where:

- \perp is the error state;
- the error state generated when an expression evaluates to an unexpected value (for instance, if $\bar{\mathbb{E}}[s](\mathcal{D}, \mathcal{S}) \notin \Sigma^*$) is omitted;
- d_σ is the dataframe containing the data from the source (file, URL, ...) σ ;

³Identifiers can be either program variables or memory addresses, depending on how the semantics is formalized.

$$\begin{aligned}
\bar{\mathbb{E}}[\text{df_read}(s)](\mathcal{D}, S) &= d_\sigma \\
\bar{\mathbb{E}}[\text{df_access}(\text{df}, \text{cl})](\mathcal{D}, S) &= \begin{cases} (S, \langle \sigma'_1, \dots, \sigma'_c \rangle, [R]_{\sigma'_1, \dots, \sigma'_c}) & \text{if } \forall \sigma'_i \in \langle \sigma'_1, \dots, \sigma'_c \rangle \\ & . \sigma'_i \in C \\ \perp & \text{otherwise} \end{cases} \\
\bar{\mathbb{E}}[\text{df_transform}(\text{df}, f)](\mathcal{D}, S) &= (S, \langle \sigma'_1, \dots, \sigma'_c \rangle, R' = \langle v^i_j = f(v^i_j) \rangle) \\
\bar{\mathbb{E}}[\text{df_concat}(\text{dl})](\mathcal{D}, S) &= (\epsilon, C_c = C_1 \diamond \dots \diamond C_l, R_1 \bullet_{C_c} \dots \bullet_{C_c} R_l) \\
\bar{\mathbb{E}}[\text{df_filter}(\text{df}, s, \text{op}, e)](\mathcal{D}, S) &= (S, C, \langle \langle v_1, \dots, v_n \rangle \in R \mid v_\sigma \text{ op } v \rangle) \\
\bar{\mathbb{E}}[\text{df_assign}(\text{df}, \text{cl}, \text{df}')](\mathcal{D}, S) &= \begin{cases} (S, C, R'') & \text{if } \forall \sigma'_i \in \langle \sigma'_1, \dots, \sigma'_c \rangle . \sigma'_i \in C \\ \perp & \text{otherwise} \end{cases} \\
&\quad v''^i_\sigma = \begin{cases} v^i_\sigma & \text{if } \forall \sigma \notin \langle \sigma'_1, \dots, \sigma'_c \rangle \\ v^i_\sigma & \text{otherwise} \end{cases} \\
\text{where } \bar{\mathbb{E}}[s](\mathcal{D}, S) &= \sigma \in \Sigma^* \\
\bar{\mathbb{E}}[e](\mathcal{D}, S) &= v \in \mathbb{V} \\
\bar{\mathbb{E}}[\text{df}](\mathcal{D}, S) &= d = (S, C, R) \in \mathbb{D} \\
\bar{\mathbb{E}}[\text{cl}](\mathcal{D}, S) &= \langle \sigma'_1, \dots, \sigma'_c \rangle, c \in \mathbb{N}, \sigma_i \in \Sigma^* \\
\bar{\mathbb{E}}[\text{dl}](\mathcal{D}, S) &= \langle d_1, \dots, d_l \rangle, l \in \mathbb{N}, d_i \in \mathbb{D}
\end{aligned}$$

Figure 5.2: Concrete semantics of the atomic transformations

- $[R]_{\sigma'_1, \dots, \sigma'_c}$ is a projection of each row in R on the columns $\sigma'_1, \dots, \sigma'_c$;
- the \diamond operator performs list concatenation removing duplicates;
- the \bullet_C operator performs rows concatenation after padding the rows w.r.t. the column set C : if a row does not have a value for column σ , a NaN is inserted.

5.2.1 Obtaining the semantics of Python code

Our atomic transformations are not directly part of the language: they serve as a mean to express the semantics of library functions in terms of smaller atomic constructs. Hence, when analyzing calls to target libraries, the appropriate set of transformations must be identified and their abstract semantics can be composed to produce the post-state of the call. Figure 5.3 depicts an example of how calls to pandas functions that manipulate dataframes can be mimicked using our atomic constructs, where the original PYTHON code manipulating two dataframes is shown in Figure 5.3a. Each individual call has its counterpart in the list of atomic transformations we defined, and the program is thus intuitively equivalent to the one in Figure 5.3b. Specifically, the program starts by reading file `italy.csv` through `pandas.read_csv` method (mimicked using `df_read`), storing the resulting dataframe into variable

```

1 import pandas as pd
2 df1 = pd.read_csv('italy.csv')
3 df1['birth'] = pd.to_datetime(df1['birth'])
4 df2 = pd.read_csv('france.csv')
5 df2 = df2['age' < 50]
6 df3 = pd.concat([df1, df2])

```

(a) Minimal PYTHON DS program

```

1 import pandas as pd
2 df1 = df_read_file('italy.csv')
3 df_assign(df1, ['birth'], df_transform(df_access(df1, ['birth']),
    to_datetime))
4 df2 = df_read_file('france.csv')
5 df2 = df_filter(df2, 'age', <, 50)
6 df3 = df_concat([df1, df2])

```

(b) Instrumentation of the PYTHON program

Figure 5.3: Example PYTHON DS program and its instrumentation

`df1` at line 2. Next, the contents of column `birth` (whose access is substituted with `df_access`) are transformed into `datetime` objects exploiting `pandas.to_dataframe` (rewritten into `df_transform`) and stored back into the column through the assignment (modeled with `df_assign`) at line 3. After reading `france.csv` at line 4, all of its rows whose `age` columns contain values greater than or equal to 50 are discarded (through the conditional expression used inside square brackets, instrumented with `df_filter`) at line 5. Finally, the two dataframes are joined together along the rows axis using `pandas.concat` (replaced by `df_concat`) at line 6.

Note that this approach perfectly fits LiSA's infrastructure: implementations of library functions can be provided through *native CFGs*, and the semantics of its only node will rewrite them into symbolic expressions representing the atomic transformations. The code of Figure 5.3b will be used as a running example to explain our domain.

5.3 The dataframe graph domain

We present here an abstraction able to capture the structure of the dataframes manipulated in PYTHON code. Intuitively, we employ a graph structure to keep track of all operations that involve dataframes, with edges encoding the order in which they are performed. The graph thus represents the state of each dataframe at a given program point. Nodes of this graph can be pointed by variables of the program. The latter thus refers to the dataframe corresponding to the sub-graph obtained with a backward DFS starting from the node. We adopt a two-level mapping: instead of directly pointing to nodes of the graph, program variables point to labels, and the latter are mapped to the nodes. This enables simple handling of dataframe aliasing (i.e., two variables will be mapped to the same label) and updates (i.e., by changing the nodes pointed by a label, we indirectly update all variables pointing to the label).

As our domain is meant to be an abstraction of \mathcal{D} (that is, $\wp(\text{Var} \rightarrow \mathcal{D})$), we

denote it with $\mathcal{D}^\#$ and we define it w.r.t. an auxiliary domain $\mathcal{A}^\#$, providing an abstraction \square for the remaining portion \mathcal{S} of the concrete state. The abstract state is thus a pair $(\square, d^\#)$, with $d^\# \in \mathcal{D}^\#$. Hence, our definition is parametric to $\mathcal{A}^\#$, that is only required to provide the following abstract transformers:

- $\mathbb{E}_{\mathcal{A}^\#}^\# \llbracket \mathbf{s} \rrbracket (\square, d^\#) = (\square_1, d_1^\#, \{\sigma_1, \dots, \sigma_k\})$, $k \in \mathbb{N}$, evaluating a string expression s to a set of abstract elements $\{\sigma_1, \dots, \sigma_k\}$;
- $\mathbb{E}_{\mathcal{A}^\#}^\# \llbracket \mathbf{cl} \rrbracket (\square, d^\#) = (\square_1, d_1^\#, \{\langle \sigma_1^1, \dots, \sigma_m^1 \rangle, \dots, \langle \sigma_1^k, \dots, \sigma_m^k \rangle\})$, $m, k \in \mathbb{N}$, that evaluates a list of string expressions cl to $\{\langle \sigma_1^1, \dots, \sigma_m^1 \rangle, \dots, \langle \sigma_1^k, \dots, \sigma_m^k \rangle\}$, where each $\langle \sigma_1^i, \dots, \sigma_m^i \rangle$, $1 \leq i \leq k$ is a possible abstraction for the elements of the list;
- $\mathbb{E}_{\mathcal{A}^\#}^\# \llbracket \mathbf{dl} \rrbracket (\square, d^\#) = (\square_1, d_1^\#, \{\langle \ell_1^1, \dots, \ell_m^1 \rangle, \dots, \langle \ell_1^k, \dots, \ell_m^k \rangle\})$, $m, k \in \mathbb{N}$, providing the abstraction of a list of dataframe expressions dl as $\{\langle \ell_1^1, \dots, \ell_m^1 \rangle, \dots, \langle \ell_1^k, \dots, \ell_m^k \rangle\}$, where each $\langle \ell_1^i, \dots, \ell_m^i \rangle$, $1 \leq i \leq k$ is a possible abstraction for the elements of the list;
- $\mathbb{E}_{\mathcal{A}^\#}^\# \llbracket \mathbf{x} \rrbracket (\square, d^\#) = (\square_1, d_1^\#, \{x_1, \dots, x_n\})$, $n \in \mathbb{N}$ evaluating the left-hand side of an assignment x to a set of identifiers $\{x_1, \dots, x_n\}$.

We begin defining $\mathcal{D}^\#$ by introducing the building blocks for our domain. We start from the graph structure, where a graph $g^\# = (N, E)$ is composed of a set of nodes $N \subseteq \mathcal{N}$ and a set of edges $E \subseteq \mathcal{E}$. Elements of \mathcal{N} are:

- $\text{read}(\sigma)$, representing the initialization of a dataframe with the contents of file σ ;
- $\text{access}(\sigma_1, \dots, \sigma_k)$, symbolizing the access to columns $\sigma_1, \dots, \sigma_k$, $k \in \mathbb{N}$;
- $\text{transform}(f)$, transforming values through an auxiliary function f ;
- concat , representing a concatenation of multiple dataframes;
- $\text{filter}(\sigma, op, v)$, portraying the selection of rows where $v_\sigma op v$ holds;
- $\text{assign}(\sigma_1, \dots, \sigma_k)$, corresponding to the assignment of columns $\sigma_1, \dots, \sigma_k$, $k \in \mathbb{N}$ to a new value.

where σ and v are string and value abstractions in $\mathcal{A}^\#$, $op \in \{=, \neq, >, \geq, <, \leq\}$ and f is the signature of a PYTHON function. Instead, elements of \mathcal{E} are (with $n, n' \in \mathcal{N}$, $i \in \mathbb{N}$):

- $n \rightarrow n'$ is a simple edge, encoding the sequential order of operations;
- $n \rightsquigarrow_i n'$ is a concatenation edge, indicating that n is the i -th dataframe taking part in the concatenation that builds n' (note that n' can have more incoming concatenation edges using the same i , indicating multiple candidates for the same index);

- $n \rightarrow n'$ is an assign edge, portraying the usage of n as the right-hand side of the assignment depicted in n' (once more, n' can have more incoming assign edges, indicating multiple candidates for the right-hand side).

Note that, as $\langle \wp(\mathcal{N}), \subseteq, \cup, \cap, \emptyset, \mathcal{N} \rangle$ and $\langle \wp(\mathcal{E}), \subseteq, \cup, \cap, \emptyset, \mathcal{E} \rangle$ are complete lattices built over powersets, each graph $g^\#$ is an element of the Cartesian product $\mathcal{G}^\# = \langle \wp(\mathcal{N}) \times \wp(\mathcal{E}), \subseteq, \cup, \cap, \emptyset \times \emptyset, \mathcal{N} \times \mathcal{E} \rangle$, that is a complete lattice itself.

We now proceed defining the mapping from labels to graph nodes. A label $\ell \in \mathcal{L}$ is an arbitrary synthetic identifier that serves as an abstract name for a set of nodes in \mathcal{N} , where \mathcal{L} is the finite set of all possible labels. While we do not impose any specific structure on \mathcal{L} , a common characterization of labels is to have one for each program point. We denote as $\bar{\ell}$ a fresh and unused label. $\mathcal{D}^\#$ contains a function $\mathcal{L} \rightarrow \wp(\mathcal{N})$ from labels to sets of nodes. As the co-domain is composed by elements of the complete lattice of nodes, each map $l^\#$ is an element of the functional lift $\mathcal{L}^\# = \langle \mathcal{L} \rightarrow \wp(\mathcal{N}), \subseteq, \cup, \cap, \perp, \top \rangle$, that is still a complete lattice.

Lastly, we define the mapping from variables to labels. A map $\text{Var} \rightarrow \wp(\mathcal{L})$ is part of $\mathcal{D}^\#$, keeping track of which possible labels a variable can refer to. Notice that, depending on the analyzer's infrastructure, variables can correspond to abstract memory locations or program variables. Similarly to $\mathcal{L}^\#$, the co-domain is composed of elements of the complete lattice $\langle \wp(\mathcal{L}), \subseteq, \cup, \cap, \emptyset, \mathcal{L} \rangle$: each function $v^\#$ is thus an element of the functional lift (and hence complete lattice) $\mathcal{V}^\# = \langle \text{Var} \rightarrow \wp(\mathcal{L}), \subseteq, \cup, \cap, \perp, \top \rangle$.

We can now define $\mathcal{D}^\#$ as the Cartesian product $\langle \mathcal{V}^\# \times \mathcal{L}^\# \times \mathcal{G}^\#, \subseteq, \cup, \cap, \perp, \top \rangle$ that, once more, is a complete lattice. In the following, we will denote elements of $\mathcal{D}^\#$ as either $d^\#$ or $(v^\#, l^\#, g^\#)$, depending on what is more convenient.

One concern with infinite lattices such as $\mathcal{D}^\#$ is the convergence of fixpoint iterations over them. As $\mathcal{G}^\#$ intuitively does not satisfy ACC⁴, a widening operator is required. As, in our experience, the DS notebooks that this domain targets mostly contain sequential code with very few loops that stabilize in few iterations, we employ the naive widening $d_1^\# \nabla d_2^\# = \top$, $\forall d_1^\#, d_2^\# \in \mathcal{D}^\#$. With such an operator we ensure termination of the analysis, and we leave the study of a more precise widening operator as future work.

Example 5.3.1. Figure 5.4 reports the $d^\#$ instance abstracting the code of Figure 5.3b when a simple constant propagation abstraction is chosen as auxiliary domain $\mathcal{A}^\#$. For the sake of clarity, nodes of $g^\#$ are enriched with a numerical identifier on the top-left corner to easily identify them. Such identifiers are used in the co-domain of $l^\#$ to represent them. We show how this graph is constructed step by step in Section 5.3.1.

The connection between \mathcal{D} and $\mathcal{D}^\#$ is established by the abstraction function α

⁴As \mathcal{N} and \mathcal{E} are infinite sets, one can keep adding new nodes and edges without the graph ever stabilizing.

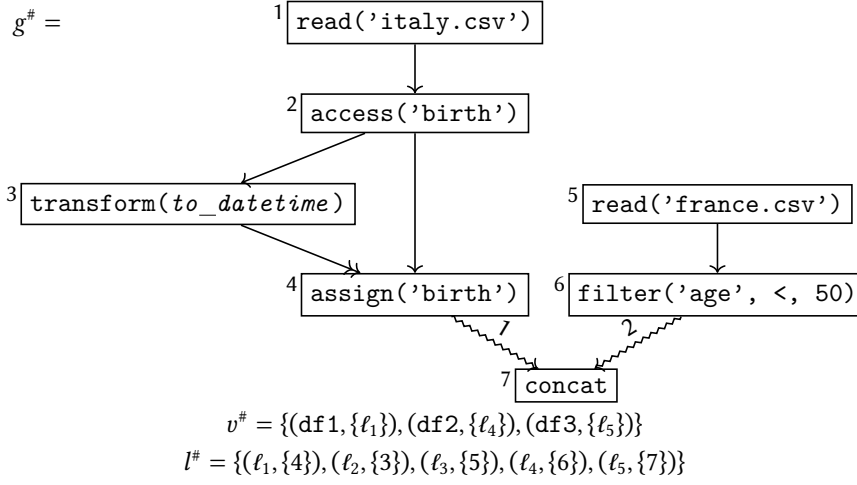


Figure 5.4: Example $d^\#$ abstracting the code of Figure 5.3b

and the concretization function γ . A set $\{\bar{d}_1, \dots, \bar{d}_m\}, \bar{d}_i \in \mathcal{D}, 1 \leq i \leq m, m \in \mathbb{N}^\infty$ can be abstracted to an element $d^\# \in \mathcal{D}^\#$ through function $\alpha : \wp(\mathcal{D}) \rightarrow \mathcal{D}^\# \equiv \wp(\text{Var} \rightarrow \mathcal{D}) \rightarrow (\mathcal{V}^\# \times \mathcal{L}^\# \times \mathcal{G}^\#)$. Such function is defined as the lub of the abstractions of each individual \bar{d} :

$$\alpha(\{\bar{d}_1, \dots, \bar{d}_m\}) = \bigcup_{\bar{d} \in \{\bar{d}_1, \dots, \bar{d}_m\}} \bar{\alpha}(\bar{d})$$

where $\bar{\alpha} : \mathcal{D} \rightarrow \mathcal{D}^\#$ is defined as follows:

$$\begin{aligned} \bar{\alpha}(\bar{d}) &= (v^\#, l^\#, g^\#), \text{ where } \bar{d} = \{(v_1, d_1), \dots, (v_k, d_k)\}, 1 \leq i \leq k, k \in \mathbb{N} \\ (g_i^\#, n_i) &= \text{shape}(d_i), g^\# = \bigcup_i g_i^\#, \\ l^\# &= \{(\ell_1, \{n_1\}), \dots, (\ell_k, \{n_k\})\}, \\ v^\# &= \{(v_1, \{\ell_1\}), \dots, (v_k, \{\ell_k\})\} \end{aligned}$$

The abstraction of a single dataframe map exploits $\text{shape} : \mathcal{D} \rightarrow \mathcal{G}^\# \times \mathcal{N}$, an auxiliary function that extracts the shape of a concrete dataframe into a single-path graph, returning the graph itself and its unique leaf:

$$\text{shape}(d) = \begin{cases} (\{n, n'\}, \{n \rightarrow n'\}, n') & \text{if } S \neq \epsilon \\ (\{n'\}, \emptyset, n') & \text{otherwise} \end{cases}$$

$$\text{where } n = \text{read}(\alpha_{\mathcal{A}^\#}(S)), n' = \text{access}(\alpha_{\mathcal{A}^\#}(C))$$

The abstraction of a set of states is thus the union of the abstraction of each individual state, generated by creating the graph $g^\#$ containing the shape (that is, the access to all columns C optionally preceded by the reading of source S) of all existing dataframes, having each variable refer to the corresponding node in $g^\#$.

Theorem 1. α is a complete join-preserving function. Formally:

$$\forall \mathcal{D}' \subseteq \mathcal{D}. \alpha(\bigcup \mathcal{D}') = \bigcup^{\times} \alpha(\mathcal{D}')$$

Proof.

$$\begin{aligned} & \alpha(\bigcup \mathcal{D}') \\ = & \alpha(\bigcup \{ \{\bar{d}_1^1, \dots, \bar{d}_{m_1}^1\}, \dots, \{\bar{d}_1^k, \dots, \bar{d}_{m_k}^k\} \}) && \{ \text{def. } \mathcal{D}' \} \\ = & \alpha(\{\bar{d}_1^1, \dots, \bar{d}_{m_1}^1, \dots, \bar{d}_1^k, \dots, \bar{d}_{m_k}^k\}) && \{ \text{def. } \cup \} \\ = & \bar{\alpha}(\bar{d}_1^1) \check{\cup} \dots \check{\cup} \bar{\alpha}(\bar{d}_{m_1}^1) \check{\cup} \dots \check{\cup} \bar{\alpha}(\bar{d}_1^k) \check{\cup} \dots \check{\cup} \bar{\alpha}(\bar{d}_{m_k}^k) && \{ \text{def. } \alpha \} \\ = & \alpha(\{\bar{d}_1^1, \dots, \bar{d}_{m_1}^1\}) \check{\cup} \dots \check{\cup} \alpha(\{\bar{d}_1^k, \dots, \bar{d}_{m_k}^k\}) && \{ \text{def. } \alpha \} \\ = & \bigcup^{\times} \alpha(\mathcal{D}') && \{ \text{def. } \mathcal{D}' \} \end{aligned}$$

□

An abstract element $\mathcal{D}^\#$ can instead be concretized to a set $\{\bar{d}_1, \dots, \bar{d}_m\}$, $\bar{d}_i \in \mathcal{D}$ through function $\gamma : \mathcal{D}^\# \rightarrow \wp(\mathcal{D}) \equiv (\mathcal{V}^\# \times \mathcal{L}^\# \times \mathcal{G}^\#) \rightarrow \wp(\text{Var} \rightarrow \mathbb{D})$, defined as:

$$\gamma(d^\#) = \bigcup^{\times} \{ \mathcal{D}' \in \wp(\mathcal{D}) \mid \alpha(\mathcal{D}') \check{\subseteq} d^\# \}$$

since α is a complete join-preserving function. Note that, as reported in Section 2.2.2, this also induces the Galois connection $\langle \wp(\mathcal{D}), \subseteq \rangle \xleftrightarrow[\alpha]{\gamma} \langle \mathcal{D}^\#, \check{\subseteq} \rangle$.

5.3.1 Abstract semantics

As mentioned earlier in this section, we define $\mathcal{D}^\#$ w.r.t. an auxiliary domain $\mathcal{A}^\#$ that is exploited for the evaluation of non-dataframe expressions, and the program state is modeled as the pair $(\square, d^\#)$. The abstract semantics can then be defined as:

$$\mathbb{S}^\# \llbracket \text{st} \rrbracket : \square \times \mathcal{D}^\# \rightarrow \square \times \mathcal{D}^\#$$

that in turn relies on the abstract semantics of expressions:

$$\mathbb{E}^\# \llbracket e \rrbracket : \square \times \mathcal{D}^\# \rightarrow \square \times \mathcal{D}^\# \times \dagger$$

where \dagger is the abstract value resulting from the evaluation of e , if any. As for the concrete case, we let the abstract expression semantics return a new state $(\square, d^\#)$ to model possible side effects of the expression's evaluation. When $e = \text{df}$, $\dagger = \wp(\mathcal{L})$, otherwise it is an abstraction produced by $\mathcal{A}^\#$. A dataframe expression thus evaluates to a set of labels, identifying nodes representing the dataframes that correspond to e . In the following, we define the semantics of expressions that involve dataframes, and we rely on the abstract semantics of $\mathcal{A}^\#$ for the remaining ones.

Note that, thanks to the GC $\langle \wp(\mathcal{D}), \subseteq \rangle \xleftrightarrow[\alpha]{\gamma} \langle \mathcal{D}^\#, \dot{\subseteq} \rangle$, soundness of the abstract semantics can be proven by showing that $\forall x \in X. \alpha(f(x)) \dot{\subseteq}_x \bar{f}(\alpha(x))$. As the abstractions presented here are meant as an early draft, we leave the soundness proofs as future work.

Assignment. Whenever the right-hand side of an assignment is a dataframe expression df , $v^\#$ must be updated for the corresponding variable. The assignment's semantics is thus defined as:

$$\mathbb{E}^\# \llbracket x = df \rrbracket (\square, d^\#) \triangleq (\square_2, (v_3^\#, l_2^\#, g_2^\#), \{\ell_1, \dots, \ell_w\})$$

where:

- the right-hand side is first evaluated, producing the set of labels corresponding to the dataframe to be assigned: $\mathbb{E}^\# \llbracket df \rrbracket (\square, d^\#) = (\square_1, d_1^\#, \{\ell_1, \dots, \ell_w\})$;
- the left-hand side is evaluated using $\mathcal{A}^\#$'s semantics, resulting in a set of variables that must be assigned: $\mathbb{E}_{\mathcal{A}^\#}^\# \llbracket x \rrbracket (\square_1, d_1^\#) = (\square_2, (v_2^\#, l_2^\#, g_2^\#), \{x_1, \dots, x_n\})$;
- finally, we store the new mapping for each variable of the left-hand side evaluation: $v_3^\# = v_2^\# [x_i \mapsto \{\ell_1, \dots, \ell_w\}, \forall x_i \in \{x_1, \dots, x_n\}]$.

