Teaching Through Practice: Advanced Static Analysis with LiSA

Luca Negrini[1], Vincenzo Arceri[2(✉)], Luca Olivieri[1], Agostino Cortesi[1], and Pietro Ferrara[1]

[1] Ca' Foscari University of Venice, Venice, Italy
{luca.negrini,luca.olivieri,cortesi,pietro.ferrara}@unive.it
[2] University of Parma, Parma, Italy
vincenzo.arceri@unipr.it

Abstract. Nowadays, ready-to-use libraries and code generation are often used to streamline and speed up the software development process. The resulting programs are thus a collection of different modules that cooperate: proving their safety and reliability is increasingly complex, requiring sound formal techniques, such as static program analysis. However, while teaching static analysis to master's or PhD students, the predominant focus on theoretical concepts often leaves limited space for students to engage with the practical aspects of implementing static analyses and is limited to developing elementary ones. In this paper, we show how the infrastructure offered by LiSA can be exploited to learn how to implement advanced static analyses, such as string and relational numerical analyses, just focusing on their distinctive aspects. This would help to narrow the gap between theoretical and practical contents in static analysis courses, bringing the learning experience beyond the rudimentary implementation of static analyses to more sophisticated applications.

1 Introduction

Static analysis based on formal methods requires a non-trivial theoretical background and development skills. Traditional static analysis courses based on formal methods at the master's or doctoral level often give priority to mathematical-theoretical concepts, leaving limited room for students to actively engage with the practical aspects of implementing static analyses. Moreover, the design and implementation of new static analyses require building an infrastructure providing several basic building blocks, depending on the mathematical theory used. Therefore, developing even a toy static analyzer from scratch is a significant effort that could discourage students and teachers.

The goal of this paper is to show how we can exploit LiSA[10,15] to reduce practical and development efforts for the building of static analysis solutions based on abstract interpretation [6,7], allowing students to put their hands on more advanced static analyses based on formal methods, and providing ready-to-use components to develop their custom implementations. In particular, we choose to discuss and present two of the static analyses: the prefix abstract

E. Sekerinski and L. Ribeiro (Eds.): FMTea 2024, LNCS 14939, pp. 43–57, 2024.
https://doi.org/10.1007/978-3-031-71379-8_3

domain [5], and the pentagon abstract domain [12], showing how LiSA allows us to focus just on the implementation of the peculiar aspects of a static analysis of interest to come up with a complete and ready-to-run analysis. We then discuss how we use LiSA to teach practical static analysis in a Computer Science master course.

Paper Structure. Section 2 provides an informal and high-level introduction to static analysis by abstract interpretation. Section 3 describes the key components of LiSA. Section 4 reports the implementation details in LiSA of two advanced abstract domains, namely prefix abstract domain and pentagon abstract domain. Section 5 discusses how we integrated LiSA in the master course we teach. Finally, Sect. 6 concludes and illustrates how we intend to use LiSA to improve the teaching experience within static analysis courses.

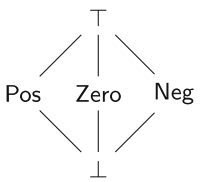
2 Static Analysis by Abstract Interpretation

Static analysis is a technique used for inspecting program properties without *concretely* executing the program. Examples of these properties may be whether the program terminates, which program variables are constants, or whether a program contains safety and security issues (e.g., buffer overflows [9], injection vulnerabilities [21], data leaks [11]).

For static analysis to *guarantee* the presence (or absence) of code properties, bugs, and vulnerabilities, one must adopt an approach based on a formal methods framework. Among these frameworks, a notable example is certainly abstract interpretation [6,7]. Abstract interpretation is a theoretical framework that provides a systematic way to correctly approximate program behaviors and reason about some properties of the program of interest on such approximation. One of the fundamental concepts of abstract interpretation is the notion of *abstract domain*, which provides an abstraction of the concrete program states as a set of *abstract values*. The goal of an abstract domain, typically modeled as a (complete) lattice [7], is to capture just the relevant aspects of program behavior while discarding details irrelevant to the analysis of interest. Regarding teaching, abstract interpretation principles require a non-trivial theoretical background, such as the notions of lattices, domains, fix-point theorems, and Galois connections [6]. Theoretical concepts thus tend to dominate the available time for the course, allowing only a shallow (and often not practical) exploration of simple abstractions.

