Reduced product between Intervals and Parity

Sandro Baccega, Giulio Zausa
Ca' Foscari University of Venice
865024@stud.unive.it, 870040@stud.unive.it

Abstract Interpretation[1] is a way to analyze the behaviour of a program by approximating the values of the expressions, allowing to check some properties of a program. While different abstract domains can check some specific properties of the values, it's also possible to join them together using the Cartesian Product operator, using a Reduced Product to further refine the information carried by both domains. We implement the Reduced Product between Interval and Parity with LiSA[2] static analyzer, showing how it can be used to analyze IMP code.

I. Introduction

LiSA[2] is a Java library to build static analyzers, developed by UniVE-SSV. It contains a framework for analyzing CFG graphs using a fixpoint algorithm, with the possibility to implement analysis such as reaching definitions and type analysis. In our work we implemented the Reduced Product operation between interval and parity abstract domains, as described in [3]. We will use the IMP language to implement some tests and check how our implementation of the analysis can catch some bugs in some non–trivial examples.

A. *Cartesian Product between Abstract Domains*

Abstract Interpretation is a powerful analysis tool that allows to model the concrete values of a program execution into abstract values, given a certain mapping function, allowing an approximate yet sound analysis of the behaviour of a program. Abstract value domains can be for example sign (+,− and 0) and parity (even and odd), but also more advanced ones such as affine relationship between numbers [4] or string values [5].

A crucial part of the analysis is the combination of the abstract values using abstract operations. To be semantically sound abstract operation must preserve the same behaviour as its concrete model regarding the property used to abstract values. Some abstract operation, though, may cause a loss of precision in the analysis, where the output abstract value cannot be fully determined from the inputs. An example of this is the sum operation between positive and negative abstract values. In such cases we represent the output of this computation using the top ($\top$) lattice value.

When a single abstract domain fails to maintain enough precision to give useful results after the operations computations it's possible to use multiple abstract domains to model the program values, performing the cartesian product between all the abstract values of the models (eq. 1). The partial order is defined as the conjunction of the partial orders of the two domains (eq. 2), and the least upper bound and the greatest lower bound operators are defined as the component–wise application of the operators of the two domains. The operation semantics function is applied componentwise (eq. 3). This way the cartesian product between two abstract domains forms a lattice. Further formal details can be found in [3].

$$\mathbf{R} = \mathbf{A} \times \mathbf{B} \tag{1}$$

$$(a, b) \leq_{\mathbf{R}} (a', b') \quad \text{if } a \leq_{\mathbf{A}} a' \text{ and } \leq_{\mathbf{B}} b' \tag{2}$$

$$\sigma_{\mathbf{R}}[(a, b)] = (\sigma_{\mathbf{A}}[a], \sigma_{\mathbf{B}}[b]) \tag{3}$$

Unfortunately, as pointed out by Patrick Cousot [6], "the Cartesian product discovers in one shot the information found separately by the component analyses", but "we do not learn more by performing all analyses simultaneously than by performing them one after another and finally taking their conjunctions". This is because the cartesian product may result in some elements that represents the same information, such as for example if we make the product between sign and interval. If we consider the elements $([-1 \dots 1], +)$, $([0 \dots 1], +)$ and $([1 \dots 1], +)$, we know that the intersection between those three results in the singleton 1, hence some of them are not minimal.

B. Reduced Product between Abstract Domains

To have better analysis results when using the cartesian product with abstract domains we can use the Reduced Product[1] between the two, solving the non–minimals problem described earlier. The reduced product of two domains $\mathbf{A}$ and $\mathbf{B}$ is the subset of $\mathbf{A} \times \mathbf{B}$ whose elements are $\{\rho(a, b) : (a, b) \in \mathbf{A} \times \mathbf{B}\}$ where the reduction operator ρ is defined by $\rho(a, b) = \text{glb}\{(a', b') : \gamma_{\mathbf{R}}(a, b) \leq_C \gamma_{\mathbf{R}}(a', b')\}$. In practice, given the abstract states of the two domains (a, b) we try to find the smallest state (a', b') such that the $\gamma_A(a') \subseteq \gamma_{\mathbf{A}}(a)$ and $\gamma_{\mathbf{B}}(b') \subseteq \gamma_B(b)$, but the intersection of the two concretizations stays the same $(\gamma_{\mathbf{R}}(a, b) = \gamma_{\mathbf{R}}(a', b'))$.