Example 5.3.2. When evaluating the assignment at line 1 of Figure 5.3b, the semantics stores the label pointing to the read node (whose creation is dictated by the abstract semantics of df_read). $v^\#$ is thus extended with the pair $(df1, \{\ell_1\})$, where $\{\ell_1\}$ is the label returned by the semantics of df_read .

Variable evaluation. Whenever a variable is referenced throughout the program, our semantics must evaluate it to the corresponding labels if it refers to a dataframe. The remaining variables are instead handled by $\mathcal{A}^\#$. Thus, when x resolves to a dataframe, the semantics is formalized as:

$$\mathbb{E}^\# \llbracket x \rrbracket (\square, d^\#) \triangleq (\square, d^\#, v^\#(x))$$

Example 5.3.3. When evaluating the concatenation at line 8 of Figure 5.3b, $df1$ and $df2$ must be first resolved to the dataframes they represent before proceeding with the evaluation. $\mathbb{E}^\# \llbracket df1 \rrbracket (\square, d^\#)$ thus returns $(\square, d^\#, \{\ell_1\})$ while $\mathbb{E}^\# \llbracket df2 \rrbracket (\square, d^\#)$ returns $(\square, d^\#, \{\ell_4\})$ instead, as the two variables are mapped to $\{\ell_1\}$ and $\{\ell_4\}$ in $v^\#$, respectively.

Dataframe initialization. When dataframes are initialized with the contents of an external resource through df_read , the operation cannot be precisely modeled

statically as the contents of the resource are unknown at analysis time. We thus symbolically record the source of the data by adding `read` nodes to the graph. Instead, as this is an initialization of a new dataframe, no edges are added. The semantics then returns a unique label that points to the freshly added nodes. Formally:

$$\mathbb{E}^{\#} \llbracket df_read(s) \rrbracket (\square, d^{\#}) \triangleq (\square_1, (v_1^{\#}, l_2^{\#}, g_2^{\#}), \{\bar{l}\})$$

where:

- the resource identifier is abstracted through $\mathcal{A}^{\#}$'s semantics, yielding a set of abstract strings: $\mathbb{E}_{\mathcal{A}^{\#}}^{\#} \llbracket s \rrbracket (\square, d^{\#}) = (\square_1, (v_1^{\#}, l_1^{\#}, g_1^{\#}), \{\sigma_1, \dots, \sigma_k\})$;
- we instantiate a node for each abstract string: $n_i = \text{read}(\sigma_i), \forall \sigma_i \in \{\sigma_1, \dots, \sigma_k\}$;
- all new nodes are stored as the image of a fresh label: $l_2^{\#} = l_1^{\#} \cup \{\bar{l}, \{n_1, \dots, n_k\}\}$;
- the graph is extended with the new nodes: $g_2^{\#} = g_1^{\#} \dot{\cup} (\{n_1, \dots, n_k\}, \emptyset)$.

Example 5.3.4. When the `df_read` at line 1 of Figure 5.3b is evaluated, $g^{\#}$ is still empty. As $\mathcal{A}^{\#}$ is set to constant propagation, there is a unique abstraction of the resource to be read, and thus a single `read('italy.csv')` node is added to $g^{\#}$. The semantics then extends $l^{\#}$ with a fresh label l_1 , that is mapped to a set containing the node itself, and is used as the only label in the result of the evaluation.

Column access. The `df_access` transformation accesses a set of columns of the target dataframe. As this operation does not create a new dataframe, it is modeled as an in-place operation, directly affecting the nodes pointed by the labels of its argument. The semantics adds a new node for each possible abstraction of the column list, connecting it to the nodes pointed by the first argument. It is defined as follows:

$$\mathbb{E}^{\#} \llbracket df_access(df, cl) \rrbracket (\square, d^{\#}) \triangleq (\square_2, (v_2^{\#}, l_3^{\#}, g_3^{\#}), \{\ell_1, \dots, \ell_w\})$$

where:

- the dataframe that receives the access is first evaluated, determining the set of labels that correspond to it: $\mathbb{E}^{\#} \llbracket df \rrbracket (\square, d^{\#}) = (\square_1, d_1^{\#}, \{\ell_1, \dots, \ell_w\})$;
- the column list is evaluated by $\mathcal{A}^{\#}$, producing a set of abstractions for the names: $\mathbb{E}_{\mathcal{A}^{\#}}^{\#} \llbracket cl \rrbracket (\square_1, d_1^{\#}) = (\square_2, (v_2^{\#}, l_2^{\#}, g_2^{\#}), \{\langle \sigma_1^1, \dots, \sigma_m^1 \rangle, \dots, \langle \sigma_1^k, \dots, \sigma_m^k \rangle\})$;
- an access node is created for each possible abstraction of the list of column names: $n_i = \text{access}(\sigma_1^i, \dots, \sigma_m^i), \forall \langle \sigma_1^i, \dots, \sigma_m^i \rangle \in \{\langle \sigma_1^1, \dots, \sigma_m^1 \rangle, \dots, \langle \sigma_1^k, \dots, \sigma_m^k \rangle\}$;
- all labels resulting from the evaluation of `df` are remapped to the new nodes, symbolizing the side effect: $l_3^{\#} = l_2^{\#}[\ell_j \mapsto \{n_1, \dots, n_k\}, \forall \ell_j \in \{\ell_1, \dots, \ell_w\}]$;

- the graph is extended with the new nodes, each connected those of df with normal edges: $g_3^\# = g_2^\# \dot{\cup} (\{n_1, \dots, n_k\}, \{n' \rightarrow n_i \mid n' \in l_2^\#(\ell_j), \forall \ell_j \in \{\ell_1, \dots, \ell_w\}, \forall n_i \in \{n_1, \dots, n_k\}\})$.

Example 5.3.5. When the df_access at line 4 of Figure 5.3b is evaluated, $df1$ is first processed, producing $\{\ell_1\}$ as abstract value. Then, the evaluation of the column names yields a unique constant list $\langle 'birth' \rangle$, resulting in the creation of a single node $access('birth')$. The graph is then extended adding (i) the newly created node with id 2, and (ii) a normal edge connecting nodes 1 and 2. Furthermore, ℓ_1 is remapped to $\{2\}$ inside the resulting $l^\#$ (the mapping is not visible in Figure 5.4 as evaluation of following statements overwrites it).

Value transformation. Tracking transformations of dataframe values can be problematic, as the functions carrying the transformation must be summarized somehow. Instead, we provide a lightweight semantics that records the signature of the used function inside a node of the graph, deferring further reasoning to successive abstractions. As this is not an in-place operation, the semantics creates a new label that is mapped to the transformation node, and the latter is connected to nodes representing the target dataframe. The label is then returned as the unique result of the evaluation. Formally:

$$\mathbb{E}^\# \llbracket df_transform(df, f) \rrbracket (\square, d^\#) \triangleq (\square_1, (v_1^\#, l_2^\#, g_2^\#), \{\bar{\ell}\})$$

where:

- the dataframe that is being transformed is evaluated first, producing the set of labels to be targeted: $\mathbb{E}^\# \llbracket df \rrbracket (\square, d^\#) = (\square_1, (v_1^\#, l_1^\#, g_1^\#), \{\ell_1, \dots, \ell_w\})$;
- a unique transformation node is created, embedding the function's signature in it: $n = transform(f)$;
- the new node is used as the image of a fresh label: $l_2^\# = l_1^\# \cup \{\bar{\ell}, \{n\}\}$;
- the graph is extended, adding the new node together with normal edges between every node pointed by a label of df and the transformation node itself: $g_2^\# = g_1^\# \dot{\cup} (\{n\}, \{n' \rightarrow n \mid n' \in l_1^\#(\ell_j), \forall \ell_j \in \{\ell_1, \dots, \ell_w\}\})$.

Example 5.3.6. When the $df_transform$ at line 4 of Figure 5.3b is evaluated, our semantics first evaluates the column access, producing $\{\ell_1\}$ as shown earlier. The semantics then (i) creates a unique $transform(to_datetime)$ node, (ii) adds it to the graph with id 3, and (iii) connects it to node 2 with a normal edge. The mapping between ℓ_2 , a fresh and unused label, and the singleton set containing

node 3 is also introduced in $l^\#$, and $\{\ell_2\}$ is returned as the evaluation's result.

Dataframe assignment. When a portion of a dataframe is overwritten with new values, the semantics of variable assignment cannot be employed, as no variable changes value. Instead, the semantics of df_assign records the assignment as a node in the graph, connected to both the dataframe receiving it and the value being stored:

$$\mathbb{E}^\# \llbracket df_assign(df, cl, df') \rrbracket (\square, d^\#) \triangleq (\square_3, (v_3^\#, l_4^\#, g_4^\#), \{\ell_1, \dots, \ell_w\})$$

where:

- the assigned dataframe is evaluated: $\mathbb{E}^\# \llbracket df \rrbracket (\square, d^\#) = (\square_1, d_1^\#, \{\ell_1, \dots, \ell_w\})$;
- the names of the columns to be assigned are evaluated by $\mathcal{A}^\#$, producing their set of abstractions: $\mathbb{E}_{\mathcal{A}^\#} \llbracket cl \rrbracket (\square_1, d_1^\#) = (\square_2, d_2^\#, \{\langle \sigma_1^1, \dots, \sigma_m^1 \rangle, \dots, \langle \sigma_1^k, \dots, \sigma_m^k \rangle\})$;
- the dataframe used as right-hand in the assignment is then evaluated, resulting in a second set of labels: $\mathbb{E}^\# \llbracket df' \rrbracket (\square_2, d_2^\#) = (\square_3, (v_3^\#, l_3^\#, g_3^\#), \{\ell'_1, \dots, \ell'_p\})$;
- an `assign` node is created for each abstraction of the column names: $n_i = \text{assign}(\sigma_1^i, \dots, \sigma_m^i), \forall \langle \sigma_1^i, \dots, \sigma_m^i \rangle \in \{\langle \sigma_1^1, \dots, \sigma_m^1 \rangle, \dots, \langle \sigma_1^k, \dots, \sigma_m^k \rangle\}$;
- labels identifying `df` are remapped to the new nodes, as the assignment has side effects: $l_4^\# = l_3^\#[\ell_j \mapsto \{n_1, \dots, n_k\}, \forall \ell_j \in \{\ell_1, \dots, \ell_w\}]$;
- the graph is extended with the new nodes, connecting each of them with (i) each node pointed by a label of `df` with a normal edge, and (ii) each node pointed by labels of `df'` through an assign edge: $g_4^\# = g_3^\# \dot{\cup} (\{n_1, \dots, n_k\}, \{n' \rightarrow n_i \mid n' \in l_3^\#(\ell'_j), \forall \ell'_j \in \{\ell'_1, \dots, \ell'_p\}, \forall n_i \in \{n_1, \dots, n_k\}\} \cup \{n' \rightarrow n_i \mid n' \in l_3^\#(\ell_j), \forall \ell_j \in \{\ell_1, \dots, \ell_w\}, \forall n_i \in \{n_1, \dots, n_k\}\})$.

Example 5.3.7. When the df_assign at line 3 of Figure 5.3b is analyzed, `df1` is evaluated to $\{\ell_1\}$, the column names evaluate to the constant list $\langle \text{'birth'} \rangle$, and the $df_transform$ expression is resolved to $\{\ell_2\}$. The semantics proceeds by (i) creating a unique `assign('birth')` node, (ii) adding it to the graph with id 4, (iii) connecting it to node 3 (image of ℓ_2 in $l^\#$) with an assign edge, and finally (iv) connecting it to node 2 (image of ℓ_1 in $l^\#$) with a normal edge. To conclude, ℓ_1 is remapped to a set containing node 4 in $l^\#$, as the assignment has side effects.

Rows filtering. Similarly to value transformations, abstracting row filters can be problematic, as different facts about the filtering conditions can be of interest depending on the target analysis. We thus record the condition as-is within a node, delegating its interpretation to successive abstractions. Note that this is not an in-place operation: original dataframes are not modified, but a filtered view of them is

returned by the operation. We reflect this in our formalization by yielding a fresh label pointing to the filtering node. Formally:

$$\mathbb{E}^\# \llbracket df_filter(df, s, op, e) \rrbracket(\square, d^\#) \triangleq (\square_3, (v_3^\#, l_4^\#, g_4^\#), \{\bar{\ell}\})$$

where:

- the filtered dataframe is first evaluated: $\mathbb{E}^\# \llbracket df \rrbracket(\square, d^\#) = (\square_1, d_1^\#, \{\ell_1, \dots, \ell_w\})$;
- the name of the column targeted by the filtering is abstracted through $\mathcal{A}^\#$'s semantics, yielding a set of abstract strings: $\mathbb{E}_{\mathcal{A}^\#}^\# \llbracket s \rrbracket(\square_1, d_1^\#) = (\square_2, d_2^\#, \{\sigma_1, \dots, \sigma_k\})$;
- similarly, the value used in the comparison with the column's values is processed by $\mathcal{A}^\#$, computing a set of generic abstract values: $\mathbb{E}_{\mathcal{A}^\#}^\# \llbracket e \rrbracket(\square_2, d_2^\#) = (\square_3, (v_3^\#, l_3^\#, g_3^\#), \{v_1, \dots, v_m\})$;
- a filter node is then created for each possible column-value combination: $n_j^i = \text{filter}(\sigma_i, op, v_j), \forall \sigma_i \in \{\sigma_1, \dots, \sigma_k\}, \forall v_j \in \{v_1, \dots, v_m\}$;
- the nodes are used as images of a fresh new label: $l_4^\# = l_3^\# \cup \{(\bar{\ell}, \{n_1^1, \dots, n_m^k\})\}$;
- the graph is extended, adding all new nodes and connecting them with each node pointed by a label of df through a normal edge: $g_4^\# = g_3^\# \dot{\cup} (\{n_1^1, \dots, n_m^k\}, \{n' \rightarrow n_j^i \mid n' \in l_3^\#(\ell_j), \forall \ell_j \in \{\ell_1, \dots, \ell_w\}, \forall n_j^i \in \{n_1^1, \dots, n_m^k\}\})$.

Example 5.3.8. When the df_filter at line 7 of Figure 5.3b is analyzed, the semantics first evaluates its arguments: $df2$ is evaluated to $\{\ell_3\}$, the column name evaluates to 'age', and the value used for the comparison to the constant 50. A unique $\text{filter}('age', <, 50)$ node is then created, and it is added the graph with id 6. A normal edge connecting it to node 5 (image of ℓ_3 in $l^\#$) is also introduced, and the label l_4 is created and mapped to a set containing node 6 in $l^\#$.

Concatenation. The semantics of the concatenation must create a new dataframe with the contents of all of its arguments, that come compacted into a list. For of our domain, this means connecting all nodes of each argument to a concatenation node, that will be the image of a new label. The semantics is defined as follows:

$$\mathbb{E}^\# \llbracket df_concat(dl) \rrbracket(\square, d^\#) \triangleq (\square_1, (v_1^\#, l_2^\#, g_2^\#), \{\bar{\ell}\})$$

where:

- the list of dataframes to be concatenated evaluates to a set of abstractions (i.e., combinations of labels, each composed by one label for each dataframe involved): $\mathbb{E}_{\mathcal{A}^\#}^\# \llbracket dl \rrbracket(\square, d^\#) = (\square_1, (v_1^\#, l_1^\#, g_1^\#), \{\langle \ell_1^1, \dots, \ell_m^1 \rangle, \dots, \langle \ell_1^k, \dots, \ell_m^k \rangle\})$;
- a unique concatenation node is created: $n = \text{concat}$;

- the new node is used as the image of a fresh label: $l_2^\# = l_1^\# \cup \{(\bar{l}, \{n\})\}$;
- the graph is extended adding the new node, connecting it with each node of the source dataframes through concatenation nodes indexed with the argument position: $g_2^\# = g_1^\# \dot{\cup} \{\{n\}, \{n' \rightsquigarrow_j n \mid n' \in l_1^\#(\ell_j^\#), \forall i \in [1..k], \forall j \in [1..m]\}\}$.

Example 5.3.9. When the `df_concat` at line 8 of Figure 5.3b is evaluated, the semantics first evaluates the list of target dataframes, producing $\langle \ell_1, \ell_4 \rangle$ as unique possible abstraction for the elements of the list. The semantics then (i) creates the `concat` node, (ii) adds it to the graph with id 7, and (iii) connects it to nodes 4 and 6 (images of ℓ_1 and ℓ_4 in $l^\#$, respectively) with concatenation edges indexed with 1 and 2. A fresh label ℓ_5 is generated, and is mapped to a singleton set containing node 7 in $l^\#$.

5.4 A first application: inferring dataframes shape

Tracking dataframe transformations through $\mathcal{D}^\#$ is just the beginning. As $\mathcal{D}^\#$ provides a simplified and coherent view of the DS code manipulating the data, exploring the transformation and writing analyses over them becomes much simpler. In general, successive analyses targeting $\mathcal{D}^\#$ instead of starting from the PYTHON code can still be formalized and implemented as abstract interpretations. In fact, fixpoint algorithms can be applied over instances of $\mathcal{G}^\#$, treating the nodes as code. In this context, one has to define the abstract semantics w.r.t. each node's meaning, possibly taking into account the edges attached to them. In this section, we explore one of the possible objectives in the analysis of DS programs, as we aim at inferring the shape of each dataframe read from an external source.

We start by defining the domain $\mathcal{C}^\#$ of columns, that is still parametric to the same auxiliary domain $\mathcal{A}^\#$ used to build $\mathcal{D}^\#$. Denoting as $\bar{\Sigma}$ the set of possible string abstractions of $\mathcal{A}^\#$, $\mathcal{C}^\#$ is defined as the complete lattice $\langle \wp(\bar{\Sigma}) \rightarrow (\wp(\bar{\Sigma}) \times \wp(\bar{\Sigma})), \subseteq, \dot{\cup}, \dot{\cap}, \perp, \top \rangle$ obtained by functional lifting of the Cartesian product $\wp(\bar{\Sigma}) \times \wp(\bar{\Sigma})$. Elements of $\mathcal{C}^\#$ are functions whose domain is composed by sets abstract strings $\{\sigma_1, \dots, \sigma_p\}$ representing sources of data, and its co-domain is built over pairs of sets $(\{\hat{\sigma}_1, \dots, \hat{\sigma}_n\}, \{\hat{\sigma}'_1, \dots, \hat{\sigma}'_m\})$. Each $\hat{\sigma}_i$ is a column that is accessed before being assigned (and thus must be part of the original dataframe to prevent runtime errors), and each $\hat{\sigma}'_i$ is a column that is assigned during the execution (and thus might not exist in the original dataframe). The columns domain thus aims at inferring, for each external source of data, what columns must exist for the program to not crash. In our abstraction, we consider sets of sources as function keys since dataframes can be created through concatenation, and can thus have multiple sources. In this case, each accessed columns must exist in at least one source.

We define the semantics of $\mathcal{C}^\#$ w.r.t. nodes of $\mathcal{G}^\#$, as this kind of analysis does not need variable information stored in $\mathcal{L}^\#$ and $\mathcal{V}^\#$. The semantics is captured by

function:

$$\mathbb{S}^{C^\#} \llbracket n \rrbracket : \mathcal{G}^\# \times C^\# \rightarrow C^\#$$

that, given a node, a graph, and a pre-state, computes a post-state with updated column information. We require the graph as input of the semantics as, in general, reasoning about the neighborhood of the node might be required. In the following, we define $\mathbb{S}^{C^\#}$ for each of the possible nodes appearing in $\mathcal{G}^\#$. The definitions rely on the auxiliary function $\text{sources} : \mathcal{G}^\# \times \mathcal{N} \rightarrow \wp(\overline{\Sigma})$ that extracts all sources that influence the dataframe identified by the given node:

$$\text{sources}(g^\#, n) = \{ \sigma \mid \exists r \in \text{roots}(\text{bDFS}(g^\#, n)). r = \text{read}(\sigma) \}$$

The latter in turn exploits two helper functions whose definition is left implicit: $\text{bDFS} : \mathcal{G}^\# \times \mathcal{N} \rightarrow \mathcal{G}^\#$, that yields the portion of $g^\#$ extracted through a backward DFS starting from one of its nodes, and $\text{roots} : \mathcal{G}^\# \rightarrow \wp(\mathcal{N})$ that given a graph yields its roots (that is, its nodes with no predecessors).

Read. When reading data from an external resource, no particular column is accessed. Instead, we define an entry in our state corresponding to the possible abstractions of the resource identifier:

$$\mathbb{S}^{C^\#} \llbracket \text{read}(\sigma) \rrbracket (g^\#, c^\#) \triangleq c^\# \cup \{(\{\sigma\}, (\emptyset, \emptyset))\}$$

Concat. Similarly to `read`, `concat` does not access any column, but instead introduces a new entry in the post-state corresponding to the union of its sources:

$$\mathbb{S}^{C^\#} \llbracket \text{concat} \rrbracket (g^\#, c^\#) \triangleq c^\# \cup \{(\text{sources}(g^\#, \text{concat}), (\emptyset, \emptyset))\}$$

Transform. A transformation does not access any column explicitly, as it operates on the entirety of the dataframe that receives the transformation. As such, its semantics is defined as the identity function:

$$\mathbb{S}^{C^\#} \llbracket \text{transform}(f) \rrbracket (g^\#, c^\#) \triangleq c^\#$$

Access. The column access is the main vector for referencing columns by-name. This is reflected by its semantics, that adds every column name to the left-most set of the corresponding sources if we have no evidence of it being defined earlier:

$$\begin{aligned} \mathbb{S}^{C^\#} \llbracket \text{access}(\sigma_1, \dots, \sigma_m) \rrbracket (g^\#, c^\#) &\triangleq c^\# [k \mapsto (\hat{a} \cup \{ \sigma \in \{ \sigma_1, \dots, \sigma_m \} \mid \sigma \notin \hat{a} \}, \hat{a})], \\ k = \text{sources}(g^\#, \text{access}(\sigma_1, \dots, \sigma_m)), c^\#(k) &= (\hat{a}, \hat{a}) \end{aligned}$$

Note that names of columns that have already been assigned are ignored, as they are guaranteed to exist in the sources.