The classic candidate to teach abstract domains is the *sign domain* [7], where numerical variables are abstracted to capture their sign, i.e., positive (Pos), negative (Neg), or zero (Zero). It is often chosen for its simplicity and intuitiveness. In particular, the sign abstract domain is depicted by its Hasse diagram reported in Fig. 1a, where the partial order between abstract values, the least upper bound, and the greatest lower bound are highlighted. For instance, Fig. 1b shows what the sign analysis infers on a simple code fragment.

At line 1, variables a and b are abstracted to the abstract value Pos because the concrete assigned values 5 and 7 are positive integer numbers. At line 3,



(a) Hasse diagram of signs.

```
1  a = 5, b = 7;
2  ⟅a ↦ Pos, b ↦ Pos⟆
3  x = a + b;
4  ⟅a ↦ Pos, b ↦ Pos, x ↦ Pos⟆
5  y = x * -2;
6  ⟅a ↦ Pos, b ↦ Pos, x ↦ Pos, y ↦ Neg⟆
7  z = y - 8;
8  ⟅a ↦ Pos, b ↦ Pos, x ↦ Pos, y ↦ Neg, z ↦ ⊤⟆
```

(b) The sign domain in action.

Fig. 1. The sign abstract domain.

in order to infer the correct sign of the variable `x`, it is necessary to define the so-called *abstract semantics* of the assignment and the sum operator. In general, once we have defined how our concrete values (e.g., integers) are abstracted into an abstract domain (e.g., sign), it is also necessary to define the abstract semantics of the operations in a program of interests, i.e., how each operator affects the abstract states represented by the abstract domain. Back to our example, the abstract semantics of the sign domain for the sum operator corresponds to the classical sign rules (e.g., $\mathsf{Pos} + \mathsf{Pos} = \mathsf{Pos}$). Hence, the analysis infers that the sign of `x` is Pos. At line 7, the abstract semantics of the minus operator between two Pos abstract values returns the top abstract value $\top$, and it assigns it to `z`, meaning that the analysis is not able to determine the sign for `z`, because we are reasoning on abstract values (e.g., signs) and not on the concrete ones (e.g., integers).

Non-relational and Relational Abstract Domains. The sign domain shown above can be seen as *"the Hello World of static analysis"*, being typically the first (and most common) numerical domain to be used to introduce static analysis by abstract interpretation, for instance, in master courses or PhD schools; this abstract domain lends itself well for this being a *non-relational abstract domain*, i.e., an abstract domain that does not explicitly model relationships between different variables, treating each variable independently. In contrast, *relational abstract domains*, such as pentagons [12], octagons [13,20] or convex polyhedra [3,8], also capture relationships between different variables.

Generally speaking, relational abstract domains offer higher precision than non-relational ones, but they are also more complex to define and require additional computational efforts to track and maintain the relationships between program variables.

3 LiSA

LiSA is a modular framework for developing static analyzers based on the abstract interpretation theory. LiSA was born as a tool for research purposes (e.g., [16–18]). However, its modular infrastructure also enabled us to use it to teach static analysis by abstract interpretation. The high-level analysis process

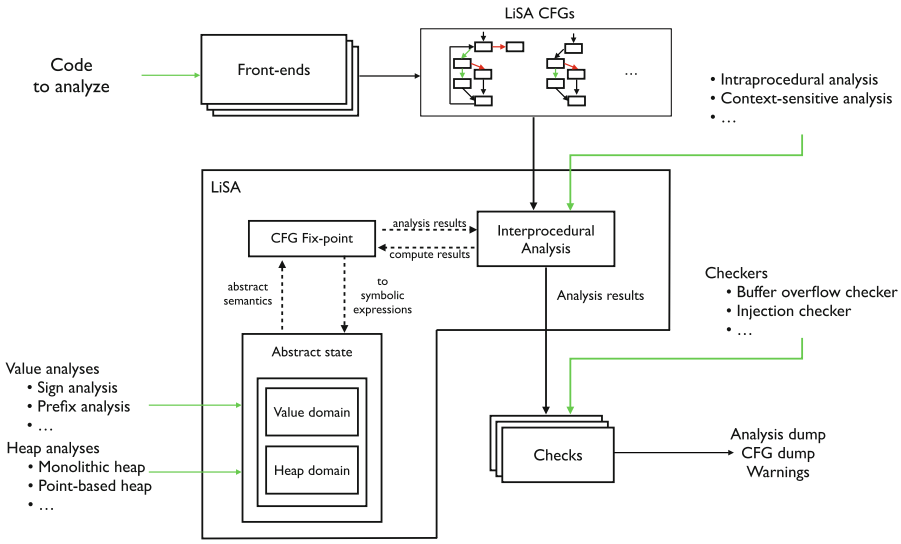


Fig. 2. LiSA overall execution.