Applying the Reduced Product between our two domains (Interval and Parity) we can use the information from a domain to refine the other, giving us better analysis results. As an example, we can consider the abstract value $([2, 4], odd)$. We can easily see how $\rho([2, 4], odd) = \rho([3, 4], odd) = \rho([3, 3], odd)$, resulting in the glb between the three $([3, 3], odd)$ that concretizes by $\gamma_{\mathbf{R}}$ to $\{3\}$.

Unfortunately, the reduction operator ρ is not computable in general, so its implementation has to be specific for the domains we are refining. To solve our problem, we need to define a specific one for refining Interval and Parity.

C. Granger's product

Granger's product[7] allows to compute an over-approximation of the reduction operator. It works by defining two operators $\rho_1 : \mathbf{R} \to \mathbf{A}$ and $\rho_2 : \mathbf{R} \to \mathbf{B}$, each one refining a domain with the info from the other one. The reduction is obtained by applying the two operators ρ_1 and ρ_2 until a fixpoint is reached. In order to preserve soundness ρ_1 and ρ_2 has to satisfy the conditions eq. 4 (i.e. each refine their own domain while keeping the same concretization).

$$\begin{aligned} \rho_1(a, b) \leq_{\mathbf{A}} a \wedge \gamma_{\mathbf{R}}(\rho_1(a, b), b) = \gamma_{\mathbf{R}}(a, b) \\ \rho_2(a, b) \leq_{\mathbf{B}} b \wedge \gamma_{\mathbf{R}}(a, \rho_2(a, b)) = \gamma_{\mathbf{R}}(a, b) \end{aligned} \tag{4}$$

For our domains (interval and parity) we define our ρ_1 and ρ_2 as such ($\mathcal{P} = \{\top, \bot, \text{odd}, \text{even}\}$):

$$\rho_1([a, b], \mathcal{P}) = \begin{cases} \bot & \text{if } a = b \text{ and } a \notin \mathcal{P} \\ [a + 1, b] & \text{if } (a + 1) \in \mathcal{P} \\ [a, b - 1] & \text{if } (b - 1) \in \mathcal{P} \\ [a, b] & \text{else} \end{cases} \tag{5}$$

$$\rho_2([a, b], \mathcal{P}) = \begin{cases} \bot & \text{if } a = b \text{ and } a \notin \mathcal{P} \\ \text{odd} & \text{if } a = b \text{ and } a \text{ is odd} \\ \text{even} & \text{if } a = b \text{ and } a \text{ is even} \\ \mathcal{P} & \text{else} \end{cases} \tag{6}$$

It's possible to prove that ρ_1 and ρ_2 are sound and satisfies eq. 4, since they result in the same concretization.

II. Implementation

To perform the Reduced Product with LiSA we implemented an abstract class called `ReducedCartesianProduct` that allows a generic combination between two `NonRelationalValueDomain` classes, allowing code reuse. By implementing ρ_1 and ρ_2 custom functions we can perform the Granger's product. To do so every time an expression is evaluated we call ρ_1 and ρ_2 multiple times, until a fixpoint is reached. This is achieved using a do–while loop, repeated until the result stays the same as the previous iteration. We evaluate the Granger's product also every time a *glb*, a *lub* or a widening is calculated, to allow evaluation of enviroments in code branches.

For our two domains (Interval and Parity) we inherited from the `ReducedCartesianProduct` abstract class, implementing ρ_1 and ρ_2 as described in eq. 5 and eq. 6 in our new class `IntervalParityDomain`. For the two abstract domains we were able to reuse the implementations already provided with LiSA (`Interval` and `Parity` classes), applying some improvements for our test cases, one of which ended up as contribution in the main repository[1].

- We changed the `Interval` class to deal with modulo operations, where previously $\top$ was always the result ($r = \max(|c|, |d|) - 1$):
  - $[0,0] \mod [a,b] = [0,0]$
  - $[0,0] \mod [a,b] = [0,0]$
  - $[a,a] \mod [b,b] = [a \mod b, a \mod b]$
  - $[a,b] \mod [c,d] = \begin{cases} [0,r] & \text{if } a > 0 \\ [-r,0] & \text{if } b \leq 0 \\ [-r,r] & \text{else} \end{cases}$
- We improved the `Parity` class to handle some uncovered cases in evaluation, assumption and satisfiability:
  - Multiplication with even values, as $(\top * \textit{even})$, always return an even number;
  - A number with parity $\mathcal{P}$ modulo an even number results[2]; in a number with parity $\mathcal{P}$
  - When assuming a value in a branch an expression of type $x \mod 2 = 0$ we can set its parity in the two branches as surely even and odd respectively, since mod 2 checks in fact the parity;
- We made some methods public to allow better code reuse (`isSingleton`, `isEven`, `isOdd`, `getFromInt`).

