Static Analysis for Dummies: Experiencing LiSA

Pietro Ferrara
pietro.ferrara@unive.it
Ca' Foscari University of Venice
Italy

Luca Negrini
luca.negrini@unive.it
Ca' Foscari University of Venice, Corvallis S.r.l.
Italy

Vincenzo Arceri
vincenzo.arceri@unive.it
Ca' Foscari University of Venice
Italy

Agostino Cortesi
cortesi@unive.it
Ca' Foscari University of Venice
Italy

Abstract

Semantics-based static analysis requires a significant theoretical background before being able to design and implement a new analysis. Unfortunately, the development of even a toy static analyzer from scratch requires to implement an infrastructure (parser, control flow graphs representation, fixpoint algorithms, etc.) that is too demanding for bachelor and master students in computer science. This approach difficulty can condition the acquisition of skills on software verification which are of major importance for the design of secure systems. In this paper, we show how LiSA (Library for Static Analysis) can play a role in that respect. LiSA implements the basic infrastructure that allows a non-expert user to develop even simple analyses (e.g., dataflow and numerical non-relational domains) focusing only on the design of the appropriate representation of the property of interest and of the sound approximation of the program statements.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Theory of computation** → *Program analysis.*

Keywords: Static analysis, Abstract interpretation, Dataflow analysis

ACM Reference Format:
Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2023. Static Analysis for Dummies: Experiencing LiSA. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Static program analyses provide information about behavioral properties of target programs at compilation time, i.e., working on their source code. Various frameworks are in use, such as abstract interpretation [3], model checking [1] and symbolic execution [5], only to cite a few. Several static analysis techniques are often taught in bachelor or master courses, in response of the increasing need of 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. For instance, let us consider static analysis by abstract interpretation. It requires notions about lattice and domain theory, control-flow graphs, fix-point algorithms, and Galois connections. Based on our teaching experience[1], the theoretical background and notions take most of the available time, allowing to show just some popular abstractions (e.g., sign and interval domains). The 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 to be neglected. This also limits the involvement of brilliant students in this research area.

In this paper, we present how LiSA (Library for Static Analysis[2]) can provide a complete and easy-to-use infrastructure to develop simple static analyses focusing on peculiar aspects. LiSA is an open source platform developed in Java that provides: (i) a very minimal Java-like object-oriented dynamically-typed target programming language (called IMP[3]), (ii) an internal and extensible control-flow graph representation, (iii) a common analysis framework for the development of new abstract domains, (iv) a simple interface for the development of common analyses, such as non relational or dataflow analyses, and finally (v) a fix-point algorithm on control-flow graphs.

[1]Prof. 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.
[2]https://github.com/UniVE-SSV/lisa
[3]Documentation available at https://unive-ssv.github.io/lisa

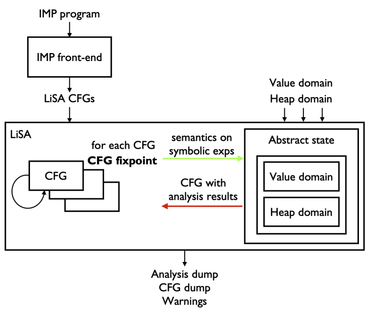

Figure 2. LiSA Overall Execution

Through the paper, we will adopt as running example the snippet of IMP code reported in Fig. 1, corresponding to a slight modification of Example 2.4 reported in [8].

```
def a = 1;
def x = a + b;
def y = a * b;
while (y > a) {
  a = a + 1;
  x = a + b;
}
return x;
```

Figure 1. The Running Example

The rest of the paper is organized as follows. Sect. 2 introduces LiSA and its main components, Sect. 3 presents some implementation of static analyses, and Sect. 4 concludes giving some pointers about the lessons learnt from the use of LiSA within a CS master course.

2 LiSA Architecture

In this section, we describe the LiSA architecture and its main components using IMP as reference language.

Fig. 2 depicts a high-level flow of execution of LiSA. In particular, given an IMP program, the IMP front-end, built upon LiSA, translates it into a set of control-flow graphs (CFG for short), one per method.