of LiSA is reported in Fig. 2. LiSA analyzes control-flow graphs (CFGs) [1], a representation that expresses the control structure of the code using graph notation. In particular, LiSA uses a general design for CFGs, where statements do not have predefined semantics; instead, users of the framework can define custom statement instances implementing language-specific semantic functions, enabling the analysis of a wide range of programming languages and the development of multilanguage analyses. The analysis infrastructure is partitioned into three main areas: call evaluation, memory modeling, and value analysis. Each area corresponds to a configurable analysis component that operates agnostically concerning how the others are implemented. The analysis begins in the *Interprocedural Analysis*, which executes a program-wide fixpoint by computing each individual CFG's fixpoint. Whenever a call is encountered, the computation of its result is delegated back to the *Interprocedural Analysis*. Instead, non-calling statements are decomposed into a sequence of atomic operations, called *symbolic expressions*, each with precise semantics that the abstract domains can interpret. Memory-dealing expressions are handled by the *Heap Domain*, tracking their effect and rewriting them as abstract identifiers representing possible memory locations. Finally, the *Value Domain* tracks properties about variables (either program variables or abstract identifiers) and computes invariants for each program point. At the end of the analysis, results can be inspected through *Checks*, which are program visitors that can access the computed invariants and that can use them to warnings about points in the program where, for example, a property of interest holds. Code parsing logic and the definition of language-specific statements are provided by *Frontends*, which can also provide implementations for LiSA's components. These constitute effective static analyzers for individual

languages that can be combined to obtain multilanguage analyses. Several frontends have already been implemented, with new ones in the works. This paper and our courses use a frontend for Imp: a simple object-oriented language built for testing and demonstration. Imp is feature-rich enough to let the students practice with realistic static analysis without over-complicating the semantics abstraction. The Imp frontend is shipped with LiSA.[1] Students can use the various implementations of each analysis component provided within LiSA, thus focusing only on the one they are experimenting with.

Key Components for Implementing an Abstract Domain. In abstract interpretation courses, students usually become familiar with non-relational numeric domains, such as sign and interval [7]. We provide here the necessary notions not only to implement those domains but also other non-relational ones possibly dealing with non-numeric data, as well as relational one.

A (relational or non-relational) domain tracking values of variables must implement the `ValueDomain` interface. This requires providing all of the lattice operators of the domain (e.g., lub, partial order, widening) as well as *semantic transformers* to be invoked during fixpoints to track the semantics of each symbolic expression processed. `ValueDomain` implementers must provide, among other ones, (i) an `assign` method, invoked to store the result of an expression into a variable, and (ii) a `smallStepSemantics` method, invoked when the effect of a non-assigning symbolic expression is to be evaluated. Such methods transform the domain instance that receives the call into a new one according to the received expression(s) effects.

When coding a non-relational domain, some aspects of the implementation are independent of the domain itself. There is always a mapping from variables to instances of the domain's values, lattice operators are defined through functional lifting over such map, and the domain only evolves with assignments after recursively evaluating the right-hand side to a domain's value. To simplify such implementation tasks, LiSA ships with a `ValueDomain` implementation named `ValueEnvironment`, parametric to a `NonRelationalValueDomain` (`NRVD` for short). Such domain (i) is effectively a map from variables to instances of the `NRVD`, (ii) uses functional lifting for lattice operators, delegating to `NRVD` for the values of the mapping, (ii) has a `smallStepSemantics` implementation that is a no-op, (iii) has an `assign` implementation that evaluates the right-hand side to an instance of the `NRVD` and maps it to the target variable. Thus, an `NRVD` is mainly required to provide (i) lattice operator *for individual values* (e.g., between signs, instead of the whole mapping), and (ii) an evaluation logic for expressions. As the recursive visiting of a symbolic expression is independent of the `NRVD` of choice, LiSA also ships a subclass of it called `BaseNonRelationalValueDomain` (`BNRVD` for short) providing such logic. Implementers of `BNRVD` thus only have to provide (other than lattice operators) the evaluation logic of individual expressions given the value of their arguments.

[1] The Imp specification is available at https://lisa-analyzer.github.io/imp/.