A. Additional refinement

When testing our implementation we realized how in some cases it was possible to check the satisfiability of expressions in branches when modulo 2 operation was performed. In fact when checking the result of an expression mod 2 we are simply reading the parity of a number, so the Parity abstract domain can be used to check the satisfiability of that expression. Unfortunately, though, this cannot be fully implemented into the Parity class, since it has to be able to read the Interval domain environment. For example when performing $a \mod b == c$ we may read the interval value of b and c, and if they are respectively singletons with 2 and 0 we can assume a parity in branches.

To do so we overrided the `satisfies` method in IntervalParityDomain, adding additional checks for modulo 2 operation, using the `eval` method to evaluate the interval and parity.

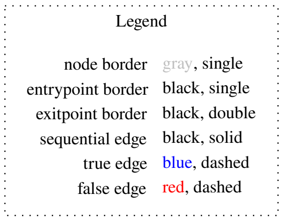
III. Test Cases

We tested our implementation with some non trivial test cases, showing how it is able to infer some properties about the program using both Interval and Parity. We implemented our tests using the IMP language, and they are available on the file `redprod.imp`[3]. Most of our test cases uses modulo operation as a mean to obtain numeric intervals.

[1]https://github.com/UniVE-SSV/lisa/pull/80

[2]$(a \mod b) \mod 2 = a \mod 2, \quad \text{if } b \mod 2 = 0$

[3]https://github.com/giulioz/lisa/blob/master/lisa/redprod.imp



Legend

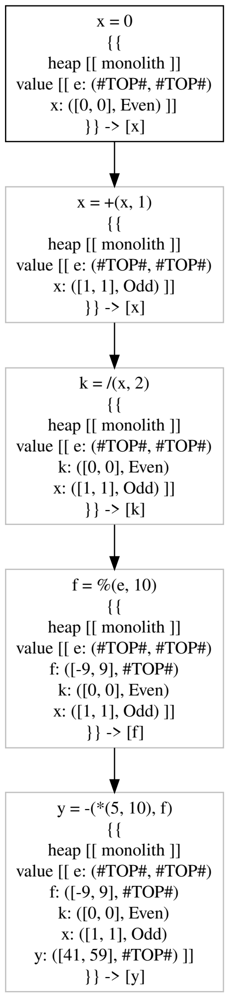
node border	gray, single
entrypoint border	black, single
exitpoint border	black, double
sequential edge	black, solid
true edge	blue, dashed
false edge	red, dashed

Figure 1: Notation used for the CFGs

We show how LiSA, using our `IntervalParityDomain` implementation, is analyzing the CFGs of the test functions. All ours test cases use the notation described in fig. 1.

A. Test 1: Arithmetic under Parity and Interval

In this test (fig. 2) we show that we can derive the interval and parity from various arithmetic operations, like additions, products, divisions and modulo. We can see how, when performing the modulo of a $\top$ value, we have a positive and negative interval as result, since in IMP a negative number stays negative when performing modulo.



```
x = 0
{{
heap [[ monolith ]]
value [[ e: (#TOP#, #TOP#)
x: ([0, 0], Even) ]]
}} -> [x]

x = +(x, 1)
{{
heap [[ monolith ]]
value [[ e: (#TOP#, #TOP#)
x: ([1, 1], Odd) ]]
}} -> [x]

k = /(x, 2)
{{
heap [[ monolith ]]
value [[ e: (#TOP#, #TOP#)
k: ([0, 0], Even)
x: ([1, 1], Odd) ]]
}} -> [k]

f = %(e, 10)
{{
heap [[ monolith ]]
value [[ e: (#TOP#, #TOP#)
f: ([-9, 9], #TOP#)
k: ([0, 0], Even)
x: ([1, 1], Odd) ]]
}} -> [f]

y = -(*(5, 10), f)
{{
heap [[ monolith ]]
value [[ e: (#TOP#, #TOP#)
f: ([-9, 9], #TOP#)
k: ([0, 0], Even)
x: ([1, 1], Odd)
y: ([41, 59], #TOP#) ]]
}} -> [y]
```

Figure 2: Basic arithmetic using modulo and the results obtained in the abstract domain

B. Test 2: Branches and Modulo

In this test (fig. 3) we can see how the branching operations are performed in the IntervalParityDomain. For example we can see that in the first branch (if($x \geq 0$)) the CFG splits and the value of x is assumed accordingly: $x \in [-\infty, -1]$ and $x \in [0, +\infty]$. Our branching also handles modulo 2 checks using both Parity and Interval domain(if($c\%mod = eq$), as described in section II-A, and we can see that the false branch is unreachable, since the parity of c is Even, $mod = 2$ and $eq = 0$. To obtain non-singleton intervals with non-top parity we use sum and multiplication of a modulo of a $\top$ value, after being restricted to $[0, +\infty]$ with the $x \geq 0$ check.