On each CFG, LiSA applies the typical worklist fix-point algorithm on CFG nodes. Each specific CFG node is rewritten as a composition of *symbolic expressions*, namely a set of expressions written in the internal language of LiSA. Symbolic expressions model the semantics of each CFG node. Informally speaking, we can say that the LiSA CFGs express the *syntax* of the program of interest, while LiSA symbolic expressions express *semantics* of CFGs, specifying the meaning of each CFG node. For this reason, as we will discuss later, LiSA abstract domains only deal with symbolic expressions and not with CFG nodes. At the end of the analysis, LiSA produces an entry and an exit abstract state for each node (aka, statement) in the CFGs. Then, different information can be dumped, for instance, the CFGs, the warning generated by the analyses, or the labelling of the CFGs with the analysis results assigned to each node.

2.1 LiSA CFG

LiSA CFG structure is designed to be as flexible as possible, adapting a structure for representing functions, methods and procedures coming from different programming languages.

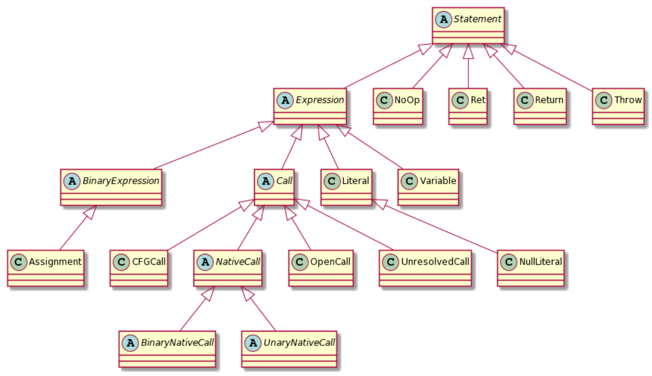

Figure 3. LiSA `Statement` Class Hierarchy

Fig. 3 depicts the Statement class hierarchy, corresponding to LiSA CFG nodes. The most part of the statements (e.g., Assignment, Variable, and Return) are self explanatory. As common in many programming languages, expressions are statements. Therefore, an expression like x (that is, accessing variable x) is a valid statement. The only non-standard part is the Call subtree. In particular, LiSA defines four types of calls: `UnresolvedCall`, a call towards another CFG, that is yet to be resolved to its actual target(s), `CFGCall`, a call towards one or more of the CFGs submitted as input to LiSA, `OpenCall`, a call towards a CFG that has not been submitted to LiSA, that therefore has no knowledge on (e.g., call to a library function), `NativeCall`, a call to a language native function that is simulated through a call (e.g., +, array access) that immediately provides its semantics through rewriting into a symbolic expression. However, call resolution and evaluation concern inter-procedural programs, a feature of static analysis that is often omitted in bachelor or master courses.

The class Edge links a source node to a target node and, as usual, we can have three types of edges: `SequentialEdge`, modelling a sequential/unconditional flow from the source node to the target node, `TrueEdge`, modelling a conditional flow from the source node to the target node, if the condition contained in the source node holds, `FalseEdge`, modelling a conditional flow from the source node to the target node, if the condition contained in the source node does not hold.

2.2 LiSA Front-end

We have described how programs are syntactically modelled in LiSA by means of the flexible CFG structure previously presented. In order to analyze IMP programs, LiSA needs an additional component, called front-end, that manages the translation from IMP source code to LiSA CFGs. In general, a LiSA front-end for a generic language L, translates L programs into LiSA CFGs. In this paper, we do not go into details about how a LiSA front-end works, since this concerns mostly how the syntax of a programming language is transformed into a structured representation. If on the one hand such component might still be interesting for courses

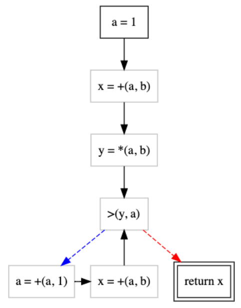

Figure 4. CFG of the Running Example

on compiler construction, on the other hand it is a side (but needed) component w.r.t. the core concepts of static analysis.

Fig. 4 reports the CFG of the running example presented in Sect. 1. In particular, the first three nodes contain the assignments at lines 1-3, while the rest of the CFG represents the while loop (lines 4-7) an the return statement (line 8).

2.3 LiSA Symbolic Expressions

We now deepen into the formal meaning of the content of CFG nodes. In particular, LiSA provides a set of internal *symbolic expressions*, with a well-defined semantics. Each Statement rewrites itself as a composition of these symbolic expressions. Symbolic expressions can be seen as the internal language of LiSA: indeed, abstract domains define and implement their semantics on such expressions, and not on instances of Statement.