4 Analyses Implementation with LiSA

4.1 An Example String Analysis: the Prefix Domain

String analysis focuses on tracking program properties concerning strings. In the context of string static analysis by abstract interpretation, several abstract domains have been proposed, each with different properties of interest and complexity: prefix, suffix, char inclusion, bricks, and string graph abstract domains [5], automata- and regex-based abstractions [4,14,22], and relational string analyses [2], are just a few examples. While LiSA implements most of the aforementioned abstract domains, in the rest of this paper, we show the teaching-related peculiar aspects of implementing the prefix abstract domain.[2] As the name suggests, the prefix abstract domain is a non-relational abstraction that keeps track of the prefix of string variables. Consider the following code fragment to give a flavor of how the prefix domain works.

```
1  if (x == 2) {
2    s = "javaSE";
3    ⟅s ↦ javaSE∗⟆
4  } else {
5    s = "javascript";
6    ⟅s ↦ javascript∗⟆
7  }
8  ⟅s ↦ java∗⟆
```

Supposing that the variable `x` has a statically unknown value, the value of the Boolean guard at line 1 is not determined, and to infer the prefix abstraction for the variable `s` at line 8 (i.e., at the end of `if`-statement), both branches must be taken into account by the analysis; the `true`-branch abstracts the value of `s` to $javaSE*$, that is the (concrete) value of `s` starts with the string"$javaSE$". Similarly, the `false`-branch abstracts the value of `s` to $javascript*$. At line 8, the analysis infers that the abstract value of `s` is $java*$ by applying the prefix's least upper bound between the two abstract values.

Prefix Abstract Domain Implementation. In the following, we report the implementation of the prefix abstract domain with LiSA.[3] In particular, thanks to LiSA, one only needs to implement the peculiar parts of the abstract domain to come up with a ready-to-use analysis. Concerning the prefix abstract domain, the key points are that: (i) it must be a *non-relational* domain, (ii) it needs a mechanism to keep track of abstract prefixes, and (iii) the lattice-related operators and abstract semantics must be implemented.

We can define the domain as a class called `Prefix`, implementing the `BNRVD` interface (Fig. 3). The class is characterized by the `prefix` field (Fig. 3 at line 3) to keep track of string prefix abstract values. The value of this field is set during the construction of an abstract value element (Fig. 3 at line 5).

[2] Full details about the definition and formalization of the prefix abstract domain can be found at [5].

[3] Full code available at https://github.com/lisa-analyzer/lisa/blob/master/lisa/lisa-analyses/src/main/java/it/unive/lisa/analysis/string/Prefix.java.

```
1  public class Prefix implements BaseNonRelationalValueDomain<Prefix>
2  {
3    private final String prefix;
4    // [...]
5    public Prefix(String prefix) { this.prefix = prefix; }
6    // [...]
7  }
```

Fig. 3. Implementation of the class `Prefix`.

Concerning lattice-based operations such as least upper bound and partial order operation, LiSA provides a base implementation for lattice abstract domains called `BaseLattice`, also implemented by `BNRVD`, that provides a base implementation for lattice operations refactoring behaviors that are common to most of the non-relational abstract domains. For instance, the least upper bound between the top (resp. bottom) element and any other abstract value, always returns top (resp. the other abstract value). Since `Prefix` implements `BNRVD` (and in turn `BaseLattice`), the least upper bound operator is implemented through `lubAux`, reported in Fig. 4, that can ignore cases where both `this` and `other` are the same abstract value, or are equal to bottom or top (being handled in `BaseLattice`). `lubAux` computes the longest common prefix between `this` and `other` (Fig. 4 at lines 3, where the definition of `longestCommonPrefix` is left implicit). If the common prefix is not empty, a new prefix abstract element is returned; otherwise, the top element is returned (Fig. 4 at line 4). Partial order and the greatest lower bound can be implemented similarly.

```
1  @Override
2  public Prefix lubAux(Prefix other) {
3    String result = longestCommonPrefix(this.prefix, other.prefix);
4    return result.isEmpty() ? TOP : new Prefix(result.toString());
5  }
```

Fig. 4. Least upper bound operator of `Prefix`.

Finally, `BNRVD` provides callbacks for evaluating the abstract semantics, one for each LiSA symbolic expression type. The following shows the abstract semantics for non-null constant values and binary string domain-specific expressions. To define the abstract semantics for string literals, we need to implement the `evalNonNullConstant` method, reported in Fig. 5, that returns a new prefix abstract value a value if the constant is a string, or the top element for the non-string constant values (e.g., integers).