Filter. As row filtering is expressed as a condition over the value of a specific column, it indirectly represents a column access. This is once more reflected as the addition of the column name to the left-most set of the corresponding sources if it was not defined before:

$$\begin{aligned} \mathbb{S}^{C^\#} \llbracket \text{filter}(\sigma, op, v) \rrbracket (g^\#, c^\#) &\triangleq c^\#[k \mapsto (\hat{a} \cup (\{\sigma\} \setminus \hat{a}), \hat{a})], \\ k = \text{sources}(g^\#, \text{filter}(\sigma, op, v)), c^\#(k) &= (\hat{a}, \hat{a}) \end{aligned}$$

Assign. The dataframe assignment is the only node kind that can safely define non-existing columns. The semantics of this node adds the abstract column names to the right-most set of the corresponding sources:

$$\begin{aligned} \mathbb{S}^{C^\#} \llbracket \text{assign}(\sigma_1, \dots, \sigma_m) \rrbracket (g^\#, c^\#) &\triangleq c^\#[k \mapsto (\hat{a}, \hat{a} \cup \{\sigma_1, \dots, \sigma_m\})], \\ k = \text{sources}(g^\#, \text{assign}(\sigma_1, \dots, \sigma_m)), c^\#(k) &= (\hat{a}, \hat{a}) \end{aligned}$$

Example 5.4.1. We now use $C^\#$ to infer the shape of the dataframes abstracted by the graph in Figure 5.4. For the sake of clarity, we first analyze the left-most branch of the graph as a whole, followed by the right-most one, and conclude the analysis with the concat node. Note that, as the example uses constant propagation as auxiliary domain $\mathcal{A}^\#$, sources and column names appearing inside instances of $C^\#$ will be constant strings. The analysis begins by applying the semantics of read to node 1, using an empty $C^\#$ instance $c_0^\# = \{\}$ as pre-state, and producing the entry for the read resource:

$$\mathbb{S}^{C^\#} \llbracket \text{read}('italy.csv') \rrbracket (g^\#, c_0^\#) = c_1^\# = \{(\{'italy.csv'\}, (\emptyset, \emptyset))\}$$

This is in turn used as pre-state for the evaluation of node 2, where the semantics of access populates the function with the accessed column:

$$\mathbb{S}^{C^\#} \llbracket \text{access}('birth') \rrbracket (g^\#, c_1^\#) = c_2^\# = \{(\{'italy.csv'\}, (\{'birth'\}, \emptyset))\}$$

As the semantics of transform is the identity function, the pre-state of node 4 is the join of the post-state of both predecessors: $c_2^\# \dot{\cup} c_2^\# = c_2^\#$. This is then used as argument for the assign semantics, that produces the following post-state:

$$\mathbb{S}^{C^\#} \llbracket \text{assign}('birth') \rrbracket (g^\#, c_2^\#) = c_3^\# = \{(\{'italy.csv'\}, (\{'birth'\}, \{'birth'\}))\}$$

Equivalently, the analysis starts with the same empty pre-state $c_0^\#$ at node 5, applying the read semantics and producing the following post-state:

$$\mathbb{S}^{C^\#} \llbracket \text{read}('france.csv') \rrbracket (g^\#, c_0^\#) = c_4^\# = \{(\{'france.csv'\}, (\emptyset, \emptyset))\}$$

Then, $c_4^\#$ is used to compute the result of the filter semantics at node 6:

$$S^{C^\#}[\text{filter}('age', <, 50)](g^\#, c_4^\#) = c_5^\# = \{(\{'france.csv'\}, (\{'age'\}, \emptyset))\}$$

Lastly, at node 7, the pre-state is built as the lub of the predecessors' post-states:

$$c_6^\# = c_3^\# \dot{\cup} c_5^\# = \left\{ \begin{array}{l} (\{'italy.csv'\}, (\{'birth'\}, \{'birth'\})), \\ (\{'france.csv'\}, (\{'age'\}, \emptyset)) \end{array} \right\}$$

The semantics of concat can then be applied to $c_6^\#$, producing the final state of the analysis:

$$S^{C^\#}[\text{concat}](g^\#, c_6^\#) = c_7^\# = \left\{ \begin{array}{l} (\{'italy.csv'\}, (\{'birth'\}, \{'birth'\})), \\ (\{'france.csv'\}, (\{'age'\}, \emptyset)), \\ (\{'italy.csv'\}, \{'france.csv'\}, (\emptyset, \emptyset)) \end{array} \right\}$$

5.5 An early experiment using PyLISA

Both $\mathcal{D}^\#$ and $\mathcal{C}^\#$ have been implemented in PyLISA⁵, a PYTHON frontend for LISA. PyLISA analyzes JuPyter notebooks by extracting the PYTHON code from the cells that contain it. Cells are analyzed according to a specific user-defined execution sequence, defaulting to the order in which cells are defined in the notebook.

As our initial focus is on pandas, we provide the semantics of the library's functions through LISA's *native CFGs*, whose explicit semantics rewrites them into atomic transformations as suggested in Section 5.2.1. This means that the code is only instrumented at a semantic level, when the analysis resolves calls to pandas functions, and the original program is not syntactically modified. We also fix the auxiliary domain $\mathcal{A}^\#$ to a simple constant propagation tracking strings, integers, and floats, as well as constant lists of these types and constant dictionaries using them as keys. The choice was guided by experience, as most DS notebooks we found used explicit column names and row numbers whenever manipulating dataframes. $\mathcal{D}^\#$ has been implemented as a *Value Domain* (Section 3.3.3) that embeds a constant propagation domain, implemented as a *Non-Relational Domain*. At the end of the analysis, a simple *Check* visits the program and extracts the $\mathcal{G}^\#$ instance from the post-state of the last instruction. The check then executes a fixpoint over the graph using $\mathcal{C}^\#$. In the end, warnings are issued to inform the user about columns accessed and assigned for each data source.

As a first experiment, we selected the “*Coronavirus (COVID-19) Visualization & Prediction*”⁶ dataframe, one of the most popular notebooks aggregating data from

⁵<https://github.com/lisa-analyzer/pylisa>.

⁶<https://www.kaggle.com/code/therealcyberlord/coronavirus-covid-19->

different sources on Kaggle, a public repository of JuPyter notebooks for DS. The graph produced when analyzing such code is published on a GitHub Gist⁷ as it is too large for this manuscript. Note that the implemented analysis supports additional pandas constructs w.r.t. the ones presented in this chapter, that have been omitted as they do not contribute further to the intuition behind the domain. In the graph, these take the form of additional node kinds, whose intuitive meaning is explained in the Gist's introduction. The analysis generates the following warnings (where URL of csv files have been trimmed for compactness), correctly identifying all column names that appear in the notebook:

```
[File: daily_reports/08-23-2022.csv] Columns accessed before being assigned: 'Confirmed', 'Province_State', 'Country_Region', 'Incident_Rate', 'Deaths'
[File: daily_reports_us/08-23-2022.csv] Columns accessed before being assigned: 'Province_State', 'Testing_Rate', 'Total_Test_Results'
[File: time_series_covid19_confirmed_global.csv] Columns accessed before being assigned: 'Country/Region'
[File: time_series_covid19_deaths_global.csv] Columns accessed before being assigned: 'Country/Region'
```

5.6 Conclusion

This chapter presents an abstract interpretation approach to analyze PYTHON programs employed in data science and machine learning. Such programs manipulate dataframes, that is, complex in-memory tables collecting data that can be used to guide decision processes or train machine learning models. We designed an abstract domain that extracts the operations performed over dataframes, building a graph that encodes the order in which they are performed. Such a graph can be the subject of further analyses, inferring several properties such as the *shape* of the dataframes read by the program, or the absence of data leakages between training and testing phases of a machine learning process. As a guiding example of how to exploit our domain, we defined a simple abstract interpretation that computes, for each file read by the source program (and thus present inside the graph), the set of columns that are either accessed before being assigned or defined through an assignment. We provided an early implementation of both domains in PyLiSA, a LiSA frontend for PYTHON programs.

This work demonstrates the applicability of LiSA to dynamically typed and interpreted languages. Combined with the work presented in Chapter 4, we have given a preliminary demonstration of the wide spectrum of languages that LiSA's framework is able to analyze, providing the foundations for an in-depth study of multilanguage static analyses of real-world programs.

visualization-prediction, version 722, accessed on August 26th, 2022.

⁷<https://gist.github.com/lucaneg/9621f3296b7b47b12c5ee1c52066b3d1>.

Part III

String analysis

6 String analysis

Chapter Contents

6.1	The IMP language	126
6.2	The TARSIS abstract domain	126
6.2.1	Abstract domain and widening	126
6.2.2	String abstract semantics of IMP	129
6.3	Experimental Results	137
6.3.1	Precision of the domains on test cases	138
6.3.2	Evaluation on realistic code samples	139
6.3.3	Efficiency	141
6.4	Conclusion	142

In this chapter, we formalize TARSIS, a new abstract domain for string values based on finite state automata (FSA). Standard FSA has been shown to provide precise abstractions of string values when all the components of such strings are known, but with high computational cost. Instead of considering standard finite automata built over an alphabet of single characters, TARSIS considers automata that are built over an alphabet of strings. The alphabet comprises a special value to represent statically unknown strings. This avoids the creation of self-loops with any possible character as input, that would otherwise significantly degrade performance. We define the abstract semantics of mainstream string operations, namely `substring`, `length`, `indexOf`, `replace`, `concat`, and `contains`, either defined directly on the automaton or its equivalent regular expression.

As TARSIS was developed before LISA, it has been implemented into a prototypical static analyzer supporting a subset of JAVA. By comparing TARSIS with other cutting-edge domains for string analysis, results show that (i) when applied to simple code that causes a precision loss in simpler domains, TARSIS correctly approximates string values within a comparable execution time, (ii) on code that makes the standard automata domain unusable due to the complexity of the analysis, TARSIS is in position to perform in a limited amount of time, making it a viable domain for complex and real codebases, and (iii) TARSIS is able to precisely abstract complex string operations that have not been addressed by state-of-the-art domains. In LISA's infrastructure, TARSIS can be implemented as a *Non-Relational Domain* (Section 3.3.3), thus exploiting the factorization of the mapping from variables to its instances.

This chapter is based on the published paper [103], and it is structured as follows. Section 6.1 introduces a small IMP language that will be used to formalize abstract operations. Section 6.2 defines TARSIS, with all lattice operators and the semantics of six popular string manipulations. Finally, Section 6.3 reports experiments and comparisons with other domains.

```

a ∈ AE ::= x ∈ ID | n ∈ ℤ | a + a | a - a | a * a | a / a
          | length(s) | indexOf(s,s)
b ∈ BE ::= x ∈ ID | true | false | b && b | b || b | ! b
          | e < e | e == e | contains(s1,s2)
s ∈ SE ::= x ∈ ID | εσ ε | substr(s,a,a)
          | concat(s,s) | replace(s,s,s) (σ ∈ Σ*)
e ∈ E ::= a | b | s
st ∈ STMT ::= st ; st | skip | x = e | if (b) {st} else {st}
            | while (b) {st}
P ∈ IMP ::= st ;

```

Figure 6.1: IMP syntax

6.1 The IMP language

We introduce a minimal core language IMP, whose syntax is reported in Figure 6.1, that will be used for the formalization of the domain. Such language supports the main operators over strings. In particular, IMP supports arithmetic expressions (AE), Boolean expressions (BE), and string expressions (SE). Primitive values are $\text{VAL} = \mathbb{Z} \cup \Sigma^* \cup \{\text{true}, \text{false}\}$, namely integers, strings, and Booleans. Program states $\mathbb{M} : \text{ID} \rightarrow \text{VAL}$ map identifiers to primitive values, ranged over the meta-variable m . The concrete semantics of IMP statements is captured by the function $\mathbb{S}[\text{st}] : \mathbb{M} \rightarrow \mathbb{M}$ that is defined in a classical way and has thus been omitted. Such semantics relies on the one of expressions, that we capture as $\mathbb{E}[e] : \mathbb{M} \rightarrow \text{VAL}$. While the semantics concerning arithmetic and Boolean expressions is straightforward (and not of interest of this chapter), we define the part concerning strings in Figure 6.2.

6.2 The TARSIS abstract domain

In this section, we recast the original finite state automata abstract domain working over an alphabet of characters Σ , reported in Section 2.3.2, to an augmented abstract domain based on finite state automata over an alphabet of strings.

6.2.1 Abstract domain and widening

The key idea of TARSIS is to adopt the same abstract domain of [14], changing the alphabet on which finite state automata are defined to a set of strings, namely Σ^* . Clearly, the main concern here is that Σ^* is infinite and this would not permit us to adopt the finite state automata model, that requires the alphabet to be finite. Thus, in

$$\begin{aligned}
\mathbb{E}[\text{substr}(s, a, a')] \mathbb{m} &= \sigma_i \dots \sigma_j && \text{if } i \leq j < |\sigma|, i = \mathbb{E}[a] \mathbb{m}, j = \mathbb{E}[a'] \mathbb{m} \\
\mathbb{E}[\text{length}(s)] \mathbb{m} &= |\sigma| \\
\mathbb{E}[\text{indexOf}(s, s')] \mathbb{m} &= \begin{cases} \min\{i \mid \sigma_i \dots \sigma_j = \sigma'\} & \text{if } \exists i, j \in \mathbb{N}. \sigma_i \dots \sigma_j = \sigma' \\ -1 & \text{otherwise} \end{cases} \\
\mathbb{E}[\text{replace}(s, s', s'')] \mathbb{m} &= \begin{cases} \sigma[\sigma'/\sigma''] & \text{if } \sigma' \curvearrowright_s \sigma \\ \sigma & \text{otherwise} \end{cases} \\
\mathbb{E}[\text{concat}(s, s')] \mathbb{m} &= \sigma \cdot \sigma' \\
\mathbb{E}[\text{contains}(s, s')] \mathbb{m} &= \begin{cases} \text{true} & \text{if } \exists i, j \in \mathbb{N}. \sigma_i \dots \sigma_j = \sigma' \\ \text{false} & \text{otherwise} \end{cases} \\
&\text{where } \sigma = \mathbb{E}[s] \mathbb{m}, \sigma' = \mathbb{E}[s'] \mathbb{m}, \sigma'' = \mathbb{E}[s''] \mathbb{m}
\end{aligned}$$

Figure 6.2: Concrete semantics of IMP string expressions

order to solve this problem, we make this abstract domain *parametric* to the program we aim to analyze and in particular to its strings. Given an IMP program P , we denote by Σ_p^* any substring of strings appearing in P^1 , *delimiting* the space of string properties we aim to check only on P .

At this point, we can instantiate the automata-based framework proposed in [14] with the new alphabet as

$$\langle \mathcal{TFA}_{/\equiv}, \sqsubseteq_{\mathcal{T}}, \sqcup_{\mathcal{T}}, \sqcap_{\mathcal{T}}, \text{Min}(\emptyset), \text{Min}(\mathcal{A}_p^*) \rangle$$

The alphabet on which finite state automata are defined is $\mathcal{A}_p \triangleq \Sigma_p^* \cup \{\top\}$, where \top is a special symbol that we intend as "any possible string". Let \mathcal{TFA} be the set of all deterministic finite state automata over the alphabet \mathcal{A}_p . Since we can have more automata recognizing a given language, $\mathcal{TFA}_{/\equiv}$ is the quotient set of \mathcal{TFA} w.r.t. the equivalence relation induced by language equality, that is, the elements of the domain are equivalence classes. For simplicity, when we write $A \in \mathcal{TFA}_{/\equiv}$, we intend the equivalence class of A . $\sqsubseteq_{\mathcal{T}}$ is the partial order induced by language inclusion, $\sqcup_{\mathcal{T}}$ and $\sqcap_{\mathcal{T}}$ are the lub and the glb over elements of $\mathcal{TFA}_{/\equiv}$, computing the equivalence class of the union and the intersection of the two automata representing the corresponding classes, respectively. The bottom element is $\text{Min}(\emptyset)$, corresponding to the automaton recognizing the empty language, and the maximum is $\text{Min}(\mathcal{A}_p^*)$, namely the automaton recognizing any string over \mathcal{A}_p .

As the change in the alphabet of automata does not modify the lattice structure (recall that all lattice operators are defined w.r.t. language equivalence), properties proved in in [14] still hold on $\mathcal{TFA}_{/\equiv}$. Briefly, $\langle \mathcal{FA}_{/\equiv}, \sqsubseteq_{\mathcal{FA}}, \sqcup_{\mathcal{FA}}, \sqcap_{\mathcal{FA}}, \text{Min}(\emptyset), \text{Min}(\Sigma^*) \rangle$ is a

¹The set Σ_p^* can be easily computed collecting the constant strings in P by visiting its abstract syntax tree and then computing their substrings.

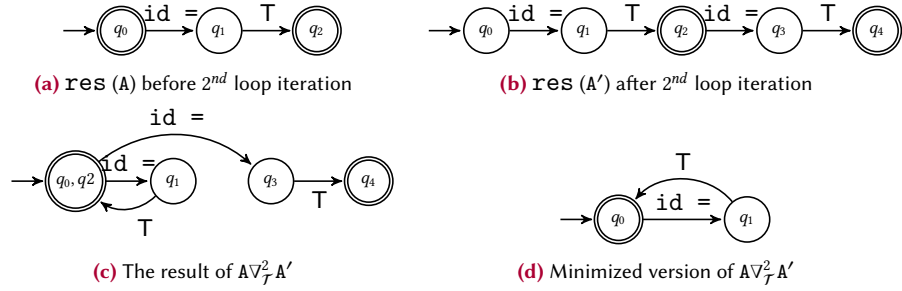


Figure 6.3: Example of widening application

lattice, but it is not complete. This is due to, in general, the union of an infinite set of regular languages being potentially context-free. Consider, for instance, the family of regular languages $\mathcal{L}_i = \{a^i b^i\}, i \in \mathbb{N}$. Being regular implies that an automaton $A_i \in \text{Fa}_{/\equiv}$ exist for each of them. As the union of these languages $\mathcal{L}' = \bigcup_i \mathcal{L}_i = \{a^i b^i \mid i \in \mathbb{N}\}$ is context-free, an automaton $A \in \text{Fa}_{/\equiv}$ such that $\mathcal{L}(A) = \mathcal{L}'$ cannot exist. Such result also holds for $\mathcal{T}\text{Fa}_{/\equiv}$, resulting in the absence of a Galois Connection with the concrete string domain $\wp(\Sigma^*)$. Nevertheless, this is not a concern since weaker forms of abstract interpretation are still possible [47] still guaranteeing soundness relations between concrete and abstract elements (e.g., polyhedra [50]). In particular, we can still ensure soundness by comparing the concretizations of our abstract elements (cf. Section 8 of [47]). Hence, we define the concretization function $\gamma_{\mathcal{T}} : \mathcal{T}\text{Fa}_{/\equiv} \rightarrow \wp(\Sigma^*)$ as $\gamma_{\mathcal{T}}(A) \triangleq \bigcup_{\sigma \in \mathcal{L}(A)} \text{Flat}(\sigma)$, where Flat converts a string over A_{p} into a set of strings over Σ^* . For instance, $\text{Flat}(a \text{ T T } b b \ c) = \{a \sigma b b c \mid \sigma \in \Sigma^*\}$. Note that, the language of strings (over the alphabet Σ) recognized by A corresponds to the concretization function reported above, namely $\mathcal{L}(A) = \gamma_{\mathcal{T}}(A)$.

Widening. Similarly to the standard automata domain $\text{Fa}_{/\equiv}$, also $\mathcal{T}\text{Fa}_{/\equiv}$ does not satisfy ACC, meaning that fixpoint computations over $\mathcal{T}\text{Fa}_{/\equiv}$ may not converge in a finite time. Hence, we need to equip $\mathcal{T}\text{Fa}_{/\equiv}$ with a widening operator to ensure the convergence of the analysis. We define the widening operator $\nabla_{\mathcal{T}}^n : \mathcal{T}\text{Fa}_{/\equiv} \times \mathcal{T}\text{Fa}_{/\equiv} \rightarrow \mathcal{T}\text{Fa}_{/\equiv}$, parametric to $n \in \mathbb{N}$, taking two automata as input and returning an over-approximation of the least upper bounds between them, as required by widening definition. We rely on the standard automata widening reported in Section 2.3.2, that, informally speaking, can be seen as a *subset construction* algorithm [52] up to languages of strings of length n .

Example 6.2.1. To better explain the widening $\nabla_{\mathcal{T}}^n$, consider the following function manipulating strings².

```
function f(v) {
```

²For the sake of readability, in the program examples presented in this chapter the + operation between strings corresponds to the string concatenation.

```

2  res = "";
3  while (?)
4  res = res + "id = " + v;
5  return res;
6  }

```

Function f takes as input parameter v and returns variable res . Let us suppose that v is a statically unknown string, corresponding to the automaton recognizing T (i.e., $\text{Min}(\{T\})$). The result of function f is a string of the form $id = T$, repeated zero or more times. Since the `while` guard is unknown, the number of iterations is statically unknown, and in turn, also the number of concatenations performed inside the loop body. The goal here is to over-approximate the value returned by function f , i.e., the value of `res` at the end of the function.

Let A , reported in Figure 6.3a, be the automaton abstracting the value of `res` before starting the second iteration of the loop, and let A' , reported in Figure 6.3b be the automaton abstracting the value of `res` at the end of the second iteration. At this point, we want to apply the widening operator $\nabla_{\mathcal{T}}^n$ to A and A' , proceeding as follows. We first compute $A \sqcup_{\mathcal{T}} A'$ (corresponding to the automaton reported in Figure 6.3b except that q_0 is also a final state). On this automaton, we merge all states that recognize the same A_P -strings of length n , with $n \in \mathbb{N}$. In our example, let n be 2. The resulting automaton is reported in Figure 6.3c, where q_0 and q_2 are put together, and the other states are left as singletons since they cannot be merged with other ones. Figure 6.3d depicts the minimized version of Figure 6.3c.

The parametric widening $\nabla_{\mathcal{T}}^n$ has been proved to meet the widening requirements (i.e., over-approximation of the least upper bound and convergence on infinite ascending chains) in [57]. The parameter n , tuning the widening precision, is arbitrary and can be chosen by the user. As highlighted in [14], higher values of n result in greater precision in over-approximating lub of infinite ascending chains (i.e., in fixpoint computations).

A classical improvement on widening-based fixpoint computations is to integrate a threshold [40], namely widening is applied to over-approximate lub when a certain threshold (usually over some property of abstract values) is overcome. In fixpoint computations, we decide to apply the previously defined widening $\nabla_{\mathcal{T}}^n$ only when the number of the states of the lubbed automata becomes greater than some threshold $\tau \in \mathbb{N}$. This permits us to postpone the widening application, getting more precise abstractions when the automata sizes are below the threshold. At the moment, the threshold τ is not automatically inferred and is left as future work.

6.2.2 String abstract semantics of IMP

In this section, we define the abstract semantics of the string operators defined in Figure 6.2 over the new string domain $\mathcal{TFA}_{/\equiv}$. Since IMP supports strings, integers, and Booleans values, we need a way to merge the corresponding abstract domains. In

particular, we abstract integers with the well-known `Interval` abstract domain [43] defined as $\text{Intv} \triangleq \{ [a, b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{+\infty\}, a \sqsubseteq b \} \cup \{\perp_{\text{Intv}}\}$ and Booleans with $\text{Bool} \triangleq \wp(\{\text{true}, \text{false}\})$. As usual, we denote by \sqcup_{Intv} and \sqcup_{Bool} the lubs between intervals and Booleans, respectively. We merge such abstract domains in $\text{VAL}^\#$ by the smashed sum abstract domain [12] as $\text{VAL}^\# \triangleq \mathcal{TFA}_{/\equiv} \oplus \text{Intv} \oplus \text{Bool}$. Informally, the smashed sum abstract domain introduces a top element, and it *smashes* (i.e., joins) the bottom elements of the involved domains. The abstract program state is represented through abstract program memories $\mathbb{M}^\# : \text{ID} \rightarrow \text{VAL}^\#$ from identifiers to abstract values. The abstract semantics is captured by the function $\mathbb{S}^\# \llbracket \text{st} \rrbracket : \mathbb{M}^\# \rightarrow \mathbb{M}^\#$, relying on the abstract semantics of expressions defined as $\mathbb{E}^\# \llbracket e \rrbracket : \mathbb{M}^\# \rightarrow \text{VAL}^\#$. We focus on the abstract semantics of string operations, as the semantics of the remaining ones are standard and do not involve strings. We provide soundness proofs for our semantics in Appendix A.

To define the abstract semantics of `IMP` over `TARSIS`, it is worth highlighting that one can think to reuse the transformers adopted in the standard finite state automata abstract domain [14]: unfortunately, this is not possible since those deal with automata defined on an alphabet of single characters, and do not handle the character `T` used in `TARSIS` that must be treated as a special symbol.