Fig. 5 depicts the SymbolicExpression class hierarchy, where we can note two major types of symbolic expressions. ValueExpressions are the symbolic expressions defined on constant values and identifiers. Here, for example, we can find the NUMERIC_ADD and STR_CONCAT symbolic binary expressions, corresponding to the numerical addition and the string concatenation operations. Instead, HeapExpressions are the symbolic expressions that model operations on the heap, namely AccessChild, HeapAllocation and HeapReference. In this way, we can clearly split the expressions dealing with the heap, and the ones that instead concern only local identifiers. Consider for instance the expression `a+b` assigned to variable `x` at line 2 of our running example in Fig. 1. Such expression is translated into a symbolic `BinaryExpression` whose operator is a numerical addition, and this is represented by `+(a,b)` in the CFG in Fig. 4.[4]

2.4 Analysis Infrastructure

Before presenting the structure of the LiSA abstract state, we need to introduce two fundamental elements of the analysis infrastructure: Lattice and SemanticDomain interfaces.

Lattice represents elements of a lattice and it is parametric to the concrete instance L, that needs to implement the bottom and top elements, least upper bound (lub for short) and widening operations, and the partial order.

Similarly, SemanticDomain is parametric on the concrete instance D, and it represents domains that know how to reason about semantics of symbolic expressions. Specifically, a semantic domain reasons about symbolic expressions of type E and identifiers of type I, that are generic type parameters of the SemanticDomain interface. For instance, numerical abstract domains such as intervals or signs only reason about value symbolic expressions. A semantic domain D must implement the following methods:

- D assign(I id, E exp) yields a copy of the domain modified by the assignment of the abstract value corresponding to the evaluation of exp to id;
- D assume(E exp) yields a copy of a domain modified by assuming that exp holds;
- D forgetIdentifier(I id) forgets all information gathered by the domain about the identifier id;
- D forgetIdentifiers(Collection<I>ids) forgets all information about all identifiers ids;
- Satisfiability satisfies(E exp) yields whether exp is satisfied in the program state represented by a domain (it may return SAT, UNSAT, UNKNOWN or BOTTOM);
- D smallStepSemantics(E exp) yields a copy of a domain modified by the evaluation of the semantics of exp.

This separation permits, when implementing an abstract domain, to split the more algebraic domain information (e.g., lub, widening, partial order, ...) from the more semantic one (e.g., perform an assignment on the domain, satisfy a symbolic expression in the domain, ...).

LiSA Abstract State. An abstract state wraps together a value domain V and a heap domain H. In particular, ValueDomain is an interface, parametric on the concrete type V, that extends Lattice<V> and SemanticDomain<V, ValueExpression, Identifier>, meaning that can handle any ValueExpression and any Identifier. HeapDomain is an interface, parametric on the concrete type H, that extends Lattice<H> and SemanticDomain<SymbolicExpression, Identifier> meaning that it can handle any SymbolicExpression and any Identifier.

The structure of the abstract state is then implemented in LiSA following the framework presented in [4]. In a nutshell, when the abstract state needs to evaluate a (symbolic) expression, it is first evaluated on the heap abstract domain that, other than updating itself accordingly to the semantics, rewrites the expression by removing all the bits regarding heap operations with HeapIdentifiers, in order to make the expression processable by ValueDomains. Then, the rewritten expression, that does not contain any heap reference anymore, is processed and evaluated by the value abstract domain. All this is implemented in the AbstractState class. In this way, one can implement a value (e.g., numerical) domain without the need to implement the semantics of heap accesses as well. This is particularly convenient since

[4]Such a transformation is not known as a priori but it depends on the input abstract state (e.g., on the types of the expressions).