Concerning binary string expressions, we instead implement the `evalBinaryExpression` method, reported in Fig. 6, that takes two abstract values `left` and `right` and the binary operator `op` that applies to them. Among the LiSA string symbolic expressions, the prefix abstract domain can infer a non-top prefix abstract value only for string concatenation (Fig. 6 at line 4). In contrast, for all the others, it returns top. Specifically, the prefix abstract value of `left` concatenated with `right`, returns `left`.

```
1  @Override
2  public Prefix evalNonNullConstant(Constant constant) {
3    if (constant.getValue() instanceof String) {
4      String str = (String) constant.getValue();
5      if (!str.isEmpty()) { return new Prefix(str); }
6    }
7    return TOP;
8  }
```

Fig. 5. Abstract semantic implementation of non-null constant values.

```
1  @Override
2  public Prefix evalBinaryExpression(BinaryOperator op, Prefix left,
3      Prefix right) {
4    if (op instanceof StringConcat) { return left; }
5    return TOP;
6  }
```

Fig. 6. Abstract semantic implementation of binary expressions.

4.2 An Example Relational Analyses: the Pentagon Domain

In Sect. 2, we highlighted that relational analyses are usually more complex abstractions to be designed and implemented. In terms of teaching, these may not be trivial for students first approaching static analysis. Here, we detail the implementation in LiSA of the Pentagon abstract domain [12]. This weakly relational numeric abstract domain is relatively simple compared to others but contains all the basic notions to understand relational domains. The Pentagon domain captures relations of the form $x \in [a, b] \wedge x < y$ and consists of two sub-domains: a non-relational interval abstraction ($x \in [a, b]$) combined with a relational *strict upper bound* domain ($x < y$). The two components are combined via (an abstraction of the) *reduced product* [7], corresponding to a Cartesian product where the two domains mutually exchange information in order to refine each other.

```
1  ⟅x ↦ [0, +∞], y ↦ [0, +∞]⟆
2  if (x > y) {
3      ⟅x ↦ [0, +∞], y ↦ [0, +∞], x > y⟆
4      r = x - y
5      ⟅x ↦ [1, +∞], y ↦ [0, +∞], r ↦ [0, +∞], x > y⟆
6      assert(r >= 0)
7      ⟅x ↦ [1, +∞], y ↦ [0, +∞], r ↦ [0, +∞], x > y⟆
8  }
```

Fig. 7. Code fragment example taken from [12, Sect. 6.2.1].

Let us consider the code fragment reported in Fig. 7, where variables `x` and `y` are initially abstracted to $[0, +\infty]$. The interval abstract domain is not precise enough to prove the assertion at line 6 (intuitively, the assignment of `r` at line 4 yields the interval $[-\infty, +\infty]$). Instead, the Pentagon abstract domain also tracks strict relations between variables, as highlighted by the invariant related to `x` and `y` at line 3. Thus, exploiting the information `x` $>$ `y`, we can refine `x` $-$

y, and in turn, the interval assigned to r, in $[0, +\infty]$;[4] this is enough to prove the assertion at line 6.

Pentagon Abstract Domain Implementation. Here, we report part of the implementation of the Pentagon abstract domain. The key points are that: (i) it must be a *relational* domain, (ii) it needs a mechanism to manage the information related to non-relational interval and strict upper bounds domains, (iii) the lattice-related operators and abstract semantics must be implemented.

```
1  public class Pentagon implements ValueDomain< Pentagon>,
2      BaseLattice<Pentagon> {
3    private final ValueEnvironment<Interval> intervals;
4    private final ValueEnvironment<UpperBounds> upperBounds;
5    // [...]
6  }
```

Fig. 8. Implementation of the class Pentagon.

Unlike the prefix domain implementation presented in Sect. 4.1, Pentagon is a relational domain and cannot exploit the BNRVD interface. The class Pentagon is reported in Fig. 8, implementing both the ValueDomain and BaseLattice interfaces. The domain is characterized by the field interval, keeping track of the interval of each variable, and the field upperBounds, keeping track of the relations of the form $x < y$ between two variables; the latter is implemented as an environment mapping each variable to a set of variables: for instance, $x \mapsto \{y, z\} \implies x < y \wedge x < z$. We will focus on how the two components of this domain can interact and exchange information, omitting implementation details of the components related to interval and upper bounds abstract domains.[5]

Being Pentagon a ValueDomain implementer, it must provide the implementation for the smallStepSemantics and assign methods. Concerning the former, the implementation is as simple as relying on the smallStepSemantics of the two sub-components of the domain, as reported in Fig. 9 at lines 1–4.

More attention is needed to implement the assign method, reported in Fig. 9 at lines 6–25, where refinement of the two sub-components of the Pentagon abstract domain may occur. The assign method takes an expression e that needs to be assigned to an identifier id (line 6). Lines 7–8 perform the assignment on the interval and the upper bounds abstractions calling their respective assign methods. Next, we check if it is possible to refine the abstraction: here, we discuss the refinement concerning assignments of the form id = x − y. If the assigned

[4] Formal specification of the Pentagon's abstract semantics for subtraction can be found in [12].