Concat. Given $A, A' \in \mathcal{TFA}_{/\equiv}$, the abstract semantics of `concat` returns a new automaton recognizing the language $\{ \sigma \cdot \sigma' \mid \sigma \in \mathcal{L}(A), \sigma' \in \mathcal{L}(A') \}$, that is, the concatenation between the strings of $\mathcal{L}(A)$ with the strings of $\mathcal{L}(A')$. This is easily achievable relying on the standard automata concatenation [52]. Let $s, s' \in \text{SE}$ and suppose that $\mathbb{E}^\# \llbracket s \rrbracket \mathbb{m}^\# = \langle Q, A, \delta, q_0, F \rangle \in \mathcal{TFA}_{/\equiv}$, $\mathbb{E}^\# \llbracket s' \rrbracket \mathbb{m}^\# = \langle Q', A', \delta', q'_0, F' \rangle \in \mathcal{TFA}_{/\equiv}$. Then, the abstract semantics of `concat` is defined as:

$$\mathbb{E}^\# \llbracket \text{concat}(s, s') \rrbracket \mathbb{m}^\# \triangleq \text{Min}(\langle Q \cup Q', A, \delta \cup \delta' \cup \{ (q_f, \epsilon, q'_0) \mid q_f \in F \}, q_0, F' \rangle)$$

Following the standard automata concatenation, the semantics of `concat` between A with A' merges the automata introducing an ϵ -transition from each final state of A to the initial state of A' . The result's initial state is the initial state of A , while its final states are the ones of A' . Soundness of `concat` is proven in Appendix A.1.

Length. Given $A \in \mathcal{TFA}_{/\equiv}$, the abstract semantics of `length` returns an interval $[c_1, c_2]$ such that $\forall \sigma \in \mathcal{L}(A). c_1 \leq |\sigma| \leq c_2$. Let $s \in \text{SE}$, supposing that $\mathbb{E}^\# \llbracket s \rrbracket \mathbb{m}^\# = A \in \mathcal{TFA}_{/\equiv}$. The `length` abstract semantics is:

$$\mathbb{E}^\# \llbracket \text{length}(s) \rrbracket \mathbb{m}^\# \triangleq \begin{cases} [|\text{minPath}(A)|, +\infty] & \text{if } \text{cyclic}(A) \vee \text{readsTop}(A) \\ [|\text{minPath}(A)|, |\text{maxPath}(A)|] & \text{otherwise} \end{cases}$$

where $\text{readsTop}(A) \Leftrightarrow \exists q, q' \in Q. (q, T, q') \in \delta$. Note that, when evaluating the length of the minimum path, `T` is considered to have a length of 0. Soundness of `length` is



Figure 6.4: Example automata demonstrating length's semantics

proven in Appendix A.2.

Example 6.2.2. Consider the automaton A reported in Figure 6.4a. The minimum path of A is $(q_0, aa, q_1), (q_1, T, q_2), (q_2, bb, q_4)$ and its length is 4. Since a transition labeled with T is in A (and its length cannot be statically determined), the abstract length of A is $[4, +\infty]$. Consider now the automaton A' reported in Figure 6.4b. In this case, A' has no cycles and has no transitions labeled with T and the length of all strings recognized by A' can be determined. The length of the minimum path of A' is 3 (lower path of A'), the length of the maximum path of A' is 7 (upper path of A') and consequently the abstract length of A' is $[3, 7]$.

Contains. Given $A, A' \in \mathcal{TF}_{A/\equiv}$, the abstract semantics of `contains` should return `true` if all strings of A' are surely contained into all strings of A, `false` if no string of A' is contained in any string of A, and $\{\text{true}, \text{false}\}$ in the other cases. Our semantics exploits the definition of *single-path automaton* [15], that is, an automaton such that all strings in its language are prefixes of the longest string in the language. If A' is a single-path automaton, the containment of the longest string of A' on each path of A is enough to yield `true`. Note that a single-path automaton cannot read the symbol T. We rely on the predicate `singlePath(A)` to test if A is a non-cyclic single-path automaton, and we denote by σ_{sp} its longest string. Let $s, s' \in sE$, supposing that $E^\# \llbracket s \rrbracket m^\# = A \in \mathcal{TF}_{A/\equiv}$, $E^\# \llbracket s' \rrbracket m^\# = A' \in \mathcal{TF}_{A'/\equiv}$. The `contains` abstract semantics is:

$$E^\# \llbracket \text{contains}(s, s') \rrbracket m^\# \triangleq \begin{cases} \text{false} & \text{if } A' \sqcap_{\mathcal{T}} \text{FA}(A) = \text{Min}(\emptyset) \\ \text{true} & \text{if } \text{singlePath}(A') \\ & \wedge \forall \pi \in \text{paths}(A^{ac}). \sigma_{sp} \curvearrow_s \sigma_\pi \\ \{\text{true}, \text{false}\} & \text{otherwise} \end{cases}$$

In the first case, we denote by $\text{FA}(A)$ the factor automaton of A, i.e., the automaton recognizing all substrings of A. If A does not share any of its substrings with A', the abstract semantics safely returns `false` (checking the emptiness of the greatest lower bound between $\text{FA}(A)$ and A'). Then, if A' is a single path automaton, the abstract semantics returns `true` if all paths of A^{ac} read the longest string of A', with A^{ac} being a copy of A where all the cycles have been removed. Here, we abuse notation denoting with $\sigma_{sp} \curvearrow_s \sigma_\pi$ the fact that σ_{sp} is a substring of each string in $\text{Flat}(\sigma_\pi)$. Otherwise,

`{true, false}` is returned. Soundness of `contains` is proven in Appendix A.3.

IndexOf. Given $A, A' \in \mathcal{TFA}_{/\equiv}$, the `indexOf` abstract semantics returns the interval of the first indexes of the strings of $\mathcal{L}(A')$ inside strings of $\mathcal{L}(A)$, recalling that when there exists a string of $\mathcal{L}(A')$ that is not a substring of at least one string of $\mathcal{L}(A)$, the resulting interval must take into account -1 as well. Let $s, s' \in \Sigma$ and suppose $\mathbb{E}^\#[\![s]\!]m^\# = A$ and $\mathbb{E}^\#[\![s']]\!]m^\# = A'$. The abstract semantics of `indexOf` is defined as:

$$\mathbb{E}^\#[\![\text{indexOf}(s, s')]\!]m^\# \triangleq \begin{cases} [-1, +\infty] & \text{if } \text{cyclic}(A) \vee \text{cyclic}(A') \vee \text{readsTop}(A') \\ [-1, -1] & \text{if } \forall \sigma' \in \mathcal{L}(A') \nexists \sigma \in \mathcal{L}(A) . \sigma' \curvearrow_s \sigma \\ \bigsqcup_{\sigma \in \mathcal{L}(A')} \text{IO}(A, \sigma) & \text{otherwise} \end{cases}$$

If one of the automata has cycles or the automaton abstracting strings we aim to search for (A') has a T-transition, we return $[-1, +\infty]$. Moreover, if none of the strings recognized by A' is contained in a string recognized by A , we can safely return the precise interval $[-1, -1]$ since any string recognized by A' is never a substring of a string recognized by A ³. If none of the aforementioned conditions are met, we rely on the auxiliary function $\text{IO} : \mathcal{TFA}_{/\equiv} \times \Sigma^* \rightarrow \text{Intv}$, that, given an automaton A and a string $\sigma \in \Sigma^*$, returns an interval corresponding to the possible first positions of σ in strings recognized by A . Since A' surely recognizes a finite language (i.e., has no cycles or top transitions), the idea is to apply $\text{IO}(A, \sigma)$ to each $\sigma \in \mathcal{L}(A')$ and to return the lub of the resulting intervals. In particular, the function $\text{IO}(A, \sigma)$ returns an interval $[i, j] \in \text{Intv}$ where i and j are computed as follows:

$$i = \begin{cases} -1 & \text{if } \exists \pi \in \text{paths}(A) . \sigma \not\curvearrow_s \sigma_\pi \\ \min_{\pi \in \text{paths}(A)} \left\{ i \mid \begin{array}{l} \sigma_f \in \text{Flat}(\sigma_\pi) \\ \wedge \sigma_{f_i} \dots \sigma_{f_{i+n}} = \sigma \end{array} \right\} & \text{otherwise} \end{cases}$$

$$j = \begin{cases} -1 & \text{if } \forall \pi \in \text{paths}(A) . \sigma \not\curvearrow_s \sigma_\pi \\ +\infty & \text{if } \exists \pi \in \text{paths}(A) . \sigma \curvearrow_s \sigma_\pi \\ & \wedge \pi \text{ reads T before } \sigma \\ \max_{\pi \in \text{paths}(A)} \left\{ i \mid \begin{array}{l} \sigma_f \in \text{Flat}(\sigma_\pi) \\ \wedge \sigma_{f_i} \dots \sigma_{f_{i+n}} = \sigma \\ \wedge \sigma \not\curvearrow_s \sigma_{f_0} \dots \sigma_{f_{i+n-1}} \end{array} \right\} & \text{otherwise} \end{cases}$$

As for the abstract semantics of `contains`, we abuse notation denoting with $\sigma \curvearrow_s \sigma_\pi$ the fact that σ is a substring of each string in $\text{Flat}(\sigma_\pi)$. Given $\text{IO}(A, \sigma) = [i, j] \in \text{Intv}$, i corresponds to the minimal position where the first occurrence of σ can be found in

³Note that this is a decidable check since A and A' are cycle-free, otherwise the interval $[-1, +\infty]$ would be returned in the first case.



Figure 6.5: Example of may-replacement

A , while j to the maximal one. Let us first focus on the computation of the minimal position. If there exists a path π of A s.t. σ is not recognized by σ_π , then the minimal position where σ can be found in A does not exist and -1 is returned. Otherwise, the minimal position where σ begins across all possible π is returned. Let us consider now the computation of the maximal position. If all paths of the automaton do not recognize σ , then -1 is returned. If there exists a path where σ is recognized but the character T appears earlier in the path, then $+\infty$ is returned. Otherwise, the maximal index of the first occurrences of σ across the paths of A is returned. Soundness of `indexOf` is proven in Appendix A.4.

Replace. To give the intuition about how the abstract semantics of `replace` will work, consider three automata $A, A_s, A_r \in \mathcal{TFA}_{/\equiv}$. Roughly speaking, the abstract semantics of `replace` substitutes strings of A_s with strings of A_r inside strings of A . Let us refer to A_s as the *search automaton* and to A_r as the *replace automaton*. We need to specify two types of possible replacements, by means of the following example.

Example 6.2.3. Consider $A \in \mathcal{TFA}_{/\equiv}$ that is depicted in Figure 6.5a and suppose that the search automaton A_s is the one recognizing the string bbb and the replace automaton A_r is a random automaton. In this case, the `replace` abstract semantics performs a *must-replace* over A , namely substituting the sub-automaton composed by q_1 and q_2 with the replace automaton A_r . Instead, let us suppose that the search automaton A_s is the one recognizing bbb or cc . Since it is unknown which string *must* be replaced (between bbb and cc), the `replace` abstract semantics needs to perform a *may-replace*: when a string recognized by the search automaton is met inside a path of A , it is left unaltered in the automaton and, in the same position where the string is met, the abstract `replace` only extends A with the replace automaton. An example of may replacement is reported in Figure 6.5, where A is the one reported in Figure 6.5a, the search automaton A_s is the one recognizing the language $\{bbb, cc\}$ and the replace automaton A_r recognizes the string rr .

Before introducing the abstract semantics of `replace`, we define how to replace a string into an automaton's path. In particular, we define algorithm `RP` in Algorithm 4, that given a path π of an arbitrary automaton, a replace automaton A^r , and $\sigma^s \in \Sigma^* \cup \{T\}$ returns a new automaton built starting from the path, but where portions of the path that recognize σ^s have been replaced with A^r .

Algorithm 4: RP algorithm

Data: $\pi = (q_0, \sigma_0, q_1), \dots, (q_{n-1}, \sigma_{n-1}, q_n), A^r = \langle Q^r, A, \delta^r, q_0^r, F^r \rangle \in \mathcal{TF}_{A/\equiv}, \sigma^s \in \Sigma^* \cup \{T\}$

Result: $A \in \mathcal{TF}_{A/\equiv}$

- 1 $Q^{result} \leftarrow \{q \mid (q, \sigma, q') \in \pi \vee (q', \sigma, q) \in \pi\};$
- 2 $\delta^{result} \leftarrow \pi;$
- 3 **foreach** $(q_i, \sigma_i^s, q_{i+1}), \dots, (q_{i+n-1}, \sigma_{i+n-1}^s, q_{i+n}) \in \pi$ **do**
- 4 $\langle Q', A, \delta', q_0', F' \rangle \leftarrow \text{clone}(A^r);$
- 5 $\delta^{result} \leftarrow \delta^{result} \cup (q_i, \epsilon, q_0');$
- 6 $\delta^{result} \leftarrow \delta^{result} \cup \{(q_f, \epsilon, q_{i+n}) \mid q_f \in F'\};$
- 7 $Q^{result} \leftarrow Q^{result} \setminus \{q_{i+1}, \dots, q_{i+n-1}\};$
- 8 $\delta^{result} \leftarrow \delta^{result} \setminus \{(q_{i+1}, \sigma_1^s, q_{i+2}), \dots, (q_{i+n-2}, \sigma_{n-1}^s, q_{i+n-1})\};$
- 9 **return** $\langle Q^{result}, A, \delta^{result}, q_0^o, F^o \rangle;$

Algorithm 4 searches the given string σ^s across path π , collecting the sequences of transitions that recognize the search string σ^s and extracting them from π (line 3). Whenever a matching sequence is found, A^r is cloned to A' to ensure that all additions target a different set of nodes (line 4). Then, an ϵ -transition is introduced going from the first state of the sequence to the initial state of A' , and one such transition is also introduced for each final state of A' , connecting that state with the ending state of the sequence (lines 5-6). The list of states composing the sequence of transitions is then removed from the result (line 7), together with the transitions connecting them (line 8), since those were needed only to recognize the string that has been replaced. Note that RP corresponds to a must-replace. At this point, we are ready to define the replace abstract semantics. In particular, if either A or A_s have cycles or if one of them has a T-transition, we return $\text{Min}(\{T\})$, namely the automaton recognizing T. Otherwise, the replace abstract semantics is:

$$E^\#[\text{replace}(s, s_s, s_r)]m^\# \triangleq \begin{cases} A & \text{if } \forall \sigma_s \in \mathcal{L}(A_s) \\ & \exists \sigma \in \mathcal{L}(A). \\ & \sigma_s \curvearrowright_s \sigma \\ \bigsqcup_{\pi \in \text{paths}(A)} \text{RP}(\pi, \sigma_s, A_r) & \text{if } \mathcal{L}(A_s) = \{\sigma_s\} \\ \bigsqcup_{\substack{\sigma \in \mathcal{L}(A_s) \\ \pi \in \text{paths}(A)}} \text{RP}(\pi, \sigma, A_r \sqcup_{\mathcal{T}} \text{Min}(\{\sigma\})) & \text{otherwise} \end{cases}$$

In the first case, if none of the strings recognized by the search automaton A_s is contained in strings recognized by A , we can safely return the original automaton A without any replacement. In the special case where $\mathcal{L}(A_s) = \{\sigma_s\}$, we return the automaton obtained by replacing σ_s across all paths of A using function $\text{RP}(\pi, \sigma_s, A_r)$. In the last case, for each string $\sigma \in \mathcal{L}(A_s)$ and for each path $\pi \in \text{paths}(A)$, we perform a may replace of σ with A_r : note that, this exactly corresponds to a call RP where the replace automaton is $A_r \sqcup_{\mathcal{T}} \text{Min}(\{\sigma\})$. The so far obtained automata are finally lubbed together. Soundness of `replace` is proven in Appendix A.5.

Algorithm 5: Sb algorithm

```

Data:  $r$  regex over  $A$ ,  $i, j \in \mathbb{N}$ 
Result:  $\{ (\sigma, n_1, n_2) \mid \sigma \in \Sigma^*, n_1, n_2 \in \mathbb{N} \}$ 
1 if  $j = 0 \vee r = \emptyset$  then
2   return  $\emptyset$ ;
3 else if  $r = \sigma \in \Sigma^*$  then
4   if  $i > |\sigma|$  then return  $\{(\epsilon, i - |\sigma|, j)\}$ ;
5   else if  $i + j > |\sigma|$  then return  $\{(\sigma_i \dots \sigma_{|\sigma|-1}, 0, j - |\sigma| + i)\}$ ;
6   else return  $\{(\sigma_i \dots \sigma_{i+j}, 0, 0)\}$ ;
7 else if  $r = \top$  then
8   result  $\leftarrow \{(\epsilon, i - k, j) \mid 0 \leq k \leq i, k \in \mathbb{N}\}$ ;
9   result  $\leftarrow \text{result} \cup \{(\epsilon^k, 0, j - k) \mid 0 \leq k \leq j, k \in \mathbb{N}\}$ ;
10  return result;
11 else if  $r = r_1 r_2$  then
12  result  $\leftarrow \emptyset$ ;
13  subs1  $\leftarrow \text{Sb}(r_1, i, j)$ ;
14  foreach  $(\sigma_1, i_1, j_1) \in \text{subs}_1$  do
15    if  $j_1 = 0$  then
16      result  $\leftarrow \text{result} \cup \{(\sigma_1, i_1, j_1)\}$ ;
17    else
18      result  $\leftarrow \text{result} \cup \{(\sigma_1 \cdot \sigma_2, i_2, j_2) \mid (\sigma_2, i_2, j_2) \in \text{Sb}(r_2, i_1, j_1)\}$ ;
19  return result;
20 else if  $r = r_1 \parallel r_2$  then
21  return  $\text{Sb}(r_1, i, j) \cup \text{Sb}(r_2, i, j)$ ;
22 else if  $r = (r_1)^*$  then
23  result  $\leftarrow \{(\epsilon, i, j)\}$ ; partial  $\leftarrow \emptyset$ ;
24  repeat
25    result  $\leftarrow \text{result} \cup \text{partial}$ ; partial  $\leftarrow \emptyset$ ;
26    foreach  $(\sigma_n, i_n, j_n) \in \text{result}$  do
27      foreach  $(\text{suff}, i_s, j_s) \in \text{Sb}(r_1, i_n, i_n + j_n)$  do
28        if  $\exists (\sigma', k, w) \in \text{result} . \sigma' = \sigma_n \cdot \text{suff} \wedge k = i_s \wedge w = j_s$  then
29          partial  $\leftarrow \text{partial} \cup \{(\sigma_n \cdot \text{suff}, i_s, j_s)\}$ ;
30  until partial  $\neq \emptyset$ ;
31  return result;

```

Substring. Given $A \in \mathcal{TFA}_{/\equiv}$ and two intervals $i, j \in \text{Intv}$, the abstract semantics of `substring` returns a new automaton A' soundly approximating any substring from i to j of strings recognized by A , for any $i \in i, j \in j$ s.t. $i \sqsubseteq j$.

Given $A \in \mathcal{TFA}_{/\equiv}$, in the definition of the `substring` semantics, we rely on the corresponding regex r since the two representations are equivalent and regexes allow us to define a more intuitive formalization of such semantics. Let us suppose that $\mathbb{E}^\# \llbracket s \rrbracket m^\# = A \in \mathcal{TFA}_{/\equiv}$ and let us denote by r the regex corresponding to the language recognized by A . At the moment, let us consider exact intervals representing one integer value, namely $\mathbb{E}^\# \llbracket a_1 \rrbracket m^\# = [i, i]$ and $\mathbb{E}^\# \llbracket a_2 \rrbracket m^\# = [j, j]$, with $i, j \in \mathbb{N}$. In this case, the abstract semantics is defined as:

$$\mathbb{E}^\# \llbracket \text{substr}(s, a_1, a_2) \rrbracket m^\# \triangleq \text{Min}(\{ \sigma \mid (\sigma, 0, 0) \in \text{Sb}(r, i, j - i) \})$$

where `Sb` takes as input a regex r , two indexes $i, j \in \mathbb{N}$, and computes the set of substrings from i to j of all the strings recognized by r . In particular, `Sb` is defined by

Algorithm 5 and, given a regex r and $i, j \in \mathbb{N}$, it returns a set of triples of the form (σ, n_1, n_2) , such that σ is the *partial substring* that Algorithm 5 has computed up to now, $n_1 \in \mathbb{N}$ tracks how many characters have still to be skipped before the substring can be computed and $n_2 \in \mathbb{N}$ is the number of characters Algorithm 5 needs still to look for to successfully compute a substring. Hence, given $Sb(r, i, j)$, the result is a set of such triples; note that given an element of the resulting set (σ, n_1, n_2) , $n_2 = 0$ means that no more characters are needed and σ corresponds to a proper substring of r from i to j . Thus, from the resulting set, we can filter out the partial substrings, and retrieve only proper substrings of r from i to j by only considering the value of n_2 . Algorithm 5 is defined by structural induction on the input regex r :

1. $j = 0$ or $r = \emptyset$ (lines 1-2): \emptyset is returned since we either completed the substring or we have no more characters to add;
2. $r = \sigma \in \Sigma^*$ (lines 3-6): if $i > |\sigma|$, the requested substring happens after this atom, and we return a singleton set $\{\epsilon, i - |\sigma|, j\}$, thus tracking the consumed characters before the start of the requested substring; if $i + j > |\sigma|$, the substring begins in σ but ends in subsequent regexes, and we return a singleton set containing the substring of σ from i to its end, with $n_1 = 0$ since we began collecting characters, and $n_2 = j - |\sigma| + i$ since we collected $|\sigma| - i$ characters; otherwise, the substring is fully inside σ , and we return the substring of σ from i to $i + j$, setting both n_1 and n_2 to 0;
3. $r = T$ (lines 7-10): since r might have any length, we generate substrings that (a) gradually consume all the missing characters before the substring can begin (line 8) and (b) gradually consume all the characters that make up the substring, adding the unknown character \bullet (line 9);
4. $r = r_1 r_2$ (lines 11-20): the desired substring can either be fully found in r_1 or r_2 , or could overlap them; thus, we compute all the partial substrings of r_1 , recursively calling Sb (line 13); for all $\{\sigma_1, i_1, j_1\}$ returned, substrings that are fully contained in r_1 (i.e., when $j_1 = 0$) are added to the result (line 16) while the remaining ones are joined with ones computed by recursively calling Sb on r_2 with $n_1 = j_1$ and $n_2 = j_2$;
5. $r = r_1 | r_2$ (lines 20-21): we return the partial substring of r_1 and the ones of r_2 , recursively calling Sb on both of them;
6. $r = (r_1)^*$ (lines 22-31): we construct the set of substrings through fixpoint iteration, starting by generating $\{\epsilon, i, j\}$ (corresponding to r_1 repeated 0 times - line 23) and then, at each iteration, by joining all the partial results obtained until now with the ones generated by a further recursive call to Sb , keeping only the joined results that are new (lines 24-30).

Above, we have defined the abstract semantics of `substr` when intervals are constant. When $\mathbb{E}^\# \llbracket a_1 \rrbracket \mathfrak{m}^\# = [i, j]$ and $\mathbb{E}^\# \llbracket a_2 \rrbracket \mathfrak{m}^\# = [l, k]$, with $i, j, l, k \in \mathbb{N}$, the abstract semantics of `substr` is

$$\mathbb{E}^\# \llbracket \text{substr}(s, a_1, a_2) \rrbracket \mathfrak{m}^\# \triangleq \bigsqcup_{a \in [i, j], b \in [l, k], a \sqsubseteq b} \text{Min}(\{ \sigma \mid (\sigma, 0, 0) \in \text{Sb}(r, a, b - a) \})$$

We do not precisely handle the cases when the intervals are unbounded (e.g., $[1, +\infty]$). These have been already considered in [14] with ad-hoc treatment, and one may recast the same proposed idea in our context. Nevertheless, when these cases are met, our analysis returns the automaton recognizing any possible substring of the input automaton, still guaranteeing soundness (proven in Appendix A.6).

6.3 Experimental Results

TARSIS has been compared with five other domains, namely the prefix (Pr), suffix (Su), char inclusion (Ci), bricks (Br) domains (all defined in [41]), and $\text{Fa}_{/\equiv}$ (defined in [14], adapting their abstract semantics to JAVA, without altering their precision).