C. Test 3: For Loop

In this test (fig. 4) we can see how our domain handles a for loop, assuming the correct Interval and Parity after every branch. For example we can see how after $i < 5$ check the interval is partitioned into $[1, 4]$ (true branch) and $[5, 5]$ (false branch). In this case Granger's product comes into action inferring the Parity from $[5, 5]$ as odd, since it's a singleton. After that the $mod == 0$ check is able to correctly infer how it will be never 0, so the false branch is unreachable, inferring the final correct return value.

D. Test 4: Parity from Interval

In this test (fig. 5) we can see how our domain can reduce the set of valid results, using Parity to reduce Interval. After a couple of basic operations we can see how the $b = y\%2$

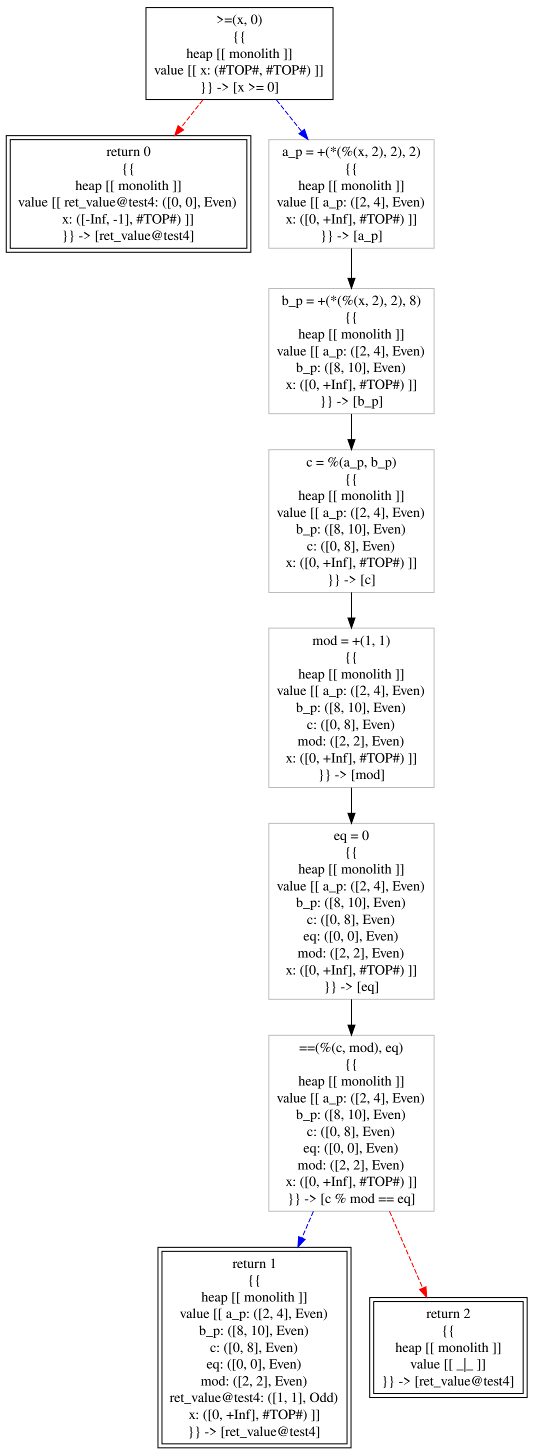

>=(x, 0) {{ heap [[monolith]] value [[x: (#TOP#, #TOP#)]] }} -> [x >= 0]
return 0 {{ heap [[monolith]] value [[ret_value@test4: ([0, 0], Even) x: ([-Inf, -1], #TOP#)]] }} -> [ret_value@test4]
a_p = +(*(%(x, 2), 2), 2) {{ heap [[monolith]] value [[a_p: ([2, 4], Even) x: ([0, +Inf], #TOP#)]] }} -> [a_p]
b_p = +(*(%(x, 2), 2), 8) {{ heap [[monolith]] value [[a_p: ([2, 4], Even) b_p: ([8, 10], Even) x: ([0, +Inf], #TOP#)]] }} -> [b_p]
c = %(a_p, b_p) {{ heap [[monolith]] value [[a_p: ([2, 4], Even) b_p: ([8, 10], Even) c: ([0, 8], Even) x: ([0, +Inf], #TOP#)]] }} -> [c]
mod = +(1, 1) {{ heap [[monolith]] value [[a_p: ([2, 4], Even) b_p: ([8, 10], Even) c: ([0, 8], Even) mod: ([2, 2], Even) x: ([0, +Inf], #TOP#)]] }} -> [mod]
eq = 0 {{ heap [[monolith]] value [[a_p: ([2, 4], Even) b_p: ([8, 10], Even) c: ([0, 8], Even) eq: ([0, 0], Even) mod: ([2, 2], Even) x: ([0, +Inf], #TOP#)]] }} -> [eq]
==(%(c, mod), eq) {{ heap [[monolith]] value [[a_p: ([2, 4], Even) b_p: ([8, 10], Even) c: ([0, 8], Even) eq: ([0, 0], Even) mod: ([2, 2], Even) x: ([0, +Inf], #TOP#)]] }} -> [c % mod == eq]
return 1 {{ heap [[monolith]] value [[a_p: ([2, 4], Even) b_p: ([8, 10], Even) c: ([0, 8], Even) eq: ([0, 0], Even) mod: ([2, 2], Even) ret_value@test4: ([1, 1], Odd) x: ([0, +Inf], #TOP#)]] }} -> [ret_value@test4]
return 2 {{ heap [[monolith]] value [[_|_]] }} -> [ret_value@test4]