[5] Implementation of the interval and upper bound abstract domains can be found at https://github.com/lisa-analyzer/lisa/blob/master/lisa/lisa-analyses/src/main/java/it/unive/lisa/analysis/numeric/Interval.java and https://github.com/lisa-analyzer/lisa/blob/master/lisa/lisa-analyses/src/main/java/it/unive/lisa/analysis/numeric/UpperBounds.java.

```
1  public Pentagon smallStepSemantics(ValueExpression expression) {
2    return new Pentagon(intervals.smallStepSemantics(expression),
3            upperBounds.smallStepSemantics(expression));
4  }
5
6  public Pentagon assign(Identifier id, ValueExpression e) {
7    ValueEnvironment<UpperBounds> newBounds = upperBounds.assign(id, e);
8    ValueEnvironment<Interval> newIntvs = intervals.assign(id, e);
9    // refinement
10   if (e instanceof BinaryExpression) {
11     BinaryExpression be = (BinaryExpression) e;
12     if (be.getOperator() instanceof SubtractionOperator
13         && be.getLeft() instanceof Identifier
14         && be.getRight() instanceof Identifier) {
15       // id = x - y
16       Identifier x = (Identifier) be.getLeft();
17       Identifier y = (Identifier) be.getRight();
18       if (newBounds.getState(y).contains(x))
19         newIntvs = newIntvs.putState(id, newIntvs.getState(id).glb(
20           new Interval(MathNumber.ONE, MathNumber.PLUS_INFINITY)));
21     }
22     // [...]
23   }
24   return new Pentagon(newIntvs, newBounds).closure();
25 }
```

Fig. 9. Semantic transformers of `Pentagon`.

expression is in the form $\texttt{x} - \texttt{y}$ (lines 12–14), it is possible to refine the interval abstraction for variable *id* if the upper bounds domain knows that $y < x$. This is checked at line 18, and if so, the interval for `id` is refined, restricting it to $[1, +\infty]$, applying the greatest lower bound operator (lines 19–20). Finally, the result is returned at line 33 after applying the transitive closure (left implicit) on the abstract value.

4.3 Run LiSA

Once the desired abstract domain is developed, it is ready to analyze programs with the following fragment.

```
1  Program p = IMPFrontend.processFile(filePath);
2  LiSAConfiguration conf = new DefaultConfiguration();
3  conf.workdir = "output";
4  conf.analysisGraphs = GraphType.DOT;
5  conf.abstractState = new SimpleAbstractState<>(
6    new MonolithicHeap(),
7    new Pentagon(),
8    new TypeEnvironment<>(new InferredTypes()));
9  new LiSA(conf).run(p);
```

Line 1 uses the Imp frontend to parse an Imp program, returning a LiSA `Program`. Lines 2–8 build a LiSA configuration, setting the working directory, how to dump the analysis results, and the desired static analysis; we choose to use the `SimpleAbstractState` class, requiring a heap domain (monolithic heap), a value domain (`Pentagons`) and a type domain (non-relational type environment), that come with LiSA. Finally, line 9 runs the analysis, producing

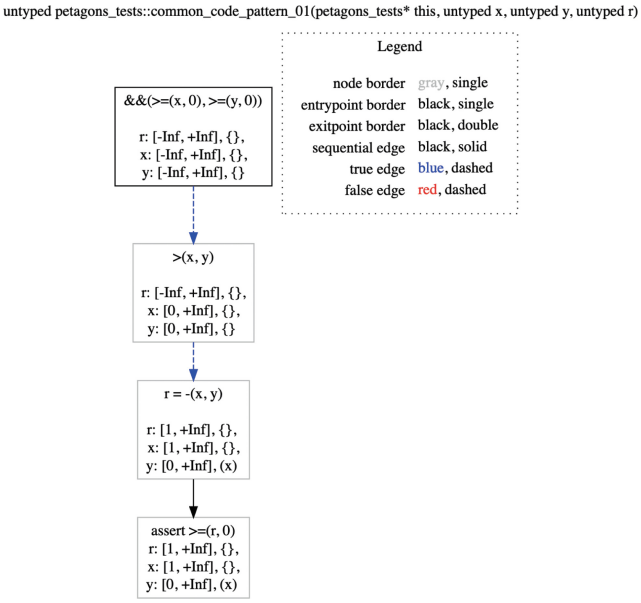


Fig. 10. LiSA dump for the program reported in Fig. 7.

the graph reported in Fig. 10 with the pentagon analysis results.[6] Note that to run the `Prefix` analysis instead, it is sufficient to change line 7 to `new ValueEnvironment<>(new Prefix())`.