All domains have been implemented in a prototype static analyzer for a subset of the JAVA language, similar to IMP (Section 6.1), with the addition of the `assert` statement. In particular, our analyzer raises a *definite* alarm (DA for short) when a failing assert (i.e., whose condition is definitely false) is met, while it raises a *possible* alarm (PA for short) when the assertion *might* fail (i.e., the assertion's condition evaluates to T_{Bool}). Comparisons have been performed by analyzing the code through the smashed sum domain specified in Section 6.2.2 with trace partitioning [116] (note that all traces are merged when evaluating an assertion), plugging in the various string domains. All experiments have been performed on an HP EliteBook G6 machine, with an Intel Core i7-8565U @ 1.8GHz processor and 16 GB of RAM memory.

To achieve fair comparison with the other string domains, the subjects of our evaluation are small hand-crafted code fragments that represent standard string manipulations occurring regularly in software. Pr, Su, Ci, and Br have been built to model simple properties and to work with integers instead of intervals, and have been evaluated on small programs: Section 6.3.1 compares them to TARSIS and $\text{Fa}_{/\equiv}$ without expanding the scope of such evaluations. Section 6.3.2 instead focuses on slightly more advanced and complex string manipulations that are not modeled by the aforementioned domains, but that $\text{Fa}_{/\equiv}$ and TARSIS can indeed tackle, highlighting differences between them.

It is important to notice that performances of programs relying on automata (highlighted in Section 6.3.3) are heavily dependent on their implementation. Both $\text{Fa}_{/\equiv}$ and TARSIS (whose sources are available on GitHub^{4,5}) come as non-optimized

⁴ $\text{Fa}_{/\equiv}$ source code: <https://github.com/SPY-Lab/fsa>.

⁵TARSIS source code: <https://github.com/UniVE-SSV/tarsis>.

```

1 void substring() {
2   String res = "substring test";
3   if (nondet)
4     res = res + " passed";
5   else
6     res = res + " failed";
7   result = res.substring(5, 18);
8   assert (res.contains("g"));
9   assert (res.contains("p"));
10  assert (res.contains("f"));
11  assert (res.contains("d"));
12 }

```

```

1 void loop() {
2   String value = read();
3   String res = "Repeat: ";
4   while (nondet)
5     res = res + value + "!";
6   assert (res.contains("t"));
7   assert (res.contains("!"));
8   assert (res.contains("f"));
9 }

```

(a) Program subs

(b) Program Loop

Figure 6.6: Program samples used for domain comparison

	Program SUBS		Program LOOP	
Pr	ring test	✗	Repeat:	✗
Su	ε	✗	ε	✗
Ci	[] [abdefgilnprstu]	✗	[:aepRt] [!:aepRtT]	✓
Br	[{ring test fai,ring test pas}] (1,1)	✓	[{T}] (0, +∞)	✗
Fa/ _≡	ring test(pas fai)	✓	Repeat:(T)*	✓
TFA/ _≡	(ring test pas ring test fai)	✓	Repeat:(T!)*	✓

Table 6.1: Values of res at the first assert of each program

proof-of-concept libraries (specifically, TARSIS has been built following the structure of Fa/_≡ to ensure a fair performance comparison) whose performances can be greatly improved.

6.3.1 Precision of the domains on test cases

We start by considering programs SUBS (Figure 6.6a) and LOOP (Figure 6.6b). SUBS calls `substring` on the concatenation between two strings, where the first one is constant and the second one is chosen in a non-deterministic way (i.e., `nondet` condition is statically unknown, lines 3-6). LOOP builds a string by repeatedly appending a suffix, which contains a user input (i.e., an unknown string), to a constant value. Table 6.1 reports the value approximation for `res` for each abstract domain and analyzed program when the first assertion of each program is met, as well as if the abstract domain precisely dealt with the program assertions. For the sake of readability, TARSIS and Fa/_≡ approximations are both expressed as regexes.

When analyzing SUBS, both Pr and Su lose precision since the string to append to `res` is statically unknown. This leads, at line 7, to a partial substring of the concrete one with Pr, and to an empty string with Su. Instead, the `substring` semantics of Ci moves every character of the receiver in the set of possibly contained ones, thus the abstract value at line 7 is composed of an empty set of included characters, and a set of possibly included characters containing the ones of both strings. Finally, Br, Fa/_≡, and TARSIS are expressive enough to track any string produced by any concrete

execution of `SUBS`.

When evaluating the assertions of `SUBS`, PAs should be raised on lines 9 and 10, since "p" or "f" might be in `res`, together with a DA alarm on line 11, since "d" is surely not contained in `res`. No alarm should be raised on line 8 instead, since "g" is part of the common prefix of both branches and thus will be included in the substring. Such behavior is achieved when using `Br`, `Fa/=`, or `TARSIS`. Since the `substring` semantics of `Ci` moves all characters to the set of possibly contained ones, PAs are raised on all four assertions. As `Su` loses all information about `res`, PAs are raised on lines 7-10 when using such domain. `Pr` instead tracks the definite prefix of `res`, thus the PA at line 7 is avoided.

When analyzing `LOOP`, we expect to obtain no alarm at line 6 (since character "t" is always contained in the resulting string value), and PAs at lines 7 and 8. `Pr` infers as prefix of `res` the string "Repeat: ", keeping such value for the whole analysis of the program. This allows the analyzer to prove the assertion at line 6, but it raises PAs when it checks the ones at lines 7 and 8. Again, `Su` loses all information about `res` since the `lub` operation occurring at line 3 cannot find a common suffix between "Repeat: " and "!", hence PAs are raised on lines 6-8. Since the set of possible characters contains T, `Ci` can correctly state that any character might appear in the string. For this reason, two PAs are reported on lines 7 and 8, while no alarm is raised on line 6 (again, this is possible since the string used in the `contains` call has length 1). The alternation of T and "!" prevents `Br` normalization algorithm from merging similar bricks. This will eventually lead to overcoming the length threshold k_L , hence resulting in the $\{\{T\}\}(0, +\infty)$ abstract value. In such a situation, `Br` returns T_{Bool} on all `contains` calls, resulting in PAs on lines 6-8. The parametric widening of `Fa/=` collapses the colon into T. In `TARSIS`, since the automaton representing `res` grows by two states each iteration, the parametric widening defined in Section 6.2.1 can collapse the whole content of the loop into a 2-states loop recognizing T!. The precise approximation of `res` of both domains enables the analyzer to detect that the assertion at line 6 always holds, while PAs are raised on lines 7 and 8.

In summary, `Pr` and `Su` failed to produce the expected results on both `SUBS` and `LOOP`, while `Ci` and `Br` produced exact results in one case (`LOOP` and `SUBS`, respectively), but not in the other. Hence, `Fa/=` and `TARSIS` were the two only domains that produced the desired behavior in these rather simple test cases.

6.3.2 Evaluation on realistic code samples

In this section, we explore two real-world code samples. Method `TOSTRING` (Figure 6.7a) transforms an array of names that come as string values into a single string. While it resembles the code of `LOOP` in Figure 6.6b (thus, results of all the analyses show the same strengths and weaknesses), now assertions check `contains` predicates with a multi-character string. Method `COUNT` (Figure 6.7b) makes use of `COUNTMATCHES` (reported in Section 1.4) to prove properties about its return value.

```

1 void toString(String[] names) {
2   String res = "People: {";
3   int i = 0;
4   while (i < names.length){
5     res = res + names[i];
6     if (i != names.length - 1)
7       res = res + ",";
8     i = i + 1;
9   }
10  res = res + "}";
11  assert
12    (res.contains("People"));
13  assert (res.contains(","));
14  assert (res.contains("not"));
15 }

```

```

1 void count(boolean nondet) {
2   String str;
3   if (nondet) str = "this is the
4     thing";
5   else str = "the throat";
6   int count = countMatches(str,
7     "th");
8   assert (count > 0);
9   assert (count == 0);
10  assert (count == 3);
11 }

```

(a) Program TOSTRING

(b) Program COUNT

Figure 6.7: Programs used for assessing domain precision

	Program TOSTRING	Program COUNT
Pr	People:{	\times $[0, +\infty]$ \times
Su	ϵ	\times $[0, +\infty]$ \times
Ci	$\{\{\}:Peopl\} \{\{\}: ,PeoplT\}$	\times $[0, +\infty]$ \times
Br	$\{\{T\}\} (0, +\infty)$	\times $[0, +\infty]$ \times
Fa _{/=}	People:{(T)*T}	\checkmark $[2, 3]$ \checkmark
TFA _{/=}	People:{ }People:{(T,)*T}	\checkmark $[2, 3]$ \checkmark

Table 6.2: Values of res and count at the first assert of the respective program

Since the analyzer is not interprocedural, we inlined COUNTMATCHES inside COUNT. Table 6.2 reports the results of both methods (stored in res and count, respectively) evaluated by each analysis at the first assertion, as well as if the abstract domain precisely dealt with the program assertions.

As expected, when analyzing TOSTRING, each domain showed results similar to those of LOOP. In particular, we expect to obtain no alarm at line 11 (since "People" is surely contained in the resulting string), and two PAs at lines 12 and 13. Pr, Su, Ci, and Br raise PAs on all three assert statements. Fa_{/=} and TARSIS detect that the assertion at line 11 always holds. Thus, when using them, the analyzer raises PAs on lines 12 and 13 since the comma character is part of res if the loop is iterated at least once, and T might match "not".

If COUNT (with the inlined code from COUNTMATCHES) was to be executed, count would be either 2 or 3 when the first assertion is reached, depending on the choice of str. Thus, no alarm should be raised at line 6, while a DA should be raised on line 7, and a PA on line 8. Since Pr, Su, Ci, and Br do not define most of the operations used in the code, the analyzer does not have information about the string on which COUNTMATCHES is executed, and it thus abstracts count with the interval $[0, +\infty]$. Thus, PAs are raised on lines 6-8. Instead, Fa_{/=} and TARSIS are able to detect that sub is present in all the possible strings represented by str. Thus, thanks to trace partitioning, the trace where the loop is skipped and count remains 0 gets

Domain	SUBS	LOOP	TOSTRING	COUNT
Pr	11 ms	3 ms	78 ms	29 ms
Su	10 ms	2 ms	92 ms	29 ms
Ci	10 ms	3 ms	90 ms	29 ms
Br	13 ms	3 ms	190 ms	28 ms
Fa _{/≡}	10 ms	52013 ms	226769 ms	4235 ms
TARSIS	34 ms	38 ms	299 ms	39 ms

Table 6.3: Execution times of the domains on each program

discarded. Then, when the first `indexOf` call happens, $[0, 0]$ is stored into `idx`, since all possible values of `str` start with `sub`. Since the call to `length` yields $[10, 17]$, all possible substrings from $[2, 2]$ (`idx` plus the length of `sub`) to $[10, 17]$ are computed (namely, "e throat", "is is th", "is is the", ..., "is is the thing"), and the resulting automaton is the one that recognizes all of them. Since the value of `sub` is still contained in every path of such automaton, the loop guard still holds and the second iteration is analyzed, repeating the same operations. When the loop guard is reached for the third time, the remaining substring of the shorter starting string (namely "roat") recognized by the automaton representing `str` will no longer contain `sub`: a trace where `count` equals $[2, 2]$ will leave the loop. A further iteration is then analyzed, after which `sub` is no longer contained in any of the strings that `str` might hold. Thus, a second and final trace where `count` equals $[3, 3]$ will reach the assertions, and will be merged by interval `lub`, obtaining $[2, 3]$ as final value for `count`. This allows TARSIS and Fa_{/≡} to identify that the assertion at line 7 never holds, raising a DA, while the one at line 8 might not hold, raising a PA.

6.3.3 Efficiency

The detailed analysis of two test cases and two examples taken from real-world code underlined that TARSIS and Fa_{/≡} are the only domains able to obtain precise results on them. We now discuss the efficiency of the analyses. Table 6.3 reports the execution times for all the domains on the case studies analyzed in this section. Overall, Pr, Su, Ci, and Br are the fastest domains with execution times usually below 100 msec. Thus, if on the one hand, these domains failed to prove some of the properties of interest, they are quite efficient and they might be helpful to prove simple properties. TARSIS execution times are higher but still comparable with them (about 50% overhead on average). Instead, Fa_{/≡} blows up on three out of the four test cases (and in particular on TOSTRING). Hence, TARSIS is the only domain that executes the analysis in a limited time while being able to prove all the properties of interest in these four case studies.

The reason behind the performance gap between TARSIS and Fa_{/≡} can be accounted on the alphabets underlying the automata. In Fa_{/≡}, automata are built over

an alphabet of single characters. While this simplifies the semantic operations, it also causes state and transition blow up w.r.t. the size of the string that needs to be represented. This does not happen in TARSIS, since atomic strings (not built through concatenation or other string manipulations) are part of the alphabet and can be used as transition symbols. Having fewer states and transitions to operate upon drastically lowers the time and memory requirements of automata operations, making TARSIS faster than $Fa_{/\equiv}$.

TARSIS's alphabet has another peculiarity w.r.t. $Fa_{/\equiv}$'s: it has a special symbol for representing an unknown string. Having such a symbol requires some fine-tuning of the algorithms to have them behave differently when the symbol is encountered, but without additional tolls on their performances. $Fa_{/\equiv}$'s alphabet does not have such a symbol, thus representing the unknown string is achieved through a state having one self-loop for each character in the alphabet (including ϵ). This requires significantly more resources for automata algorithms, leading to higher execution times.

6.4 Conclusion

In this chapter, we introduced TARSIS, an abstract domain for sound abstraction of string values. TARSIS models strings as regular languages, thus using finite state automata to represent them. The novelty behind TARSIS is twofold. At first, the equivalent regular expression form is used in place of automata to better model strings semantics. Moreover, the alphabet backing the automata is defined over strings instead of single characters: this leads to a direct reduction in the number of states and transitions necessary to model string values, a result that is emphasized by our preliminary yet meaningful experimental evaluation.

With such a powerful string abstraction, one can empower multilanguage analysis to precisely model runtime configurations that exploit strings, a pattern widely used in multilanguage systems such as IoT networks.

Part IV

Conclusion

7 Conclusion

In this chapter, we draw the conclusions of our work, summarizing the results we achieved and their limitations, and suggesting the future work that could arise from them.

7.1 Thesis summary

In this thesis, we presented LISA, a framework for modular multilanguage analysis with a focus on modularity and ease of use, with an open-source implementation available under MIT license. LISA operates on a program composed of extensible control flow graphs (CFGs), that are built by language-specific *frontends* to uniformly represent syntactic structures. Each CFG node defines its own language-specific semantics in terms of atomic semantic constructs, called *symbolic expressions*, that abstract domain can interpret. The state of the analysis is modularly built to let each component be agnostic and independent, enabling effortless reconfiguration of the analysis to tune precision and performances.

LISA provides support for modeling library functions through *native CFGs* and SARL. A *native CFG* is a special CFG whose code can replace its call sites, as it only contains one node that expresses the semantics of the whole function. SARL is a domain-specific language that allows one to specify the execution model of libraries and frameworks to instruct a static analyzer about their execution model. This approach enables the support of new frameworks through a readable and documentable model without modifying the analysis engine, as it is applied to a program to produce annotations that agnostically instruct the analyzer about the runtime environment.

We instantiated LISA to demonstrate its effectiveness in multilanguage contexts by analyzing an IoT network whose codebase is composed of JAVA and C++, successfully discovering a vulnerability that spanned across the two languages. Moreover, we used GoLISA, a static analyzer for Go applications based on LISA, to detect critical non-deterministic behaviors in smart contracts relying on a flow-based approach. Our analyses are able to discern usages of non-determinism that might endanger the consensus from ones that are innocuous and can be allowed. We also used PyLISA, a static analyzer for PYTHON programs based on LISA, to perform early experiments on an abstraction that tracks operations performed on data science *dataframes*, building a graph where properties about them can be easily computed.

Finally, we defined TARSIS, an abstract domain for sound abstraction of string values, aiming at providing powerful approximations of strings that can be used to set up multilanguage communication. TARSIS is based on finite state automata paired with their equivalent regular expression: a representation that allows precise modeling of complex string values. Experiments show that TARSIS achieves great precision also on code that heavily manipulates string values, while the time needed for the

analysis is comparable with the one of other simpler domains.

7.2 Future directions

Contributions presented in this work still have relevant margins for improvement and extension, as our long-term goal is to provide easy-to-use techniques with wide applicability spectrum, even outside of academia. In the following, we emphasize the first lines of work that will move us closer to our ultimate goal, identifying three different axes: applicability and ease of use, modularization of new features, and analyses evolution.

Applicability and ease of use

As a multilanguage analyzer, one of our objectives is to target as many programming languages as possible. This not only means having *frontends* for each of them, but also ensuring that the internal LiSA program model is flexible and parametric enough to represent syntactic structures, semantics, execution model, inheritance, and all of their other peculiarities. Currently, other than GoLiSA and PyLiSA, *frontends* for JAVA, JAVA BYTECODE, RUST, and MICHELSON BYTECODE are in development, but this is undoubtedly a long-term effort.

One additional vision for LiSA's future is to not only provide users with an easy-to-use tool where new analysis can be implemented quickly and tested on several languages, but where they can also easily compare different implementations with their own. As such, we plan on extending LiSA to ship with several well-known component implementations, from numerical and string domains (e.g., Octagons [98] and Bricks [41]), to property domains (e.g., Information Flow [118, 55]), to interfaces with widely accepted static analysis libraries (e.g., Apron [82] and PPL [18]).

Modularization of new features

In its current state, LiSA is missing some functionalities that hinder development of some analyses. We identify here three separate axes aimed at widening the spectrum of components that can take part in LiSA's analyses.

The abstract state presented in [61] entails decoupling of memory and values. However, it was explicitly designed for object-oriented languages, focusing on field accesses only. This means that, whenever an access to a *nested* location happens (e.g., accessing a field of an object or an array element), the expression identifying it (e.g., the field's name or the array element's index) must be statically known. Hence, evaluating dynamic accesses like the dereference of an arithmetic expression between pointers is not possible within this framework. We intend to extend the latter to enable these.

A second line of work targets modular communication between abstract domains. Notice that communication here is not a synonym for refinement, as two domains refine each other (e.g., through reduced or Granger product [38]) when they track the same kind of information (e.g., numerical values). Instead, we want to focus on finding a modular alternative to smashed sums [12]. Smashed sums are flexible combinations of domains tracking different properties, but their downside resides in the absence of modularity. For instance, when evaluating a substring expression, the domain tracking string values has to extract information about the numerical bounds of the operation from the respective domain. Logical predicates [94, 76] can be derived by abstract domain elements to over-approximate their values, but have been shown to lose precision and their extension to non-numerical values is not trivial.

Finally, LISA currently supports forward analyses only. While providing the infrastructure for backward analyses is trivial (as it would resemble what has been presented in this thesis), we intend to formalize a wider modular framework where backward and forward analyses can cooperate together.

Analyses evolution

Here we lay out our next steps to improve on the analyses and abstract domains presented in this thesis.

In our presentation of TARSIS, we did not investigate completeness properties w.r.t. the considered operations of interest. This would ensure that no loss of information is related to \mathcal{TFA}_{\equiv} due to the input abstraction process [15]. Moreover, TARSIS's precision could be greatly improved by tracking relations between T elements appearing inside (possibly separate) automata, and we intend to work in this direction to design an extension of the domain.

There are plenty of future directions that our work on JuPyter notebooks can take. As this work is still ongoing, the obvious first line of work is to prove the soundness of the proposed semantics abstractions, followed by an investigation of their completeness to further improve the domain. Besides, inferring the shape of dataframes is not the only useful analysis one can employ for DS software. In fact, after strengthening shape inference to incorporate rows and cell properties, we aim at providing abstractions to detect data leakages and biases. Lastly, we aim at extending the abstraction to more `pandas` functions, and to further libraries other than `pandas` itself.

In the context of smart contracts, we will first extend GoLISA to detect more blockchain-related vulnerabilities, such as numerical overflow, to provide an efficient analyzer to use in *on-chain* architectures [105], incorporating the verification of smart contracts in the consensus protocol. We also aim at expanding our benchmark to cover Tendermint Core and Cosmos SDK, tweaking the analysis to also be efficient on such frameworks.

Appendices

A Soundness proofs of TARSI's semantics

We prove the soundness of TARSI's abstract semantics by showing that its concretization is an over-approximation of the concrete one. As we formalized our transfer functions w.r.t. the smashed sum $\text{VAL}^\# \triangleq \mathcal{TFA}_{/=} \oplus \text{Intv} \oplus \text{Bool}$, we compare concretizations of its elements with a concrete smashed sum $\overline{\text{VAL}} \triangleq \wp(\Sigma^*) \cup \wp(\mathbb{Z}) \cup \wp(\{\text{true}, \text{false}\})$, that is defined as a collecting semantics. We abuse notation denoting with $M : \text{ID} \rightarrow \overline{\text{VAL}}$ the set of collecting memories, ranging over meta-variable m , that associate each identifier to a collecting value. The concrete expression semantics of such domain is defined as $\mathbb{E}[\![e]\!] : M \rightarrow \overline{\text{VAL}}$, evaluating e and returning its possible values. Such semantics is defined as the additive lift of the one in Figure 6.2. Function $\gamma_{\text{VAL}^\#} : \text{VAL}^\# \rightarrow \overline{\text{VAL}}$ is the smashed sum concretization and it is defined as:

$$\gamma_{\text{VAL}^\#}(a) \triangleq \begin{cases} \emptyset & \text{if } a = \perp \\ \gamma_{\text{Intv}}(a) & \text{if } a \in \text{Intv} \\ \gamma_{\text{Bool}}(a) & \text{if } a \in \text{Bool} \\ \gamma_{\mathcal{T}}(a) & \text{if } a \in \mathcal{TFA}_{/=} \\ \overline{\text{VAL}} & \text{otherwise} \end{cases}$$

where $\gamma_{\text{Intv}} : \text{Intv} \rightarrow \wp(\mathbb{Z})$ and $\gamma_{\text{Bool}} : \text{Bool} \rightarrow \wp(\{\text{true}, \text{false}\})$ correspond to the concretization functions of intervals and Booleans, respectively. We can now define the abstract memories concretization $\gamma : M^\# \rightarrow M$ as $\gamma(m^\#) \triangleq \{ (x, \gamma_{\text{VAL}^\#}(m^\#(x))) \mid x \in \text{dom}(m^\#) \}$. With this setup, we prove the abstract semantics to be sound by proving:

$$\forall m^\# \in M^\#. \mathbb{E}[\![e]\!]\gamma(m^\#) \subseteq \gamma_{\text{VAL}^\#}(\mathbb{E}^\#[\![e]\!](m^\#))$$

In the following, we remove the subscript from γ to avoid cluttering the notation, since it is clear from the context which concretization function applies. Moreover, we mark proof steps as *automata lift* if they represent the transition from a condition over languages (that is, sets of strings) to its equivalent condition over automata.