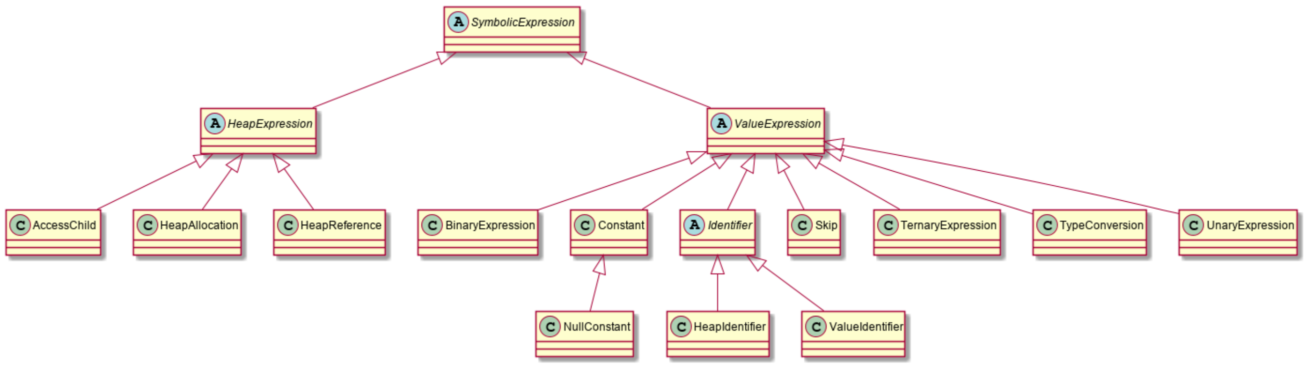

Figure 5. LiSA SymbolicExpression Hierarchy Class

students might apply standard implementation of heap abstractions in conjunction with their code without the need of knowing the inner details of such abstractions that are usually rather complex for non-experts.

LiSA Built-in Abstract Domains. LiSA provides several built-in abstract domains that ease the development of new abstract domains. Here, we discuss and present some value basic abstract domains, preparing the ground for Sect. 3.

LiSA provides a base implementation for lattice domains, called BaseLattice, that handles base cases (i.e., with top or bottom values) for lattice operations such as lub, widening and less or equal, as well as special instances for some popular lattice structures. One such structure is SetLattice<E>, modelling a lattice where elements are sets of instances of E, and where the lub is the set union. Similarly, LiSA provides the InverseSetLattice<E> class, where the lub is the set intersection.

Let us focus on value abstraction and let us consider non-relational numerical abstract domains, that is, domains that track some information (such as the interval or sign) on a single variable without relating it to other variables. For these kinds of abstract domains, LiSA provides two important built-in implementations: ValueEnvironment and BaseNonRelationalValueDomain. In particular, ValueEnvironment provides an environment for a non-relational value domain D, mapping identifiers to instances of D. In particular, a value environment keeps track of the non-relational abstract domain D and a map from identifier to elements of D, as reported below.

```
1   class ValueEnvironment<D extends NonRelationalValueDomain>
2     extends BaseLattice
3     implements SemanticDomain<Identifier, ValueExpression> {
4
5     Map<Identifier, D> function;
6     D dom;
7
8     public ValueEnvironment<D> assign(Identifier id, E exp) {
9       Map<Identifier, D> func = mkNewFunction(function);
10      D eval = dom.eval(value, this);
11      func.put(id, eval);
12      return new ValueEnvironment(dom, function);
13  }
```

Let us focus on the assign method. Its implementation is the one expected from an environment: given an assignment id = exp, evaluates exp in the current environment using the non-relational abstract domain D (lines 10-11), finally assigning the result of the evaluation to id (line 12).

BaseNonRelationalValueDomain is an abstract class that provides a base implementation for non-relational value domains. Instances of this domain are, for example, intervals, signs, integer constant propagation, and parity abstract domains. Such a base implementation internally handles the intermediate computation of the value symbolic expressions semantics. When presenting ValueEnvironment, we supposed that a non-relational value domain D provided an eval method, evaluating a value symbolic expression. A fragment of the implementation, in BaseNonRelationalValueDomain, is reported in the following.

```
1   public D eval(ValueExpression exp,  ValueEnvironment<D> env>) {
2    ...
3     if (exp instanceof BinaryExpression) {
4       BinaryExpression binary = (BinaryExpression) exp;
5
6       D left = eval(binary.getLeft(), env);
7       if (left.isBottom()) return left;
8
9       D right = eval(binary.getRight(), env);
10      if (right.isBottom()) return right;
11
12      return evalBinaryExpression(
13        binary.getOperator(), left, right);
14    }
15  }
16
17  abstract T evalBinaryExpression(BinaryOp op, D left, D right);
```

We have reported the behavior of eval only when exp is a binary expression. The others cases (e.g., ternary expressions, constant, ...) are similar. In particular, the left and right operands of the binary expression are recursively evaluated (lines 6-7 and 9-10, respectively) and when one of them corresponds to bottom, then bottom is returned. This class requires the concrete implementation to provide a method evalBinaryExpression, taking as input the binary operator and the final computed values for left and right. In this way, the concrete implementation (e.g., sign) only needs to implement the semantics of each binary operator (e.g., NUMERIC_ADD) on final values, since intermediate recursive evaluations have been already handled by BaseNonRelationalValueDomain and it can suppose that these final