5 Our Teaching Experience with LiSA

LiSA has been used for the last four years for the practical part of the Computer Science master course *Software Correctness, Security and Reliability* at University Ca' Foscari of Venice. Every year, the course features 30 to 50 students with a standard Computer Science background (in particular, set theory and object-oriented programming are given as prerequisites). Students learn lattice theory and abstract interpretation during the course, paired with practical experience using LiSA.

Course Structure. During the first three years, students took part in three lectures exclusively focused on LiSA: a first introductory lecture giving a full overview of the structure and the analysis process (Sect. 3), a second one with a focus on dataflow analyses, and a third one focusing on simple non-relational numeric analyses [10]. The second and third lectures also featured a live coding session, going through the relevant classes and interfaces and tackling the intricacies of the implementations. After each coding session, students were assigned

[6] For space limitations, we have omitted details about the heap and type analyses.

coding tasks covering the topics seen during the lecture, with a one-week deadline. Students also had the opportunity to develop an additional analysis, following a reference paper, as a final project for the course instead of taking the theoretical exam. The fourth year instead featured three more lectures on LiSA, each including live coding: one on advanced non-relational numeric analysis (typically the interval domain [7]), one on information flow analysis [19] and string analysis (Sect. 4.1), and a final one on relational numeric analyses (Sect. 4.2). Tasks were also given for these new lectures, and the theoretical exam was removed in favor of a mandatory project on the usage of LiSA to analyze real-world programs. The lectures were initially delivered through online classes (mandatory due to Covid-19 restrictions) but moved to in-person lectures during the last two years.

Settings for Coding. Coding sessions and task implementations aimed at targeting increasingly challenging scenarios without overwhelming the students with technical details or complex language features. Programs targeted were thus written in Imp to (i) exploit the Imp-frontend provided by LiSA instead of defining one from scratch, and (ii) have a simplified semantics to reason upon. Moreover, students used built-in components for the *Interprocedural Analysis* and the *Heap Domain*, allowing them to focus specifically on the peculiar aspects of an abstract domain while ignoring more complex and out-of-scope reasoning still needed to make the analysis run.

Tasks and Exams Evaluation. Tasks and final exams were graded from 0 to 10, assigning points for (i) the correct usage of LiSA as seen during classes, (ii) the correctness of the implementation w.r.t. the formal definition of the domain, and (iii) the correctness of the results produced over some simple yet expressive code snippets. Grades for the third point were assigned automatically, while the first two required manual inspection of the code submitted. Since the tasks were mandatory, all students actively engaged in solving them. Each year, marks gradually grew with each task, starting from an average of 6 out of 10 in the first task to a 7.5 in the second one. For the fourth year, having three more lectures proved beneficial: students reached an average of 8.8 marks on the fifth task. In the first three years, the number of students choosing the practical project varied between 10% to 40%, showing that most students were still hesitant to engage in a more challenging analysis development. This was the key motivation for the increase in practical lectures during the fourth year. At the time of writing, final projects for the fourth year are still in progress, so we do not have reports on the impact of such additional lectures on the students' understanding of static analysis.

Impact on Students' Careers. During all four years, we registered an increase in the number of students asking for a master's thesis on static analysis w.r.t. the number of requests in the previous editions of the course, with 2–3 students per year. Moreover, in the last three years, we disseminated LiSA through dedicated invited seminars inside other static analysis courses in two neighboring universities. Currently, the researchers and professors from such universities

are proposing bachelor and master theses projects related to LiSA, leading to additional eight students working with it, overall.

6 Conclusion

In this paper, we presented and described how LiSA can be exploited in a static analysis course to put the student's hands on two examples of advanced static analyses: a string analysis and a relational numeric analysis.

The adoption of LiSA into our educational curriculum has already yielded significant benefits in our previous courses on static analyses, allowing students to engage in the implementation of sophisticated static analyses. Furthermore, the ready-to-use components gave students a complete overview of all the essential components necessary to build static analyses via abstract interpretation while experimenting with the analyses they were tasked with. Additionally, LiSA also helps students develop a deeper understanding of the more practical applications of static analysis, especially during final course projects.

Given this success, noted by the positive feedback from the students, the next semester will see a heavier emphasis on using LiSA in our courses to bridge further the gap between theory and practice in static analysis education.