A.1 Soundness of Concat

Theorem 2. $\mathbb{E}^\#[\![\text{concat}(s, s')]\!]$ is a sound abstraction of $\mathbb{E}[\![\text{concat}(s, s')]\!]$. Formally:

$$\forall m^\# \in M^\#, \forall s, s' \in \text{SE}. \mathbb{E}[\![\text{concat}(s, s')]\!]\gamma(m^\#) \subseteq \gamma(\mathbb{E}^\#[\![\text{concat}(s, s')]\!](m^\#))$$

Proof. Soundness follows from the fact that finite state automata and regular languages are closed under finite concatenation [53]. \square

A.2 Soundness of Length

Theorem 3. $E^\#[\text{length}(s)]$ is a sound abstraction of $E[\text{length}(s)]$. Formally:

$$\forall m^\# \in M^\#, \forall s \in SE. E[\text{length}(s)]\gamma(m^\#) \subseteq \gamma(E^\#[\text{length}(s)]m^\#)$$

Proof. The collecting semantics of `length` is defined as the additive lift of the concrete one reported in Figure 6.2, namely $E[\text{length}(s)]m = \{ |\sigma| \mid \sigma \in \mathcal{L} \}$, where $E[s]m = \mathcal{L} \in \wp(\Sigma^*)$. Let us suppose that $E^\#[s]m^\# = A \in \mathcal{TF}_{A/\equiv}$ and $\gamma(A) = \mathcal{L} \in \wp(\Sigma^*)$, and let $L = \{ |\sigma| \mid \sigma \in \mathcal{L} \}$. Following the semantics definition, if $\text{cyclic}(A) \vee \text{readsTop}(A)$, we prove the soundness as:

$$\begin{aligned} & E[\text{length}(s)]\gamma(m^\#) \\ &= L && \text{\{ def. } E \} \\ &\subseteq \gamma([\min L, +\infty]) && \text{\{ def. min, } \gamma \} \\ &= \gamma([\min\text{Path}(A), +\infty]) && \text{\{ def. minPath \} \\ &= \gamma(E^\#[\text{length}(s)]m^\#) && \text{\{ def. } E^\#, 1^{st} \text{ case \}} \end{aligned}$$

Otherwise, as \mathcal{L} is a finite language, the soundness is proven as:

$$\begin{aligned} & E[\text{length}(s)]\gamma(m^\#) \\ &= L && \text{\{ def. } E \} \\ &\subseteq \gamma([\min L, \max L]) && \text{\{ def. max, } \gamma \} \\ &= \gamma([\min\text{Path}(A), \max\text{Path}(A)]) && \text{\{ def. minPath, maxPath \} \\ &= \gamma(E^\#[\text{length}(s)]m^\#) && \text{\{ def. } E^\#, 2^{nd} \text{ case \}} \end{aligned}$$

Soundness is thus proven since the interval computed by the abstract semantics is always an over-approximation of the concrete set of lengths. \square

A.3 Soundness of Contains

Theorem 4. $E^\#[\text{contains}(s, s')]$ is a sound abstraction of $E[\text{contains}(s, s')]$. Formally:

$$\forall m^\# \in M^\#, \forall s, s' \in SE. E[\text{contains}(s, s')]\gamma(m^\#) \subseteq \gamma(E^\#[\text{contains}(s, s')]m^\#)$$

Proof. The collecting semantics of `contains` is defined as the additive lift of the

concrete one, that is $\mathbb{E}[\text{contains}(s, s')]\mathbb{m} = \{b \mid b = \text{contains}(\sigma, \sigma'), \sigma \in \mathcal{L}, \sigma' \in \mathcal{L}'\}$, where $\mathbb{E}[s]\mathbb{m} = \mathcal{L} \in \wp(\Sigma^*)$, $\mathbb{E}[s']\mathbb{m} = \mathcal{L}' \in \wp(\Sigma^*)$ and $\text{contains} : \Sigma^* \times \Sigma^* \rightarrow \{\text{true}, \text{false}\}$ corresponds to the concrete semantics of Figure 6.2. Let us suppose that $\mathbb{E}^\#[s]\mathbb{m}^\# = A \cdot \gamma(A) = \mathcal{L}$ and $\mathbb{E}^\#[s']\mathbb{m}^\# = A' \cdot \gamma(A') = \mathcal{L}'$, where $A, A' \in \mathcal{TF}_{A/\equiv}$ and $\mathcal{L}, \mathcal{L}' \in \wp(\Sigma^*)$. We split the proof following possible values produced by the concrete semantics.

If $\mathbb{E}[\text{contains}(s, s')]\gamma(\mathbb{m}^\#) = \{\text{false}\}$, no substring of the strings in \mathcal{L} is in \mathcal{L}' :

$$\begin{aligned}
& \mathbb{E}[\text{contains}(s, s')]\gamma(\mathbb{m}^\#) = \{\text{false}\} \\
\iff & \forall \sigma \in \mathcal{L} \forall \sigma' \in \mathcal{L}' . \sigma' \not\curvearrowright_s \sigma && \text{\{def. E\}} \\
\iff & \mathcal{L}(\text{FA}(A)) \cap \mathcal{L}' = \emptyset && \text{\{def. FA\}} \\
\iff & \text{FA}(A) \sqcap_{\mathcal{T}} A' = \text{Min}(\emptyset) && \text{\{automata lift\}} \\
\iff & \mathbb{E}^\#[\text{contains}(s, s')]\mathbb{m}^\# = \text{false} && \text{\{def. E}^\#, 1^{\text{st}} \text{ case}\}}
\end{aligned}$$

Hence, under our starting hypothesis, $\gamma(\mathbb{E}^\#[\text{contains}(s, s')]\mathbb{m}^\#) = \{\text{false}\}$, proving soundness.

Instead, when $\mathbb{E}[\text{contains}(s, s')]\gamma(\mathbb{m}^\#) = \{\text{true}\}$, all strings in \mathcal{L} contain all the strings of \mathcal{L}' . This invalidates the first case of our abstract semantics, as $\exists \sigma \in \mathcal{L}(\text{FA}(A)) . \sigma \in \mathcal{L}'$. If $\text{singlePath}(A')$ holds, our semantics matches the concrete one:

$$\begin{aligned}
& \mathbb{E}[\text{contains}(s, s')]\gamma(\mathbb{m}^\#) = \{\text{true}\} \wedge \text{singlePath}(A') \\
\iff & \forall \sigma \in \mathcal{L} . \sigma_{\text{sp}} \curvearrowright_s \sigma && \text{\{def. E, singlePath\}} \\
\iff & \forall \sigma \in \mathcal{L}(A^{ac}) . \sigma_{\text{sp}} \curvearrowright_s \sigma && \text{\{\mathcal{L}(A^{ac}) \subseteq \mathcal{L}\}} \\
\iff & \forall \pi \in \text{paths}(A^{ac}) . \sigma_{\text{sp}} \curvearrowright_s \sigma_\pi && \text{\{automata lift\}} \\
\iff & \mathbb{E}^\#[\text{contains}(s, s')]\mathbb{m}^\# = \text{true} && \text{\{def. E}^\#, 2^{\text{nd}} \text{ case}\}}
\end{aligned}$$

Instead, when A' is not a single-path automaton, the semantics returns $\{\text{true}, \text{false}\}$, and soundness is trivially met.

Finally, when $\mathbb{E}[\text{contains}(s, s')]\mathbb{m}^\# = \{\text{true}, \text{false}\}$, soundness is trivially satisfied as none of the conditions appearing in the definition of $\mathbb{E}^\#[\text{contains}(s, s')]$ are satisfied, and the latter returns $\{\text{true}, \text{false}\}$ (3^{rd} case) as well. \square

A.4 Soundness of IndexOf

Theorem 5. $\mathbb{E}^\#[\text{indexOf}(s, s')]$ is a sound abstraction of $\mathbb{E}[\text{indexOf}(s, s')]$. Formally:

$$\forall \mathbb{m}^\# \in M^\#, \forall s, s' \in \Sigma . \mathbb{E}[\text{indexOf}(s, s')]\gamma(\mathbb{m}^\#) \subseteq \gamma(\mathbb{E}^\#[\text{indexOf}(s, s')]\mathbb{m}^\#)$$

Proof. The collecting semantics of indexOf is defined as the additive lift of the con-

crete one: $\mathbb{E}[\text{indexOf}(s, s')]\mathfrak{m} = \{ i \mid i = \text{indexOf}(\sigma, \sigma'), \sigma \in \mathcal{L}, \sigma' \in \mathcal{L}' \}$, where $\mathbb{E}[s]\mathfrak{m} = \mathcal{L} \in \wp(\Sigma^*)$, $\mathbb{E}[s']\mathfrak{m} = \mathcal{L}' \in \wp(\Sigma^*)$ and $\text{indexOf} : \Sigma^* \times \Sigma^* \rightarrow \mathbb{N}$ corresponds to the concrete semantics of Figure 6.2. Let us suppose that $\mathbb{E}^\#[\![s]\!]\mathfrak{m}^\# = A \cdot \gamma(A) = \mathcal{L}$ and $\mathbb{E}^\#[\![s']]\!]\mathfrak{m}^\# = A' \cdot \gamma(A') = \mathcal{L}'$, where $A, A' \in \mathcal{TF}_{A/\equiv}$ and $\mathcal{L}, \mathcal{L}' \in \wp(\Sigma^*)$. Note that, by definition of the concrete semantics, we have that $\mathbb{E}[\text{indexOf}(s, s')]\gamma(\mathfrak{m}^\#) \subseteq \gamma([-1, +\infty])$. When A or A' are cyclic or A' has a T transition, the abstract semantics returns the interval $[-1, +\infty]$, guaranteeing soundness. We thus continue by assuming that A and A' are not cyclic and A' has no T transitions (i.e., \mathcal{L}' is finite). We split the proof following the possible concrete values. When $\mathbb{E}[\text{indexOf}(s, s')]\gamma(\mathfrak{m}^\#) = \{-1\}$, no string of \mathcal{L}' is contained in any string of \mathcal{L} . Formally:

$$\begin{aligned} & \mathbb{E}[\text{indexOf}(s, s')]\gamma(\mathfrak{m}^\#) = \{-1\} \\ \Leftrightarrow & \forall \sigma' \in \mathcal{L}' \nexists \sigma \in \mathcal{L} . \sigma' \curvearrow_s \sigma && \text{\{necessary condition\}} \\ \Leftrightarrow & \mathbb{E}^\#[\text{indexOf}(s, s')]\mathfrak{m}^\# = [-1, -1] && \text{\{def. } \mathbb{E}^\#, 2^{\text{nd}} \text{ case\}} \end{aligned}$$

As $\gamma_{\text{Intv}}([-1, -1]) = \{-1\}$, soundness is met. Otherwise, $\mathbb{E}[\text{indexOf}(s, s')]\gamma(\mathfrak{m}^\#) = I \subseteq \{ n \in \mathbb{Z} \mid n \geq -1 \}$. $\exists i \in I . i \geq 0$. This implies $\exists \sigma \in \mathcal{L}(A), \sigma' \in \mathcal{L}(A') . \sigma' \curvearrow_s \sigma$, as the collecting semantics returns at least one value that is not -1 . Here, the abstract semantics relies on function IO that computes an interval for each string $\sigma' \in \mathcal{L}(A')$, lubbing the results together. Hence, it is enough to prove the correctness of IO. Given $\sigma' \in \mathcal{L}'$, let us define the set $I_{\sigma'} \subseteq I = \{ i \mid i = \text{indexOf}(\sigma, \sigma'), \sigma \in \mathcal{L} \}$ of positions where σ' can be found in \mathcal{L} and let $m, M \in I_{\sigma'}$ be the minimal and the maximal elements of $I_{\sigma'}$. Therefore, it is sufficient to prove that $\gamma_{\text{Intv}}([m, M]) \subseteq \gamma_{\text{Intv}}([i, j])$, where $[i, j] = \text{IO}(A, \sigma')$. For soundness to hold, $i \leq m$ and $M \leq j$ must be true, according to γ_{Intv} . We first prove $i \leq m$, identifying two cases. If $m = -1$:

$$\begin{aligned} & -1 \in I_{\sigma'} \\ \Leftrightarrow & \exists \sigma \in \mathcal{L} . \sigma' \curvearrow_s \sigma && \text{\{necessary condition\}} \\ \Leftrightarrow & \exists \pi \in \text{paths}(A) . \sigma' \curvearrow_s \sigma_\pi && \text{\{automata lift\}} \\ \Leftrightarrow & i = -1 && \text{\{def. } i, 1^{\text{st}} \text{ case\}} \end{aligned}$$

Instead, if $m > -1$:

$$\begin{aligned} & m = \min I_{\sigma'}, m \neq -1 \\ \Leftrightarrow & \exists \sigma \in \mathcal{L} . \sigma_m \dots \sigma_{m+|\sigma'|} = \sigma' \\ & \wedge \forall \sigma \in \mathcal{L} \nexists n < m . \sigma_n \dots \sigma_{n+|\sigma'|} = \sigma' && \text{\{necessary cond.\}} \\ \Leftrightarrow & \exists \pi \in \text{paths}(A) . \exists \sigma_f \in \text{Flat}(\sigma_\pi) . \sigma_{f_m} \dots \sigma_{f_{m+|\sigma'|}} = \sigma' \\ & \wedge \forall \pi \in \text{paths}(A) \forall \sigma_f \in \text{Flat}(\sigma_\pi) . \sigma_{f_k} \dots \sigma_{f_{k+|\sigma'|}} = \sigma' \Rightarrow k \geq m && \text{\{automata lift\}} \\ \Leftrightarrow & i = \min k = m && \text{\{def. } i, 2^{\text{nd}} \text{ case\}} \end{aligned}$$

We now prove that $M \leq j$, identifying three cases. If $M = -1$:

$$\begin{aligned}
& I_{\sigma'} = \{-1\} \\
\iff \forall \sigma \in \mathcal{L} . \sigma' \not\rightsquigarrow_s \sigma & \quad \{\text{necessary condition}\} \\
\iff \forall \pi \in \text{paths}(\mathbf{A}) . \sigma' \not\rightsquigarrow_s \sigma_\pi & \quad \{\text{automata lift}\} \\
\iff j = 1 & \quad \{\text{def. } j, 1^{\text{st}} \text{ case}\}
\end{aligned}$$

Instead, if $M > -1$ and $\forall \pi \in \text{paths}(\mathbf{A}) . \pi$ reads $\sigma \implies \pi$ does not read T before σ :

$$\begin{aligned}
& M = \max I_{\sigma'} \\
\iff \exists \sigma \in \mathcal{L} . \sigma_M \dots \sigma_{M+|\sigma'|} = \sigma' \\
& \quad \wedge \forall \sigma \in \mathcal{L} \exists n > M . \sigma_n \dots \sigma_{n+|\sigma'|} = \sigma' & \quad \{\text{necessary cond.}\} \\
\iff \exists \pi \in \text{paths}(\mathbf{A}) . \exists \sigma_f \in \text{Flat}(\sigma_\pi) . \sigma_{f_M} \dots \sigma_{f_{M+|\sigma'|}} = \sigma' \\
& \quad \wedge \forall \pi \in \text{paths}(\mathbf{A}) \forall \sigma_f \in \text{Flat}(\sigma_\pi) . \sigma_{f_k} \dots \sigma_{f_{k+|\sigma'|}} = \sigma' \implies k \leq M & \quad \{\text{automata lift}\} \\
\iff j = \max k = M & \quad \{\text{def. } j, 3^{\text{rd}} \text{ case}\}
\end{aligned}$$

Finally, if $M > -1$ and $\exists \pi \in \text{paths}(\mathbf{A}) . \pi$ reads T before σ , $j = +\infty$ by the 2nd case of the definition of j , that is thus greater than M . As both inequalities are always satisfied, we can conclude that soundness is met in all cases. \square

A.5 Soundness of Replace

Theorem 6. $\mathbb{E}^\#[\text{replace}(s, s_s, s_r)]$ is a sound abstraction of $\mathbb{E}[\text{replace}(s, s_s, s_r)]$.
Formally:

$$\forall m^\# \in M^\#, \forall s, s_s, s_r \in \text{SE} . \mathbb{E}[\text{replace}(s, s_s, s_r)]\gamma(m^\#) \subseteq \gamma(\mathbb{E}^\#[\text{replace}(s, s_s, s_r)]m^\#)$$

Proof. The collecting semantics of `replace` is defined as the additive lift of the concrete one, that is $\mathbb{E}[\text{replace}(s, s_s, s_r)]m = \{ \sigma' \mid \sigma' = \text{replace}(\sigma, \sigma_s, \sigma_r), \sigma \in \mathcal{L}, \sigma_s \in \mathcal{L}_s, \sigma_r \in \mathcal{L}_r \}$, where $\mathbb{E}[s]m = \mathcal{L} \in \wp(\Sigma^*)$, $\mathbb{E}[s_s]m = \mathcal{L}_s \in \wp(\Sigma^*)$, $\mathbb{E}[s_r]m = \mathcal{L}_r \in \wp(\Sigma^*)$ and $\text{replace} : \Sigma^* \times \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ corresponds to the concrete semantics of Figure 6.2. Let us suppose that $\mathbb{E}^\#[s]m^\# = \mathbf{A} . \gamma(\mathbf{A}) = \mathcal{L}$, $\mathbb{E}^\#[s_s]m^\# = \mathbf{A}_s . \gamma(\mathbf{A}_s) = \mathcal{L}_s$, and $\mathbb{E}^\#[s_r]m^\# = \mathbf{A}_r . \gamma(\mathbf{A}_r) = \mathcal{L}_r$, where $\mathbf{A}, \mathbf{A}_s, \mathbf{A}_r \in \mathcal{TFA}_{/\equiv}$ and $\mathcal{L}, \mathcal{L}_s, \mathcal{L}_r \in \wp(\Sigma^*)$. When \mathbf{A} or \mathbf{A}_s have a cycle or have a T-transition, our semantics returns $\text{Min}(\text{T})$ and is thus trivially sound. Otherwise, when no replacement happens, (i.e., $\mathbb{E}[\text{replace}(\mathcal{L}, \mathcal{L}_s, \mathcal{L}_r)]m = \mathcal{L}$):

$$\begin{aligned}
& \mathbb{E}[\text{replace}(\mathcal{L}, \mathcal{L}_s, \mathcal{L}_r)] = \mathcal{L} \\
\iff \forall \sigma_s \in \mathcal{L}_s \exists \sigma \in \mathcal{L} . \sigma_s \not\rightsquigarrow_s \sigma & \quad \{\text{necessary condition}\} \\
\iff \mathbb{E}[\text{replace}(\mathbf{A}, \mathbf{A}_s, \mathbf{A}_r)] = \mathbf{A} & \quad \{\text{def. } \mathbb{E}^\#, 1^{\text{st}} \text{ case}\}
\end{aligned}$$

Instead, when at least one replacement happens, the semantics returns the lub of several applications of RP ranging over all possible combinations of strings in \mathcal{L} and \mathcal{L}_s , that can be thoroughly explored since \mathcal{L} and \mathcal{L}_s are finite sets. Once RP has been proven correct, soundness naturally follows according to the properties of lub. We thus prove that $\forall \sigma \in \mathcal{L}, \forall \sigma_s \in \mathcal{L}_s, \forall \pi \in \text{paths}(A). \sigma_\pi = \sigma$:

$$\mathbb{E}[\text{replace}(\{\sigma\}, \{\sigma_s\}, \mathcal{L}_r)] \subseteq \gamma(\text{RP}(\pi, \sigma_s, A_r))$$

Specifically, RP removes every occurrence of σ_s in π (lines 7 and 8, where states and transitions composing σ_s are removed from the resulting automaton), substituting them with a copy of the replace automaton (line 4) that is connected to the path with ϵ -transitions. This means that all $\sigma' \curvearrowright_s \sigma_\pi . \sigma' = \sigma_s$ are substituted with *all* the strings recognized by A_r . We can then characterize the language of the automaton returned by RP as $\{\sigma_\pi[\sigma_s/\sigma_r] \mid \sigma_r \in \mathcal{L}(A_r)\}$. Soundness is thus ensured:

$$\begin{aligned} \mathbb{E}[\text{replace}(\{\sigma\}, \{\sigma_s\}, \mathcal{L}_r)] &= \{\sigma[\sigma_s/\sigma_r] \mid \sigma_r \in \mathcal{L}_r\} \\ &\subseteq \{\sigma_\pi[\sigma_s/\sigma_r] \mid \sigma_r \in \mathcal{L}(A_r)\} && \text{\{automata lift\}} \\ &= \gamma(\text{RP}(\pi, \sigma_s, A_r)) && \text{\{def. RP\}} \end{aligned}$$

Soundness is thus proven as the result on individual strings can be lifted to languages, and since the A_r passed to RP is an over-approximation of the concrete strings it represents (as the semantics performs a may-replacement whenever $|\mathcal{L}_s| > 1$). \square

A.6 Soundness of Substring

Theorem 7. $\mathbb{E}^\#[\text{substr}(s, a_1, a_2)]$ is a sound abstraction of $\mathbb{E}[\text{substr}(s, a_1, a_2)]$.
Formally:

$$\begin{aligned} \forall m^\# \in \mathbb{M}^\#, \forall s \in \text{SE}, \forall a_1, a_2 \in \text{AE}. \\ \mathbb{E}[\text{substr}(s, a_1, a_2)]\gamma(m^\#) \subseteq \gamma(\mathbb{E}^\#[\text{substr}(s, a_1, a_2)]m^\#) \end{aligned}$$

Proof. The collecting semantics of `substr` is defined as the additive lift of the concrete one, that is $\mathbb{E}[\text{substr}(s, a_1, a_2)]m = \{\sigma \mid \sigma = \text{substr}(\sigma, i, j), \sigma \in \mathcal{L}, i \in I, j \in J\}$, where $\mathbb{E}[s]m = \mathcal{L} \in \wp(\Sigma^*)$, $I = \mathbb{E}[a_1]m$, $J = \mathbb{E}[a_2]m$ and $\text{substr} : \Sigma^* \times \mathbb{N} \times \mathbb{N} \rightarrow \Sigma^*$ corresponds to the concrete semantics of Figure 6.2. Without loss of generality, we can prove the semantics to be sound when $\mathbb{E}^\#[a_1]m^\# = [i, i]$ and $\mathbb{E}^\#[a_2]m^\# = [j, j]$, with $i, j \in \mathbb{N}, 0 \leq i \leq j$, as the abstract semantics lifts such result to non-singleton intervals applying lub. Let us suppose that $\mathbb{E}^\#[s]m^\# = A . \gamma(A) = \mathcal{L}$, and that $\mathbb{E}^\#[a_1]m^\# = [i, i]$ and $\mathbb{E}^\#[a_2]m^\# = [j, j]$, with $i, j \in \mathbb{N}$. Furthermore, let $r \equiv A$ be the regular expression equivalent to A . We can thus prove soundness of the

semantics by proving the following:

$$\mathbb{E}[\text{substr}(\mathcal{L}, \{i\}, \{j\})] \gamma(\mathfrak{m}^\#) \subseteq \gamma(\text{Min}(\{\sigma \mid (\sigma, 0, 0) \in \text{Sb}(\mathfrak{r}, i, j - i)\})).$$

Soundness is proven by structural induction over the structure of the regular expression, referencing the lines of Algorithm 5 that are involved in the computation as § x , where x is the line number. Moreover, when $\text{Sb}(\mathfrak{r}, i, j)$ produces the set $S = \{(\sigma_1, i_1, j_1), \dots, (\sigma_n, i_n, j_n)\}$, we denote the automaton $\text{Min}(\{\sigma \mid (\sigma, 0, 0) \in S\})$ as either, abusing notation, $\text{Min}(\text{Sb}(\mathfrak{r}, i, j))$ or $\text{Min}(\{(\sigma_1, i_1, j_1), \dots, (\sigma_n, i_n, j_n)\})$. With the latter notation, we abuse notation writing $\sigma_i \notin \text{Sb}$ to denote that (σ_i, i_i, j_i) is not in the final result of Sb .