Figure 3: Assumptions and satisfiability works for mod2 operation

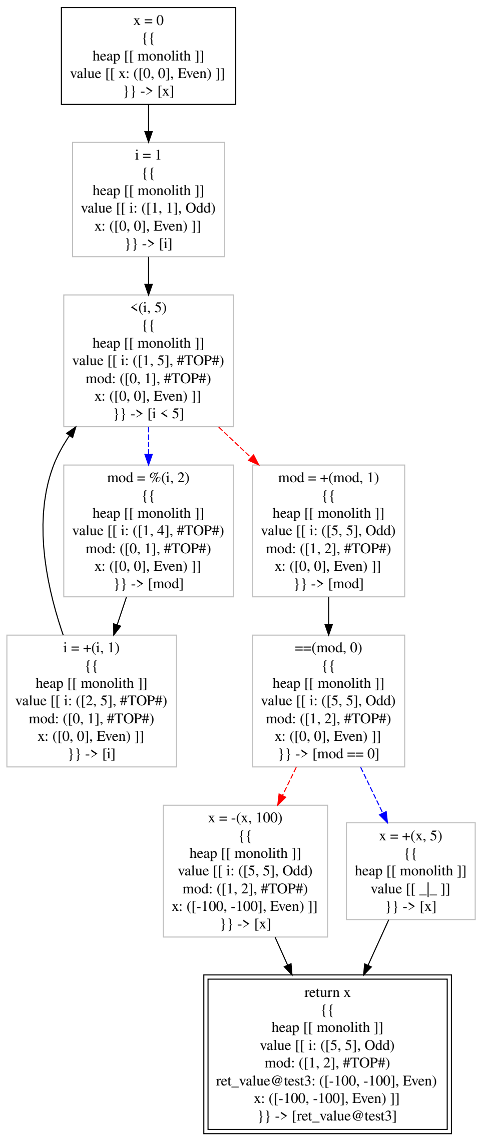

x = 0 {{ heap [[monolith]] value [[x: ([0, 0], Even)]] }} -> [x]
i = 1 {{ heap [[monolith]] value [[i: ([1, 1], Odd) x: ([0, 0], Even)]] }} -> [i]
<(i, 5) {{ heap [[monolith]] value [[i: ([1, 5], #TOP#) mod: ([0, 1], #TOP#) x: ([0, 0], Even)]] }} -> [i < 5]
mod = %(i, 2) {{ heap [[monolith]] value [[i: ([1, 4], #TOP#) mod: ([0, 1], #TOP#) x: ([0, 0], Even)]] }} -> [mod]
mod = +(mod, 1) {{ heap [[monolith]] value [[i: ([5, 5], Odd) mod: ([1, 2], #TOP#) x: ([0, 0], Even)]] }} -> [mod]
i = +(i, 1) {{ heap [[monolith]] value [[i: ([2, 5], #TOP#) mod: ([0, 1], #TOP#) x: ([0, 0], Even)]] }} -> [i]
==(mod, 0) {{ heap [[monolith]] value [[i: ([5, 5], Odd) mod: ([1, 2], #TOP#) x: ([0, 0], Even)]] }} -> [mod == 0]
x = -(x, 100) {{ heap [[monolith]] value [[i: ([5, 5], Odd) mod: ([1, 2], #TOP#) x: ([-100, -100], Even)]] }} -> [x]
x = +(x, 5) {{ heap [[monolith]] value [[_|_]] }} -> [x]
return x {{ heap [[monolith]] value [[i: ([5, 5], Odd) mod: ([1, 2], #TOP#) ret_value@test3: ([-100, -100], Even) x: ([-100, -100], Even)]] }} -> [ret_value@test3]

Figure 4: Assumptions for intervals and parity in a for loop

returns $[0, 0]$ even if normally it should return $[0, 1]$. This reduction is possible because a modulo operation between two even numbers can only return another even number, this results in $([0, 1], Even)$ that can be further reduced to $[0, 0]$ thanks to Granger's product.

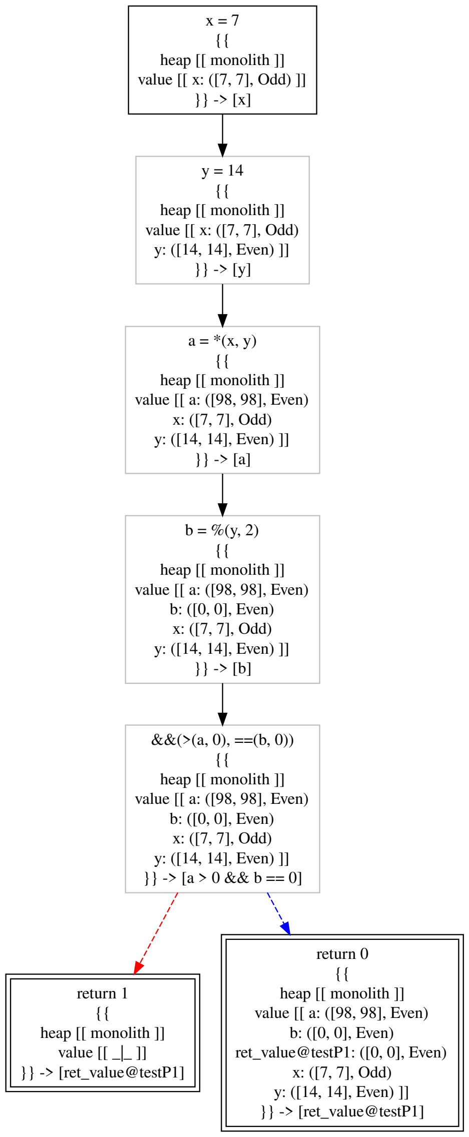