Acknowledgements. Work partially supported by Bando di Ateneo per la Ricerca 2022, funded by University of Parma, (MUR_DM737_2022_FIL_ PROGETTI_B_ARCERI_COFIN, CUP: D91B21005370003), SERICS (PE00000014, CUP H73C2200089001), and iNEST (ECS00000043 - CUP H43C22000540006) projects funded by PNRR NextGeneration EU.

References

1. 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
2. Arceri, V., Olliaro, M., Cortesi, A., Ferrara, P.: Relational string abstract domains. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 20–42. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_2
3. Becchi, A., Zaffanella, E.: PPlite: zero-overhead encoding of NNC polyhedra. Inf. Comput. **275**, 104620 (2020). https://doi.org/10.1016/J.IC.2020.104620
4. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_1
5. Costantini, G., Ferrara, P., Cortesi, A.: A suite of abstract domains for static analysis of string values. Softw. Pract. Exp. **45**(2), 245–287 (2015). https://doi.org/10.1002/SPE.2218
6. Cousot, P.: Principles of Abstract Interpretation. MIT Press (2021)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. 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
8. 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
9. Cowan, C., Wagle, F., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: attacks and defenses for the vulnerability of the decade. In: Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00, vol. 2, vol. 2, pp. 119–129 (2000). https://doi.org/10.1109/DISCEX.2000.821514
10. Ferrara, P., Negrini, L., Arceri, V., Cortesi, A.: Static analysis for dummies: experiencing lisa. In: Do, L.N.Q., Urban, C. (eds.) SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021, pp. 1–6. ACM (2021). https://doi.org/10.1145/3460946.3464316
11. Ferrara, P., Olivieri, L., Spoto, F.: Static privacy analysis by flow reconstruction of tainted data. Int. J. Softw. Eng. Know. Eng. **31**(07), 973–1016 (2021). https://doi.org/10.1142/S0218194021500303
12. Logozzo, F., Fähndrich, M.: Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. Sci. Comput. Program. **75**(9), 796–807 (2010). https://doi.org/10.1016/J.SCICO.2009.04.004
13. Miné, A.: The octagon abstract domain. High. Order Symb. Comput. **19**(1), 31–100 (2006). https://doi.org/10.1007/S10990-006-8609-1
14. Negrini, L., Arceri, V., Ferrara, P., Cortesi, A.: Twinning automata and regular expressions for string static analysis. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) VMCAI 2021. LNCS, vol. 12597, pp. 267–290. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67067-2_13
15. Negrini, L., Ferrara, P., Arceri, V., Cortesi, A.: LiSA: a Generic Framework for Multilanguage Static Analysis. In: Arceri, V., Cortesi, A., Ferrara, P., Olliaro, M. (eds.) Challenges of Software Verification. Intelligent Systems Reference Library, vol. 238, pp. 19–42 Springer, Singapore (2023). https://doi.org/10.1007/978-981-19-9601-6_2
16. Negrini, L., Shabadi, G., Urban, C.: Static analysis of data transformations in Jupyter notebooks. In: Ferrara, P., Hadarean, L. (eds.) Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2023, Orlando, FL, USA, 17 June 2023, pp. 8–13. ACM (2023). https://doi.org/10.1145/3589250.3596145
17. Olivieri, L., Jensen, T.P., Negrini, L., Spoto, F.: MichelsonLiSA: a static analyzer for Tezos. In: IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events, PerCom Workshops 2023, Atlanta, GA, USA, 13-17 March 2023, pp. 80–85. IEEE (2023). https://doi.org/10.1109/PERCOMWORKSHOPS56833.2023.10150247
18. Olivieri, L., et al.: Information flow analysis for detecting non-determinism in blockchain. In: Ali, K., Salvaneschi, G. (eds.) 37th European Conference on Object-Oriented Programming, ECOOP 2023, 17-21 July 2023, Seattle, Washington, United States. LIPIcs, vol. 263, pp. 23:1–23:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ECOOP.2023.23
19. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. A. Commun. **21**(1), 5–19 (2006)

20. Schwarz, M., Seidl, H.: Octagons revisited - elegant proofs and simplified algorithms. In: Hermenegildo, M.V., Morales, J.F. (eds.) Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14284, pp. 485–507. Springer (2023). https://doi.org/10.1007/978-3-031-44245-2_21
21. Spoto, F., et al.: Static identification of injection attacks in Java. ACM Trans. Program. Lang. Syst. **41**(3) (2019). https://doi.org/10.1145/3332371
22. Veanes, M.: Applications of symbolic finite automata. In: Konstantinidis, S. (ed.) CIAA 2013. LNCS, vol. 7982, pp. 16–23. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39274-0_3