Base cases

▷ $\mathfrak{r} = \emptyset$ ($\mathcal{L}(\mathfrak{r}) = \emptyset$):

$$\begin{aligned} \mathbb{E}[\text{substr}(\emptyset, \{i\}, \{j\})] &= \emptyset \\ &= \gamma(\text{Min}(\emptyset)) && \text{\{automata lift\}} \\ &= \gamma(\text{Min}(\text{Sb}(\emptyset, i, j - i))) && \text{\{§2\}} \end{aligned}$$

▷ $\mathfrak{r} = \sigma \in \Sigma^*$: here, we identify three cases. If $i \leq j < |\sigma|$:

$$\begin{aligned} \mathbb{E}[\text{substr}(\{\sigma\}, \{i\}, \{j\})] &= \{\sigma_i \dots \sigma_j\} \\ &= \gamma(\text{Min}(\{\sigma_i \dots \sigma_j\})) && \text{\{automata lift\}} \\ &= \gamma(\text{Min}(\text{Sb}(\{\sigma\}, i, j - i))) && \text{\{§6\}} \end{aligned}$$

Instead, when $i > |\sigma|$:

$$\begin{aligned} \mathbb{E}[\text{substr}(\{\sigma\}, \{i\}, \{j\})] &= \emptyset \\ &= \gamma(\text{Min}(\{\epsilon, i - |\sigma|, j - i\})) && \text{\{i - |\sigma| > 0 \implies \epsilon \notin \text{Sb}\}} \\ &= \gamma(\text{Min}(\text{Sb}(\{\sigma\}, i, j - i))) && \text{\{§4\}} \end{aligned}$$

computing an empty partial substring (that is still concretized as the empty set of strings), but taking into account that σ has been read $(i - |\sigma|)$ and no character from σ has been taken $(j - i)$. Finally, if $i < |\sigma|$ and $j > |\sigma|$ (where $k = j - |\sigma| + i$):

$$\begin{aligned} \mathbb{E}[\text{substr}(\{\sigma\}, \{i\}, \{j\})] &= \emptyset \\ &= \gamma(\text{Min}(\{\sigma_i \dots \sigma_{|\sigma|-1}, 0, k\})) && \text{\{k > 0 \implies \sigma_i \dots \sigma_{|\sigma|-1} \notin \text{Sb}\}} \\ &= \gamma(\text{Min}(\text{Sb}(\{\sigma\}, i, j - i))) && \text{\{§5\}} \end{aligned}$$

computing an partial substring (that is still concretized as the empty set of strings)

that is a suffix of σ , and noting that $j - (i - |\sigma_i \dots \sigma_{|\sigma|-1})$ characters still have to be read before completing the substring.

▷ $r = T$:

$$\begin{aligned}
\mathbb{E}[\text{substr}(\Sigma^*, \{i\}, \{j\})] &= \{ \sigma \mid |\sigma| = j - i \} \\
&= \gamma(\text{Min}(\{(\bullet^{j-i}, 0, 0)\})) && \text{\{automata lift\}} \\
&\cup \gamma(\text{Min}(\{(\bullet^l, 0, j-l)\}), l < j-i) && \text{\{ } j-l > 0 \implies \bullet^l \notin \text{Sb} \}} \\
&\cup \gamma(\text{Min}(\{(\epsilon, i-l, j)\}), 0 \leq l \leq i) && \text{\{ } i-l > 0 \implies \epsilon \notin \text{Sb} \}} \\
&= \gamma(\text{Min}(\text{Sb}(T, i, j-i))) && \text{\{§8, §9\}}
\end{aligned}$$

where, for the sake of clarity, strings returned by Sb are split into three sets, the first $\{(\bullet^{j-i}, 0, 0)\}$ simulating substrings generated when $i, j \leq |\sigma|$, the second one $\{(\bullet^l, 0, j-l)\}$ representing partial substrings when $i \geq |\sigma|$, and the third symbolizing partial substrings generated when $i < |\sigma| \wedge j \geq |\sigma|$. Note that only strings from the first set are part of the final concretization, while partial substrings from the second and third automata only serve in computations of successive substrings.

Inductive steps

▷ $r = r_1 \| r_2$: let $\mathcal{L}, \mathcal{L}_1, \mathcal{L}_2 \in \wp(\Sigma^*)$ be the languages recognized by r, r_1 and r_2 , respectively. It is easy to see that $\mathbb{E}[\text{substr}(\mathcal{L}, \{i\}, \{j\})] = \mathbb{E}[\text{substr}(\mathcal{L}_1, \{i\}, \{j\})] \cup \mathbb{E}[\text{substr}(\mathcal{L}_2, \{i\}, \{j\})]$. We assume $\mathbb{E}[\text{substr}(\mathcal{L}_1, \{i\}, \{j\})] \subseteq \gamma(\text{Min}(\text{Sb}(r_1, i, j-i)))$ and $\mathbb{E}[\text{substr}(\mathcal{L}_2, \{i\}, \{j\})] \subseteq \gamma(\text{Min}(\text{Sb}(r_2, i, j-i)))$ to hold for inductive hypothesis. We then prove soundness with the following:

$$\begin{aligned}
\mathbb{E}[\text{substr}(\mathcal{L}, \{i\}, \{j\})] &= \mathbb{E}[\text{substr}(\mathcal{L}_1, \{i\}, \{j\})] \\
&\cup \mathbb{E}[\text{substr}(\mathcal{L}_2, \{i\}, \{j\})] \\
&\subseteq \gamma(\text{Min}(\text{Sb}(r_1, i, j-i))) && \text{\{ind. hp.\}} \\
&\cup \gamma(\text{Min}(\text{Sb}(r_2, i, j-i))) && \text{\{ind. hp.\}} \\
&= \gamma(\text{Min}(\text{Sb}(r_1 \| r_2, i, j-i))) && \text{\{§21\}}
\end{aligned}$$

▷ $r = r_1 r_2$: let $\mathcal{L}, \mathcal{L}_1, \mathcal{L}_2 \in \wp(\Sigma^*)$ be the languages recognized by r, r_1 and r_2 , respectively. The concrete semantics is the union of two sets: $\mathbb{E}[\text{substr}(\mathcal{L}_1, \{i\}, \{j\})]$ (i.e., substrings that are fully contained in \mathcal{L}_1), and $\mathbb{E}[\text{substr}(\mathcal{L}_1 \cdot \mathcal{L}_2, \{i\}, \{j\})]$ (i.e., substrings that straddle \mathcal{L}_1 and \mathcal{L}_2). We prove soundness assuming the inductive hypotheses $\mathbb{E}[\text{substr}(\mathcal{L}_1, \{i\}, \{j\})] \subseteq \gamma(\text{Min}(\text{Sb}(r_1, i, j-i)))$ and $\mathbb{E}[\text{substr}(\mathcal{L}_2, \{i\}, \{j\})] \subseteq \gamma(\text{Min}(\text{Sb}(r_2, i, j-i)))$:

$$\begin{aligned}
\mathbb{E}[\text{substr}(\mathcal{L}, \{i\}, \{j\})] &= \mathbb{E}[\text{substr}(\mathcal{L}_1, \{i\}, \{j\})] \\
&\cup \mathbb{E}[\text{substr}(\mathcal{L}_1 \cdot \mathcal{L}_2, \{i\}, \{j\})] \\
&\subseteq \gamma(\text{Min}(\text{Sb}(r_1, i, j-i))) && \text{\{ind. hp.\}}
\end{aligned}$$

$$\begin{aligned} & \cup \gamma(\text{Min}(\{(\sigma_1^1 \cdot \sigma_2^1, i_2^1, j_2^1), \dots, (\sigma_1^n \cdot \sigma_2^n, i_2^n, j_2^n)\})) && \text{\textcircled{ind. hp.}} \\ & = \gamma(\text{Min}(\text{Sb}(\mathbf{r}_1 \mathbf{r}_2, i, j - i))) && \text{\textcircled{\S 1, \S 16, \S 18}} \end{aligned}$$

where, for the sake of clarity, strings returned by Sb are split in two sets, the first ($\text{Sb}(\mathbf{r}_1, i, j - i)$) corresponding to substrings that entirely contained into \mathbf{r}_1 , the second one ($\{(\sigma_1^1 \cdot \sigma_2^1, i_2^1, j_2^1), \dots, (\sigma_1^n \cdot \sigma_2^n, i_2^n, j_2^n)\}$) that models substrings straddling \mathbf{r}_1 and \mathbf{r}_2 , where $\forall i. (\sigma_1^i, i_1^i, j_1^i) \in \text{Sb}(\mathbf{r}_1, i, j - i), j_1^i \neq 0 \wedge (\sigma_2^i, i_2^i, j_2^i) \in \text{Sb}(\mathbf{r}_2, i_1^i, j_1^i)$. Strings in the latter set are built by offsetting substrings of \mathbf{r}_2 by the length of the substrings of \mathbf{r}_1 .

▷ $\mathbf{r} = (\mathbf{r}_1)^*$. The proof of this case is similar to the one for concatenation, since $(\mathbf{r}_1)^*$ can be seen as an (undefined) concatenation of the regular expression \mathbf{r}_1 , and is thus left implicit. \square

Bibliography

- [1] Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Dolby, J., Janku, P., Lin, H., Holík, L., Wu, W.: Efficient handling of string-number conversion. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 943–957. ACM (2020). <https://doi.org/10.1145/3385412.3386034>
- [2] Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8559, pp. 150–166. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_10
- [3] Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janku, P.: Chain-free string constraints. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 277–293. Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_16
- [4] Ali, K., Lhoták, O.: Averroes: Whole-program analysis without the whole program. In: Castagna, G. (ed.) ECOOP 2013 – Object-Oriented Programming, pp. 378–400. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39038-8_16
- [5] Allen, F.E.: Control flow analysis. In: Proceedings of a Symposium on Compiler Optimization. p. 1–19. Association for Computing Machinery, New York, NY, USA (1970). <https://doi.org/10.1145/800028.808479>
- [6] Almashfi, N., Lu, L.: Precise string domain for analyzing javascript arrays and objects. In: 2020 3rd International Conference on Information and Computer Technologies (ICICT). pp. 17–23 (2020). <https://doi.org/10.1109/ICICT50521.2020.00011>
- [7] Amadini, R., Gange, G., Stuckey, P.J.: Dashed strings for string constraint solving. *Artificial Intelligence* **289**, 103368 (2020). <https://doi.org/10.1016/j.artint.2020.103368>
- [8] Andersen, L.O.: Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen (1994), <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.6502&rep=rep1&type=pdf>

- [9] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A.D., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S.W., Yellick, J.: Hyperledger fabric: A distributed operating system for permissioned blockchains. In: Oliveira, R., Felber, P., Hu, Y.C. (eds.) Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23–26, 2018. pp. 30:1–30:15. ACM (2018). <https://doi.org/10.1145/3190508.3190538>
- [10] Antonopoulos, A.M., Wood, G.: Mastering Ethereum: Building Smart Contracts and Dapps. O'Reilly (2018)
- [11] Antonopoulos, A.M.: Mastering Bitcoin: Programming the Open Blockchain. O'Reilly Media, Inc., 2nd edn. (2017)
- [12] Arceri, V., Maffeis, S.: Abstract domains for type juggling. *Electron. Notes Theor. Comput. Sci.* **331**, 41–55 (2017). <https://doi.org/10.1016/j.entcs.2017.02.003>
- [13] Arceri, V., Mastroeni, I.: A sound abstract interpreter for dynamic code. In: Hung, C., Cerný, T., Shin, D., Bechini, A. (eds.) SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020. pp. 1979–1988. ACM (2020). <https://doi.org/10.1145/3341105.3373964>
- [14] Arceri, V., Mastroeni, I., Xu, S.: Static analysis for ecmascript string manipulation programs. *Appl. Sci.* **10**, 3525 (2020). <https://doi.org/10.3390/app10103525>
- [15] Arceri, V., Olliaro, M., Cortesi, A., Mastroeni, I.: Completeness of abstract domains for string analysis of javascript programs. In: Hierons, R.M., Mosbah, M. (eds.) Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11884, pp. 255–272. Springer (2019). https://doi.org/10.1007/978-3-030-32505-3_15
- [16] Arzt, S., Bodden, E.: Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). pp. 725–735 (2016). <https://doi.org/10.1145/2884781.2884816>
- [17] Bacon, D.F., Sweeney, P.F.: Fast static analysis of c++ virtual function calls. In: Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 324–341. OOPSLA '96, Association for Computing Machinery, New York, NY, USA (1996). <https://doi.org/10.1145/236337.236371>

-
- [18] Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming* **72**(1), 3–21 (2008). <https://doi.org/10.1016/j.scico.2007.08.001>, special Issue on Second issue of experimental software and toolkits (EST)
- [19] Ball, T., Rajamani, S.K.: Slic: A specification language for interface checking (of c). Tech. rep., Technical Report MSR-TR-2001-21, Microsoft Research (2001)
- [20] Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T. (eds.) *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. pp. 49–69. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30569-9_3
- [21] Bartzis, C., Bultan, T.: Widening arithmetic automata. In: Alur, R., Peled, D.A. (eds.) *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings. Lecture Notes in Computer Science*, vol. 3114, pp. 321–333. Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_25
- [22] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* **53**(2), 66–75 (feb 2010). <https://doi.org/10.1145/1646353.1646374>
- [23] Brotsis, S., Kolokotronis, N., Limniotis, K., Bendiab, G., Shiaeles, S.: On the security and privacy of hyperledger fabric: Challenges and open issues. In: *2020 IEEE World Congress on Services (SERVICES)*. pp. 197–204 (2020). <https://doi.org/10.1109/SERVICES48979.2020.00049>
- [24] Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains. Ph.D. thesis, University of Guelph (2016), <https://atrium.lib.uoguelph.ca/xmlui/handle/10214/9769>
- [25] Buchman, E.: Byzantine fault tolerant state machine replication in any programming language. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. p. 546. PODC '19, Association for Computing Machinery, New York, NY, USA (2019). <https://doi.org/10.1145/3293611.3338023>
- [26] Burato, E., Ferrara, P., Spoto, F.: Security analysis of the owasp benchmark with julia. *Proceedings of ITASEC* **17** (2017)
- [27] Buro, S., Crole, R.L., Mastroeni, I.: On multi-language abstraction. In: Pichardie, D., Sighireanu, M. (eds.) *Static Analysis*. pp. 310–332. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-65474-0_14
-

- [28] Buro, S., Mastroeni, I.: On the multi-language construction. In: Caires, L. (ed.) *Programming Languages and Systems*. pp. 293–321. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17184-1_11
- [29] Buterin, V.: Ethereum whitepaper (2013), available at <https://ethereum.org/en/whitepaper/>
- [30] Centonze, P., Naumovich, G., Fink, S.J., Pistoia, M.: Role-based access control consistency validation. In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. p. 121–132. ISSTA '06, Association for Computing Machinery, New York, NY, USA (2006). <https://doi.org/10.1145/1146238.1146253>
- [31] Chatterjee, K., Goharshady, A.K., Pourdamghani, A.: Probabilistic smart contracts: Secure randomness on the blockchain. In: *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019*. pp. 403–412. IEEE (2019). <https://doi.org/10.1109/BLOC.2019.8751326>
- [32] Chen, I.Y., Johansson, F.D., Sontag, D.: Why is my classifier discriminatory? In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. p. 3543–3554. NIPS'18, Curran Associates Inc., Red Hook, NY, USA (2018), <https://proceedings.neurips.cc/paper/2018/file/1f1baa5b8edac74eb4eaa329f14a0361-Paper.pdf>
- [33] Chen, L.: Microservices: Architecting for continuous delivery and devops. In: *2018 IEEE International Conference on Software Architecture (ICSA)*. pp. 39–397 (2018). <https://doi.org/10.1109/ICSA.2018.00013>
- [34] Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.* **3**(POPL), 49:1–49:30 (2019). <https://doi.org/10.1145/3290362>
- [35] Choi, T., Lee, O., Kim, H., Doh, K.: A practical string analyzer by the widening approach. In: Kobayashi, N. (ed.) *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*. Lecture Notes in Computer Science, vol. 4279, pp. 374–388. Springer (2006). https://doi.org/10.1007/11924661_23
- [36] Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. Lecture Notes in Computer Science, vol. 2694, pp. 1–18. Springer (2003). https://doi.org/10.1007/3-540-44898-5_1

-
- [37] Commercio.network: Commercio.network - white paper (2022), <https://commercio.network/project/>, accessed: March 9 2022
- [38] Cortesi, A., Costantini, G., Ferrara, P.: A survey on product operators in abstract interpretation. *Electronic Proceedings in Theoretical Computer Science* **129**, 325–336 (sep 2013). <https://doi.org/10.4204/eptcs.129.19>
- [39] Cortesi, A., Olliaro, M.: M-string segmentation: A refined abstract domain for string analysis in c programs. In: Pang, J., Zhang, C., He, J., Weng, J. (eds.) 2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018. pp. 1–8. IEEE Computer Society (2018). <https://doi.org/10.1109/TASE.2018.00009>
- [40] Cortesi, A., Zanioli, M.: Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures* **37**(1), 24–42 (2011). <https://doi.org/10.1016/j.cl.2010.09.001>
- [41] Costantini, G., Ferrara, P., Cortesi, A.: A suite of abstract domains for static analysis of string values. *Software: Practice and Experience* **45**(2), 245–287 (2015). <https://doi.org/10.1002/spe.2218>
- [42] Cousot, P.: *Principles of Abstract Interpretation*. MIT Press (2021)
- [43] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977). <https://doi.org/10.1145/512950.512973>
- [44] Cousot, P., Cousot, R.: Constructive versions of tarski’s fixed point theorems. *Pacific journal of Mathematics* **82**(1), 43–57 (1979). <https://doi.org/10.2140/pjm.1979.82.43>
- [45] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, USA, January 1979. pp. 269–282. ACM Press (1979). <https://doi.org/10.1145/567752.567778>
- [46] Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. *The Journal of Logic Programming* **13**(2), 103–179 (1992). [https://doi.org/https://doi.org/10.1016/0743-1066\(92\)90030-7](https://doi.org/https://doi.org/10.1016/0743-1066(92)90030-7)
- [47] Cousot, P., Cousot, R.: Abstract interpretation frameworks. *Journal of Logic and Computation* **2**(4), 511–547 (08 1992). <https://doi.org/10.1093/logcom/2.4.511>
-

- [48] Cousot, P., Cousot, R.: Modular static program analysis. In: Horspool, R.N. (ed.) *Compiler Construction*. pp. 159–179. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_13
- [49] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The *astree* analyzer. In: Sagiv, M. (ed.) *Programming Languages and Systems*. pp. 21–30. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_3
- [50] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, USA, January 1978. pp. 84–96. ACM Press (1978). <https://doi.org/10.1145/512760.512770>
- [51] D’Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Jagannathan, S., Sewell, P. (eds.) *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14*, San Diego, CA, USA, January 20–21, 2014. pp. 541–554. ACM (2014). <https://doi.org/10.1145/2535838.2535849>
- [52] Davis, M., Sigal, R., Weyuker, E.J.: *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Elsevier (1994). <https://doi.org/10.1016/C2013-0-10567-3>
- [53] Davis, M., Sigal, R., Weyuker, E.J.: *Computability, complexity, and languages: fundamentals of theoretical computer science*. Elsevier (1994)
- [54] Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Tokoro, M., Pareschi, R. (eds.) *ECOOP’95 – Object-Oriented Programming, 9th European Conference*, Århus, Denmark, August 7–11, 1995. pp. 77–101. Springer Berlin Heidelberg, Berlin, Heidelberg (1995). https://doi.org/10.1007/3-540-49538-X_5
- [55] Denning, D.E.: A lattice model of secure information flow. *Commun. ACM* **19**(5), 236–243 (may 1976). <https://doi.org/10.1145/360051.360056>
- [56] Distefano, D., Fähndrich, M., Logozzo, F., O’Hearn, P.W.: Scaling static analyses at facebook. *Communications of the ACM* **62**(8), 62–70 (2019). <https://doi.org/10.1145/3338112>
- [57] D’Silva, V.: *Widening for Automata*. Master’s thesis, Institut Für Informatik, Universität Zürich (2006), <https://www.merlin.uzh.ch/contributionDocument/download/2374>
- [58] ebuchman: *Cosmos-sdk vulnerability retrospective: Security advisory jackfruit, october 12, 2021* (2021), <https://forum.cosmos.network/t/cosmos->

sdk-vulnerability-retrospective-security-advisory-jackfruit-october-12-2021/5349, accessed: 23-12-2021

- [59] Emrath, P.A., Padua, D.A.: Automatic detection of nondeterminacy in parallel programs. In: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging. p. 89–99. PADD '88, Association for Computing Machinery, New York, NY, USA (1988). <https://doi.org/10.1145/68210.69224>
- [60] Ernst, M.D., Lovato, A., Macedonio, D., Spiridon, C., Spoto, F.: Boolean formulas for the static identification of injection attacks in java. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 130–145. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). https://doi.org/978-3-662-48899-7_10
- [61] Ferrara, P.: A generic framework for heap and value analyses of object-oriented programming languages. *Theoretical Computer Science* **631**, 43–72 (2016). <https://doi.org/10.1016/j.tcs.2016.04.001>
- [62] Ferrara, P., Cortesi, A., Spoto, F.: Cil to java-bytecode translation for static analysis leveraging. In: Proceedings of the 6th Conference on Formal Methods in Software Engineering. p. 40–49. FormaliSE '18, Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3193992.3193994>
- [63] Ferrara, P., Mandal, A.K., Cortesi, A., Spoto, F.: Cross-programming language taint analysis for the iot ecosystem. *Electronic Communications of the EASST* **77** (2019). <https://doi.org/10.14279/tuj.eceasst.77.1104>
- [64] Ferrara, P., Mandal, A.K., Cortesi, A., Spoto, F.: Static analysis for discovering iot vulnerabilities. *Int. J. Softw. Tools Technol. Transf.* **23**(1), 71–88 (2021). <https://doi.org/10.1007/s10009-020-00592-x>
- [65] Ferrara, P., Negrini, L.: Sarl: Oo framework specification for static analysis. In: Christakis, M., Polikarpova, N., Duggirala, P.S., Schrammel, P. (eds.) *Software Verification*. pp. 3–20. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-63618-0_1
- [66] Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: Experiencing lisa. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 1–6. SOAP 2021, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3460946.3464316>
- [67] Ferrara, P., Olivieri, L., Spoto, F.: Tailoring taint analysis to gdpr. In: Medina, M., Mittrakas, A., Rannenber, K., Schweighofer, E., Tsouroulas, N. (eds.) *Privacy Technologies and Policy - 6th Annual Privacy Forum, APF 2018, Barcelona*,

- Spain, June 13-14, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11079, pp. 63–76. Springer (2018). https://doi.org/10.1007/978-3-030-02547-2_4
- [68] Ferrara, P., Olivieri, L., Spoto, F.: Static privacy analysis by flow reconstruction of tainted data. *Int. J. Softw. Eng. Knowl. Eng.* **31**(7), 973–1016 (2021). <https://doi.org/10.1142/S0218194021500303>
- [69] Foschini, L., Gavagna, A., Martuscelli, G., Montanari, R.: Hyperledger fabric blockchain: Chaincode performance analysis. In: 2020 IEEE International Conference on Communications, ICC 2020, Dublin, Ireland, June 7-11, 2020. pp. 1–6. IEEE (2020). <https://doi.org/10.1109/ICC40277.2020.9149080>
- [70] Fosdick, L.D., Osterweil, L.J.: Data flow analysis in software reliability. *ACM Comput. Surv.* **8**(3), 305–330 (sep 1976). <https://doi.org/10.1145/356674.356676>
- [71] Furr, M., Foster, J.S.: Polymorphic type inference for the jni. In: Sestoft, P. (ed.) *Programming Languages and Systems*. pp. 309–324. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/11693024_21
- [72] Furr, M., Foster, J.S.: Checking type safety of foreign function calls. *ACM Trans. Program. Lang. Syst.* **30**(4) (aug 2008). <https://doi.org/10.1145/1377492.1377493>
- [73] Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982. pp. 11–20. IEEE Computer Society (1982). <https://doi.org/10.1109/SP.1982.10014>
- [74] Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: *Proceedings of the 1984 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 29 - May 2, 1984*. pp. 75–87. IEEE Computer Society (1984). <https://doi.org/10.1109/SP.1984.10019>
- [75] Grafberger, S., Guha, S., Stoyanovich, J., Schelter, S.: Mlinspect: A data distribution debugger for machine learning pipelines. In: *Proceedings of the 2021 International Conference on Management of Data*. p. 2736–2739. SIGMOD '21, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3448016.3452759>
- [76] Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. p. 376–386. PLDI '06, Association for Computing Machinery, New York, NY, USA (2006). <https://doi.org/10.1145/1133981.1134026>
- [77] Guth, D.: A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign (2013), <https://hdl.handle.net/2142/45275>