Figure 5: Inferring values in branches when mod 2 result is checked

IV. Conclusions

We have provided an implementation of the Reduced Product between Interval and Parity for the LiSA static analysis framework, showing how it is able to exploit the information from the two abstract domains to correctly prove more information about the execution than the two domains alone. We tested our implementation with some non-trivial test cases, showing how it is able to catch some specific bugs in the code.

References

[1] P. Cousot and R. Cousot. "Systematic design of program analysis frameworks". In: *POPL '79*. 1979.

[2] UniVE-SSV. *UniVE-SSV/lisa*. URL: https://github.com/UniVE-SSV/lisa.

[3] A. Cortesi, G. Costantini, and Pietro Ferrara. "A Survey on Product Operators in Abstract Interpretation". In: *Festschrift for Dave Schmidt*. 2013.

[4] M. Karr. "Affine relationships among variables of a program". In: *Acta Informatica* 6 (2004), pp. 133–151.

[5] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. "A suite of abstract domains for static analysis of string values". In: *Software: Practice and Experience* 45.2 (2015), pp. 245–287. DOI: https://doi.org/10.1002/spe.2218. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2218. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2218.

[6] P. Cousot. "Abstract interpretation". MIT course 16.399, http://web.mit.edu/16.399/www/. Feb. 2005.

[7] Philippe Granger. "Improving the Results of Static Analyses Programs by Local Decreasing Iteration". In: *FSTTCS*. 1992.