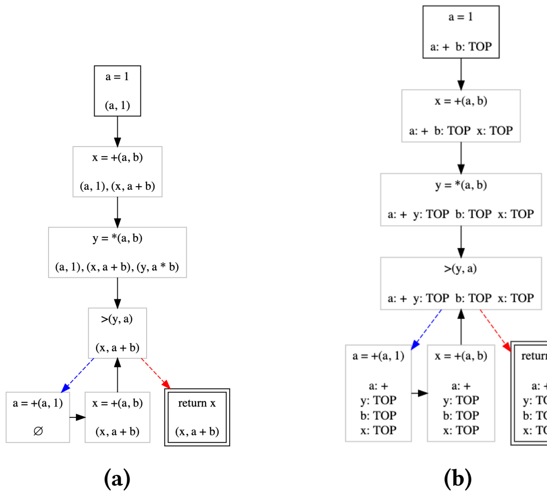

Figure 6. (a) Available Expressions (b) Sign Analysis Results on the Running Example in Fig. 1.

values are not equal to bottom (being excluded by the eval method).

As we will see in the next section, this analysis infrastructure permits to develop popular abstract domains with very few lines of codes.

3 Simple Analyses Implementation

Finally, we present the implementation of some simple value analyses that are usually formalized and taught in static analysis courses. In particular, we present the dataflow-based available expression 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 be implemented in a similar way.

How to Run LiSA. The following fragment shows how to run LiSA using the IMP front-end and how to obtain the analysis results that we will show in this section.

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

First line invokes the IMP front-end in order to process the IMP program located at filePath, returning a Program that contains the LiSA CFGs corresponding to the functions contained in the source program. Lines 2-3 create an instance of LiSA and sets the program that we aim to analyze. Lines 4-6 sets the abstract state, and in turn the heap and value domains. In this case, we use the default implementation for the abstract state but we set the monolithic heap[5] and the sign abstractions for heap and value domains, respectively.

[5]MonolithicHeap is a LiSA built-in heap domain where any heap concrete location is abstracted into a single and unique abstract location monolith.

Then, line 7 tells to LiSA to just dump the analysis results, that will be dumped in the output directory outDir specified at line 8. Finally, the analysis is executed.

3.1 Dataflow Analyses

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 dataflow analyses.

The DefiniteForwardDataflowDomain class extends ValueDomain, already discussed before, and InverseSetLattice, since it needs to track sets of dataflow elements ot type E (e.g., available expressions) on which the join operation is the set intersection, being the analysis definite.

```
1  class DefiniteForwardDataflowDomain<E extend DataflowElement>
2    extends InverseSetLattice<DefiniteForwardDataflowDomain<E>>
3    implements ValueDomain<DefiniteForwardDataflowDomain<E>> {
4
5      private final E domain;
6      ...
7      @Override
8      public DefiniteForwardDataflowDomain<E>
9        assign(Identifier id, ValueExpression exp) {
10
11       DefiniteForwardDataflowDomain<E> killed =
12         forgetIdentifiers(domain.kill(id, exp));
13       Set<E> res = new HashSet<>(killed.elements);
14       for (E generated : domain.gen(id, exp))
15         res.add(generated);
16       return new DefiniteForwardDataflowDomain<E>(domain, res);
17   }
18 }
```

Let us focus on the method assign. The concrete implementation of DataflowElement must provide the classical kill and gen methods of the dataflow analyses, that are used inside the assign method: it first applies the kill method in order to remove the killed dataflow elements (lines 11-12), and then it applies the gen method on the so-obtained result to add the dataflow elements that are generated by the assignment (lines 13-15). 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 few lines of code, as reported below.

```
class AvailableExps implements DataflowElement {
    private Identifier id;
    private SymbolicExpression exp;
    ...
    public Collection<AvailableExps> gen(
    Identifier id, ValueExpression exp,
    DefiniteForwardDataflowDomain<AvailableExps> domain) {
      Collection<AvailableExprs> result = new HashSet<>();
      if (!containsId(exp, id))
        result.add(new AvailableExps(id, exp));
      return result;
    }

    public Collection<Identifier> kill(
    Identifier id, ValueExpression exp,
    DefiniteForwardDataflowDomain<AvailableExps> domain) {
      Collection<Identifier> result = new HashSet<>();
      result.add(id);

      for (AvailableExps ae : domain.getDataflowElements()) {
        Collection<Identifier> ids = getIds(ae.exp);
        if (ids.contains(id)) result.add(ae.id);
    }
```

```
      return result;
    }
  }
```

At this point, it is enough to feed DefiniteForwardDataflowDomain with AvailableExps, set it as value domain and run LiSA as we have shown at the beginning of Sect. 3. Fig. 6a depicts the analysis results for our running example, where each node reports the computed available expressions.