-
- [78] Herndon, T., Ash, M., Pollin, R.: Does high public debt consistently stifle economic growth? a critique of reinhart and roff. *Cambridge Journal of Economics* **38**(2), 257–279 (12 2013). <https://doi.org/10.1093/cje/bet075>
- [79] Hovemeyer, D., Pugh, W.: Finding bugs is easy. *SIGPLAN Not.* **39**(12), 92–106 (dec 2004). <https://doi.org/10.1145/1052883.1052895>
- [80] Hyperledger: Hyperledger fabric documentation, <https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html#what-is-hyperledger-fabric>, accessed: 27-01-2022
- [81] Inc, T.: What is tendermint: A note on determinism (2022), <https://docs.tendermint.com/master/introduction/what-is-tendermint.html#a-note-on-determinism>, accessed: 27-01-2022
- [82] Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) *Computer Aided Verification*. pp. 661–667. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_52
- [83] Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Chakraborty, S., Navas, J.A. (eds.) *Verified Software. Theories, Tools, and Experiments*. pp. 1–18. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-41600-3_1
- [84] Khedker, U.P., Karkare, B.: Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In: Hendren, L. (ed.) *Compiler Construction*. pp. 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78791-4_15
- [85] Köhl, M.A.: An executable structural operational formal semantics for python. *CoRR* **abs/2109.03139** (2021), <https://arxiv.org/abs/2109.03139>
- [86] Kwon, J.: Tendermint: Consensus without mining (2014), available at <https://tendermint.com/static/docs/tendermint.pdf>
- [87] Kwon, J., Buchman, E.: Cosmos whitepaper (2019), available at <https://v1.cosmos.network/resources/whitepaper>
- [88] Leavens, G.T., Baker, A.L., Ruby, C.: Jml: a java modeling language. In: *Formal Underpinnings of Java Workshop '98* (1998)
- [89] Lee, S., Lee, H., Ryu, S.: Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Soft-*

- ware Engineering, p. 127–137. ASE '20, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3324884.3416558>
- [90] Li, S., Tan, G.: Finding bugs in exceptional situations of jni programs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. p. 442–452. CCS '09, Association for Computing Machinery, New York, NY, USA (2009). <https://doi.org/10.1145/1653662.1653716>
- [91] Logozzo, F.: Practical verification for the working programmer with codecontracts and abstract interpretation. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 19–22. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_3
- [92] Lv, P., Wang, Y., Wang, Y., Zhou, Q.: Potential risk detection system of hyperledger fabric smart contract based on static analysis. In: IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5-8, 2021. pp. 1–7. IEEE (2021). <https://doi.org/10.1109/ISCC53001.2021.9631249>
- [93] Madsen, M., Andreasen, E.: String analysis for dynamic field access. In: Cohen, A. (ed.) Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8409, pp. 197–217. Springer (2014). https://doi.org/10.1007/978-3-642-54807-9_12
- [94] McCloskey, B., Reps, T., Sagiv, M.: Statically inferring complex heap, array, and numeric invariants. In: Cousot, R., Martel, M. (eds.) Static Analysis. pp. 71–99. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_6
- [95] Mell, P., Grance, T., et al.: The nist definition of cloud computing. National Institute of Science and Technology, Special Publication **800**(2011), 145 (2011)
- [96] Meyer, B.: Object-oriented software construction, vol. 2. Prentice hall Englewood Cliffs (1997)
- [97] Midtgaard, J., Nielson, F., Nielson, H.R.: A parametric abstract domain for lattice-valued regular expressions. In: Rival, X. (ed.) Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9837, pp. 338–360. Springer (2016). https://doi.org/10.1007/978-3-662-53413-7_17
- [98] Miné, A.: The octagon abstract domain. Higher-order and symbolic computation **19**(1), 31–100 (2006). <https://doi.org/10.1007/s10990-006-8609-1>

- [99] Monat, R.: Static type and value analysis by abstract interpretation of Python programs with native C libraries. Ph.D. thesis, Sorbonne Université (2021), <https://tel.archives-ouvertes.fr/tel-03533030>
- [100] Monat, R., Ouadjaout, A., Miné, A.: A multilanguage static analysis of python programs with native c extensions. In: Drăgoi, C., Mukherjee, S., Namjoshi, K. (eds.) *Static Analysis*. pp. 323–345. Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-88806-0_16
- [101] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), available at <https://bitcoin.org/bitcoin.pdf>
- [102] Namaki, M.H., Floratou, A., Psallidas, F., Krishnan, S., Agrawal, A., Wu, Y., Zhu, Y., Weimer, M.: Vamsa: Automated provenance tracking in data science scripts. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. p. 1542–1551. KDD '20, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3394486.3403205>
- [103] Negrini, L., Arceri, V., Ferrara, P., Cortesi, A.: Twinning automata and regular expressions for string static analysis. In: *Verification, Model Checking, and Abstract Interpretation: 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17–19, 2021, Proceedings*. p. 267–290. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-67067-2_13
- [104] Nerode, A.: Linear automaton transformations. *Proceedings of the American Mathematical Society* **9**(4), 541–544 (1958). <https://doi.org/10.2307/2033204>
- [105] Olivieri, L., Spoto, F., Tagliaferro, F.: On-Chain Smart Contract Verification over Tendermint. In: Bernhard, M., Bracciali, A., Gudgeon, L., Haines, T., Klages-Mundt, A., Matsuo, S., Perez, D., Sala, M., Werner, S. (eds.) *Financial Cryptography and Data Security. FC 2021 International Workshops - CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 12676, pp. 333–347. Springer (2021). https://doi.org/10.1007/978-3-662-63958-0_28
- [106] Olivieri, L., Tagliaferro, F., Arceri, V., Ruaro, M., Negrini, L., Cortesi, A., Ferrara, P., Spoto, F., Talin, E.: Ensuring determinism in blockchain software with golisa: An industrial experience report. In: *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. p. 23–29. SOAP 2022, Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3520313.3534658>
- [107] Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: *Conference Proceedings on Object-Oriented Programming Systems, Languages, and*

- Applications. p. 146–161. OOPSLA '91, Association for Computing Machinery, New York, NY, USA (1991). <https://doi.org/10.1145/117954.117965>
- [108] Park, C., Im, H., Ryu, S.: Precise and scalable static analysis of jquery using a regular expression domain. In: Ierusalimsky, R. (ed.) Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016. pp. 25–36. ACM (2016). <https://doi.org/10.1145/2989225.2989228>
- [109] Pinto, P., Carvalho, T., ao Bispo, J., Ramalho, M.A., ao M.P. Cardoso, J.: Aspect composition for multiple target languages using lara. *Computer Languages, Systems & Structures* **53**, 1–26 (2018). <https://doi.org/10.1016/j.cl.2017.12.003>
- [110] Politz, J.G., Martinez, A., Milano, M., Warren, S., Patterson, D., Li, J., Chitipothu, A., Krishnamurthi, S.: Python: The full monty. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. p. 217–232. OOPSLA '13, Association for Computing Machinery, New York, NY, USA (2013). <https://doi.org/10.1145/2509136.2509536>
- [111] Porru, S., Pinna, A., Marchesi, M., Tonelli, R.: Blockchain-oriented software engineering: Challenges and new directions. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). pp. 169–171 (2017). <https://doi.org/10.1109/ICSE-C.2017.142>
- [112] Preda, M.D., Giacobazzi, R., Lakhotia, A., Mastroeni, I.: Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15–17, 2015. pp. 329–341. ACM (2015). <https://doi.org/10.1145/2676726.2676986>
- [113] Rabin, M.O., Scott, D.: Finite automata and their decision problems. *IBM Journal of Research and Development* **3**(2), 114–125 (1959). <https://doi.org/10.1147/rd.32.0114>
- [114] Reinhart, C.M., Rogoff, K.S.: Growth in a time of debt. *American Economic Review* **100**(2), 573–78 (May 2010). <https://doi.org/10.1257/aer.100.2.573>
- [115] Rice, H.G.: Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* **74**(2), 358–366 (1953). <https://doi.org/10.2307/1990888>
- [116] Rival, X., Mauborgne, L.: The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **29**(5), 26—es (08 2007). <https://doi.org/10.1145/1275497.1275501>

-
- [117] Rival, X., Yi, K.: Introduction to static analysis: an abstract interpretation perspective. Mit Press (2020)
- [118] Sabelfeld, A., Myers, A.: Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* **21**(1), 5–19 (2003). <https://doi.org/10.1109/JSAC.2002.806121>
- [119] Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* **24**(3), 217–298 (may 2002). <https://doi.org/10.1145/514188.514190>
- [120] Sharir, M., Pnueli, A., et al.: Two approaches to interprocedural data flow analysis. New York University. Courant Institute of Mathematical Sciences ... (1978)
- [121] Shatnawi, A., Mili, H., Abdellatif, M., Guéhéneuc, Y.G., Moha, N., Hecht, G., Boussaidi, G.E., Privat, J.: Static code analysis of multilanguage software systems. *ArXiv* (2019). <https://doi.org/10.48550/ARXIV.1906.00815>
- [122] Spoto, F.: Nullness analysis in boolean form. In: 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods. pp. 21–30 (2008). <https://doi.org/10.1109/SEFM.2008.8>
- [123] Spoto, F.: The julia static analyzer for java. In: Rival, X. (ed.) *Static Analysis*. pp. 39–57. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_3
- [124] Spoto, F.: A java framework for smart contracts. In: Bracciali, A., Clark, J., Pintore, F., Rønne, P.B., Sala, M. (eds.) *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC*, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers. *Lecture Notes in Computer Science*, vol. 11599, pp. 122–137. Springer (2019). https://doi.org/10.1007/978-3-030-43725-1_10
- [125] Spoto, F.: Enforcing determinism of java smart contracts. In: Bernhard, M., Bracciali, A., Camp, L.J., Matsuo, S., Maurushat, A., Rønne, P.B., Sala, M. (eds.) *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC*, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers. *Lecture Notes in Computer Science*, vol. 12063, pp. 568–583. Springer (2020). https://doi.org/10.1007/978-3-030-54455-3_40
- [126] Spoto, F., Mesnard, F., Payet, E.: A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.* **32**(3) (mar 2010). <https://doi.org/10.1145/1709093.1709095>
-

- [127] Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., Berg, R.: F4f: Taint analysis of framework-based web applications. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. p. 1053–1068. OOPSLA '11, Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/2048066.2048145>
- [128] Subotić, P., Milikić, L., Stojić, M.: A static analysis framework for data science notebooks. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 13–22 (2022). <https://doi.org/10.1145/3510457.3513032>
- [129] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for java. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 264–280. OOPSLA '00, Association for Computing Machinery, New York, NY, USA (2000). <https://doi.org/10.1145/353171.353189>
- [130] Tan, G., Morrisett, G.: Ilea: Inter-language analysis across java and c. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. p. 39–56. OOPSLA '07, Association for Computing Machinery, New York, NY, USA (2007). <https://doi.org/10.1145/1297027.1297031>
- [131] Teixeira, G., Bispo, J.a., Correia, F.F.: Multi-language static code analysis on the lara framework. In: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. p. 31–36. SOAP 2021, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3460946.3464317>
- [132] Toman, J., Grossman, D.: Concerto: A framework for combined concrete and abstract interpretation. In: Proc. ACM Program. Lang. vol. 3. Association for Computing Machinery, New York, NY, USA (jan 2019). <https://doi.org/10.1145/3290356>
- [133] Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15–21, 2009. pp. 87–97. ACM (2009). <https://doi.org/10.1145/1542476.1542486>
- [134] Urban, C.: What programs want: Automatic inference of input data specifications. CoRR **abs/2007.10688** (2020), <https://arxiv.org/abs/2007.10688>

-
- [135] Urban, C., Miné, A.: A review of formal methods applied to machine learning. ArXiv **abs/2104.02466** (2021). <https://doi.org/10.48550/arXiv.2104.02466>
- [136] Urban, C., Müller, P.: An abstract interpretation framework for input data usage. In: Ahmed, A. (ed.) *Programming Languages and Systems*. pp. 683–710. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_24
- [137] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: A java bytecode optimization framework. In: *CASCON First Decade High Impact Papers*. p. 214–224. CASCON '10, IBM Corp., USA (2010). <https://doi.org/10.1145/1925805.1925818>
- [138] Veanes, M.: Applications of symbolic finite automata. In: Konstantinidis, S. (ed.) *Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings. Lecture Notes in Computer Science*, vol. 7982, pp. 16–23. Springer (2013). https://doi.org/10.1007/978-3-642-39274-0_3
- [139] Vongsingthong, S., Smachat, S.: Internet of things: a review of applications and technologies. *Suranaree Journal of Science and Technology* **21**(4), 359–374 (2014)
- [140] Wang, H., Chen, S., Yu, F., Jiang, J.R.: A symbolic model checking approach to the analysis of string and length constraints. In: Huchard, M., Kästner, C., Fraser, G. (eds.) *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. pp. 623–633. ACM (2018). <https://doi.org/10.1145/3238147.3238189>
- [141] Wang, S., Zhang, C., Su, Z.: Detecting nondeterministic payment bugs in ethereum smart contracts. *Proc. ACM Program. Lang.* **3**(OOPSLA) (oct 2019). <https://doi.org/10.1145/3360615>
- [142] Wei, F., Lin, X., Ou, X., Chen, T., Zhang, X.: Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. p. 1137–1150. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3243734.3243835>
- [143] Yamashita, K., Nomura, Y., Zhou, E., Pi, B., Jun, S.: Potential risks of hyperledger fabric smart contracts. In: *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. pp. 1–10 (2019). <https://doi.org/10.1109/IWBOSE.2019.8666486>
-

- [144] Yang, K., Huang, B., Stoyanovich, J., Schelter, S.: Fairness-aware instrumentation of preprocessing pipelines for machine learning. In: Workshop on Human-In-the-Loop Data Analytics (HILDA'20) (2020). <https://doi.org/10.1145/3398730.3399194>
- [145] Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. *Formal Methods Syst. Des.* **44**(1), 44–70 (2014). <https://doi.org/10.1007/s10703-013-0189-1>

Estratto per riassunto della tesi di dottorato

L'estratto (max. 1000 battute) deve essere redatto sia in lingua italiana che in lingua inglese e nella lingua straniera eventualmente indicata dal Collegio dei docenti.

L'estratto va firmato e rilegato come ultimo foglio della tesi.

Studente: Luca Negrini

matricola: 956516

Dottorato: Informatica

Ciclo: 35

Titolo della tesi¹ : A generic framework for multilanguage analysis – Design of an abstract interpretation-based static analyzer

Abstract: Le tecniche moderne per lo sviluppo software sono ampiamente influenzate dalle applicazioni distribuite. Dalle reti IoT alle infrastrutture client-server, il codice che compone l'applicazione è sempre più suddiviso in sottoprogrammi separati che interagiscono tra loro. Poiché questi ultimi sono completamente indipendenti l'uno dall'altro, ciascuno di questi programmi può essere sviluppato in un linguaggio di programmazione diverso, scegliendo la soluzione migliore per l'attività da svolgere.

Dal punto di vista dell'analisi statica dei programmi, affrontare una combinazione di linguaggi è impegnativo. Storicamente, lo sviluppo di un analizzatore ha avuto come target un singolo linguaggio, o una famiglia di linguaggi simili, in modo da raffinare l'analisi sulle sue (o loro) caratteristiche. In questa situazione, l'analisi semantica di un'intera applicazione multilinguaggio può essere eseguita solo attraverso una combinazione di analizzatori, predisponendo un canale di comunicazione tra gli strumenti in modo da considerare le interazioni tra moduli scritti in linguaggi diversi. Poiché sono necessarie più analisi, che potrebbero dover essere iterate numerose volte per permettere lo scambio di informazioni, questa configurazione non consente all'analisi statica di avere un impatto apprezzabile e significativo in scenari reali.

Questa tesi definisce un framework generico in cui è possibile definire analisi statiche multilinguaggio utilizzando la teoria dell'interpretazione astratta. Il framework è stato implementato in LiSA (Library for Static Analysis), una libreria Java open-source che fornisce l'infrastruttura completa necessaria per lo sviluppo di analizzatori statici. LiSA è modulare, garantendo che tutti i componenti che prendono parte all'analisi siano facili da sviluppare e facilmente intercambiabili. LiSA garantisce inoltre che i componenti siano parametrici rispetto a tutte le funzionalità specifiche del linguaggio: semantica, modello di esecuzione e modello di memoria non sono codificati direttamente all'interno dei componenti stessi. Infatti, LiSA analizza CFGs in cui l'insieme dei possibili nodi non è prefissato: gli utenti della libreria possono definire istanze di nodi specifiche per ogni linguaggio, definendo una semantica personalizzata e consentendo comportamenti diversi per lo stesso costrutto a seconda del linguaggio di programmazione in cui è stato scritto.

Il framework è stato istanziato per analizzare smart contracts scritti in Go e notebook di data science scritti in Python. Poiché è noto che il non-determinismo è problematico per gli ecosistemi blockchain, GoLiSA (un analizzatore per Go basato su LiSA) applica analisi di information flow per rilevare quando i costrutti non deterministici possono influenzare lo stato della blockchain. Invece, PyLiSA (un analizzatore per Python basato su LiSA) fornisce un'astrazione per il software che si occupa di dataframe, costruendo un grafo che

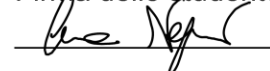
¹ Il titolo deve essere quello definitivo, uguale a quello che risulta stampato sulla copertina dell'elaborato consegnato.

traccia tutte le operazioni che il programma esegue su di essi, unificando tutti i costrutti sintattici che eseguono le stesse operazioni. Ulteriori astrazioni possono essere sviluppate basandosi su tale grafo per derivare proprietà sul programma originale, come data leakages o analisi della provenienza dei dati. Viene fornita anche una terza istanza dimostrativa, che evidenzia la capacità di LiSA di analizzare più linguaggi in un'unica analisi attraverso la scoperta di una vulnerabilità IoT che coinvolge codice C++ e Java.

Inoltre, questa tesi contiene due contributi aggiuntivi, in modo da fornire un ecosistema di analisi completo: SARL e Tarsis. SARL è un linguaggio che può essere utilizzato per modellare in modo compatto le modalità in cui framework e librerie interagiscono con l'applicazione analizzata. Attraverso SARL, è possibile produrre file di specifica concisi che un analizzatore può sfruttare per annotare automaticamente un programma, in modo tale che i componenti dell'analisi possano sfruttare la presenza (o l'assenza) di annotazioni per reagire in modo agnostico all'uso di un framework. Tarsis è invece un'astrazione per stringhe che sfrutta linguaggi regolari, modellati tramite automi a stati finiti che operano su un alfabeto di stringhe. L'uso di un tale alfabeto riduce le dimensioni gli automi, risultando in un notevole miglioramento

delle prestazioni rispetto ad astrazioni tradizionali basate su automi con alfabeti di singoli caratteri.

Firma dello studente





Università
Ca' Foscari
Venezia

DEPOSITO ELETTRONICO DELLA TESI DI DOTTORATO

DICHIARAZIONE SOSTITUTIVA DELL'ATTO DI NOTORIETA'

(Art. 47 D.P.R. 445 del 28/12/2000 e relative modifiche)

Io sottoscritto Luca Negrini

nat. o. a Isola della Scala (prov. VR) il 01/10/1993

residente a Nogara in Via Andrea Costa n. 26C

Matricola (se posseduta) 956516 Autore della tesi di dottorato dal titolo:
A generic framework for multilanguage analysis - Design of an abstract interpretation-
based static analyzer

Dottorato di ricerca in Informatica

(in cotutela con)

Ciclo 35

Anno di conseguimento del titolo 2023

DICHIARO

di essere a conoscenza:

- 1) del fatto che in caso di dichiarazioni mendaci, oltre alle sanzioni previste dal codice penale e dalle Leggi speciali per l'ipotesi di falsità in atti ed uso di atti falsi, decado fin dall'inizio e senza necessità di nessuna formalità dai benefici conseguenti al provvedimento emanato sulla base di tali dichiarazioni;
- 2) dell'obbligo per l'Università di provvedere, per via telematica, al deposito di legge delle tesi di dottorato presso le Biblioteche Nazionali Centrali di Roma e di Firenze al fine di assicurarne la conservazione e la consultabilità da parte di terzi;
- 3) che l'Università si riserva i diritti di riproduzione per scopi didattici, con citazione della fonte;
- 4) del fatto che il testo integrale della tesi di dottorato di cui alla presente dichiarazione viene archiviato e reso consultabile via internet attraverso l'Archivio Istituzionale ad Accesso Aperto dell'Università Ca' Foscari, oltre che attraverso i cataloghi delle Biblioteche Nazionali Centrali di Roma e Firenze;
- 5) del fatto che, ai sensi e per gli effetti di cui al D.Lgs. n. 196/2003, i dati personali raccolti saranno trattati, anche con strumenti informatici, esclusivamente nell'ambito del procedimento per il quale la presentazione viene resa;
- 6) del fatto che la copia della tesi in formato elettronico depositato nell'Archivio Istituzionale ad Accesso Aperto è del tutto corrispondente alla tesi in formato cartaceo, controfirmata dal tutor, consegnata presso la segreteria didattica del dipartimento di riferimento del corso di dottorato ai fini del deposito presso l'Archivio di Ateneo, e che di conseguenza va esclusa qualsiasi responsabilità dell'Ateneo stesso per quanto riguarda eventuali errori, imprecisioni o omissioni nei contenuti della tesi;
- 7) del fatto che la copia consegnata in formato cartaceo, controfirmata dal tutor, depositata nell'Archivio di Ateneo, è l'unica alla quale farà riferimento l'Università per rilasciare, a richiesta, la dichiarazione di conformità di eventuali copie.

Data 13/12/2022

Firma 

AUTORIZZO

- l'Università a riprodurre ai fini dell'immissione in rete e a comunicare al pubblico tramite servizio on line entro l'Archivio Istituzionale ad Accesso Aperto il testo integrale della tesi depositata;
- l'Università a consentire:
 - la riproduzione a fini personali e di ricerca, escludendo ogni utilizzo di carattere commerciale;
 - la citazione purché completa di tutti i dati bibliografici (nome e cognome dell'autore, titolo della tesi, relatore e correlatore, l'università, l'anno accademico e il numero delle pagine citate).

DICHIARO

- 1) che il contenuto e l'organizzazione della tesi è opera originale da me realizzata e non infrange in alcun modo il diritto d'autore né gli obblighi connessi alla salvaguardia di diritti morali od economici di altri autori o di altri aventi diritto, sia per testi, immagini, foto, tabelle, o altre parti di cui la tesi è composta, né compromette in alcun modo i diritti di terzi relativi alla sicurezza dei dati personali;
- 2) che la tesi di dottorato non è il risultato di attività rientranti nella normativa sulla proprietà industriale, non è stata prodotta nell'ambito di progetti finanziati da soggetti pubblici o privati con vincoli alla divulgazione dei risultati, non è oggetto di eventuale registrazione di tipo brevettuale o di tutela;
- 3) che pertanto l'Università è in ogni caso esente da responsabilità di qualsivoglia natura civile, amministrativa o penale e sarà tenuta indenne a qualsiasi richiesta o rivendicazione da parte di terzi.

A tal fine:

- dichiaro di aver autoarchiviato la copia integrale della tesi in formato elettronico nell'Archivio Istituzionale ad Accesso Aperto dell'Università Ca' Foscari;
- consegno la copia integrale della tesi in formato cartaceo presso la segreteria didattica del dipartimento di riferimento del corso di dottorato ai fini del deposito presso l'Archivio di Ateneo.

Data 13/12/2022

Firma 

La presente dichiarazione è sottoscritta dall'interessato in presenza del dipendente addetto, ovvero sottoscritta e inviata, unitamente a copia fotostatica non autenticata di un documento di identità del dichiarante, all'ufficio competente via fax, ovvero tramite un incaricato, oppure a mezzo posta

Firma del dipendente addetto

Ai sensi dell'art. 13 del D.Lgs. n. 196/03 si informa che il titolare del trattamento dei dati forniti è l'Università Ca' Foscari - Venezia.

I dati sono acquisiti e trattati esclusivamente per l'espletamento delle finalità istituzionali d'Ateneo; l'eventuale rifiuto di fornire i propri dati personali potrebbe comportare il mancato espletamento degli adempimenti necessari e delle procedure amministrative di gestione delle carriere studenti. Sono comunque riconosciuti i diritti di cui all'art. 7 D. Lgs. n. 196/03.