3.2 Non-relational Abstract Domains

One of the first numerical analyses that are usually introduced in static analysis courses is sign analysis, tracking, for each variable whether it is zero, positive or negative. As we have already discussed in Sect. 2.4, LiSA offers the BaseNonRelationalValueDomain to easily develop non-relational value domains, requiring to the concrete class to just implement symbolic expressions evaluation methods on non-bottom elements. In the following, we report the Sign class with its evalBinaryExpression method. Similarly, it implements the evaluation methods for the other symbolic expressions types (e.g., unary, ternary, constant, . . .).

```
1  class Sign extends BaseNonRelationalValueDomain {
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,
9      Sign left, Sign right) {
10     switch (op) {
11       ...
12       case NUMERIC_ADD:
13         if (left.isZero()) return right;
14         else if (right.isZero()) return left;
15         else if (left.equals(right)) return left;
16         else return TOP;
17     ...
18     }
19   }
20 }
```

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 our running example are reported in Fig. 6b. The only variable for which its sign can determined is a (being other variables statically unknown inputs) and LiSA correctly infers that a is always positive.

3.3 Advanced Analyses

We conclude this section listing some built-in constructs offered by LiSA for more advanced analyses. For instance, LiSA offers constructs for easily combining abstract domains implemented in LiSA by means of the Cartesian product operator, permitting to run two independent analyses in parallel, without re-implementing the composition.

Moreover, we just focused on value analyses, omitting everything concerning heap analyses. Nevertheless, LiSA also offers built-in heap analyses such as class-based and point-based heap abstractions that can be combined with value analyses to show the interaction between value and heap domains.

Finally, in this paper we have just shown the implementations of dataflow and non-relational numerical analyses. With the same effort, it is possible to develop also string abstractions (e.g., [2]) and relational abstractions (e.g., [6, 7])

4 Conclusion

In this paper, we described how LiSA can be applied to teach static analysis, presenting its analysis infrastructure and some examples of static analyses that can be presented in a static analysis course for bachelor or master students. LiSA is a work-in-progress project and we plan to provide the first stable release by mid-2021, containing support for backward analysis, trace partitioning, new fix-point algorithm parameters (e.g., loop unrolling, narrowing) and new domain combinations (e.g., smashed sum, reduced product). For this reason, any suggestion and feedback from the reviewers on LiSA would be very useful and appreciated. Nevertheless, we have already involved LiSA in the static analysis master course in Ca' Foscari University of Venice. The lectures concerning LiSA are listed here: *Introduction to LiSA* (https://youtu.be/_PRzMxFhTU0), *Dataflow analysis* (https://youtu.be/FmYz9yb4_Vo), *Abstract interpretation* (https://youtu.be/3iNkQrbi9Ig).

References

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. URL https://doi.org/10.1145/5397.5399.

[2] G. Costantini, P. Ferrara, and A. Cortesi. A suite of abstract domains for static analysis of string values. *Softw. Pract. Exp.*, 45(2):245–287, 2015. URL https://doi.org/10.1002/spe.2218.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the Fourth ACM Symposium on Principles of Programming, January 1977*, pages 238–252. ACM, 1977. URL https://doi.org/10.1145/512950.512973.

[4] P. Ferrara. A generic framework for heap and value analyses of object-oriented programming languages. *Theor. Comput. Sci.*, 631:43–72, 2016. URL https://doi.org/10.1016/j.tcs.2016.04.001.

[5] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. URL https://doi.org/10.1145/360248.360252.

[6] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.*, 75(9):796–807, 2010. URL https://doi.org/10.1016/j.scico.2009.04.004.

[7] A. Miné. The octagon abstract domain. *High. Order Symb. Comput.*, 19 (1):31–100, 2006. URL https://doi.org/10.1007/s10990-006-8609-1.

[8] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999. ISBN 978-3-540-65410-0. URL https://doi.org/10.1007/978-3-662-03811-6.