LiSA Extension — Pentagons: A weakly relational abstract domain for the efficient validation of array accesses

Andrea Munarin (879607), Simone Jovon (882028)

Year 2022/2023

1 Introduction

The objective of this task is to enhance the capabilities of LiSA by introducing the domain of pentagons. This specific domain holds significant potential for various types of analyses. As exemplified in the paper that inspired this project, one practical application involves verifying array accesses.

In particular, elements within the pentagon domain take the form of $x \in [a, b] \wedge x < y$, where x and y represent program variables, and a and b are rational values. These elements enable the representation of most variable bounds, especially those related to array indices. The intervals $[a, b]$ address the numerical aspects, such as checking for array underflows (e.g., ensuring that a is greater than or equal to 0), while the inequalities $x < y$ facilitate symbolic reasoning, such as confirming array overflows (e.g., verifying that x is less than $arr.Length$).

The final output of this pentagon domain is greater precision compared to Intervals, as it incorporates symbolic reasoning. However, it is less precise than Octagons, as it cannot capture equalities such as $x + y == 22$.

2 Domains

To commence with the implementation, it is imperative to gain a comprehensive understanding of the functioning of all the relevant domains involved. To accomplish this, we dedicated time to studying the table detailing the various operations outlined in the research paper. Of particular interest are the domains of intervals and strict upper bounds. The synergy between these domains enables us to effectively represent pentagons.

It's evident that the pentagon domain offers greater precision than the interval domain but falls slightly short of the precision provided by octagons. However, for our specific objectives, the pentagon domain suffices, particularly in the context of verifying array accesses. Consequently, we can develop an algorithm that, while not as precise as octagons, serves our intended purposes exceptionally well.

2.1 Intervals and interval environments

Interval environments represent an extension of traditional intervals within the context of our analysis. This domain effectively encompasses a collection of intervals, with each interval specifically associated with an individual variable featured in our study.

2.2 Sub

The abstract domain Sub, which focuses on strict upper bounds, is a specialized instance of the zone abstract domains. It retains symbolic information in the form of $x < y$, and we represent Sub elements using maps, denoted as $x \to y_1, ...y_n$, where it signifies that x is strictly smaller than each of the y_i. This map-based

$$\begin{aligned}
\text{Order:}\quad & \langle b_1, s_1\rangle \sqsubseteq_p \langle b_2, s_2\rangle \iff b_1 \sqsubseteq_b b_2 \\
& \qquad \wedge(\forall \mathbf{x} \in s_2 \forall \mathbf{y} \in s_2(\mathbf{x}).\mathbf{y} \in s_1(\mathbf{x}) \vee \ \sup(b_1(\mathbf{x})) < \inf(b_1(\mathbf{y}))) \\
\text{Bottom:}\quad & \langle b, s\rangle = \bot_p \Rightarrow b = \bot_b \vee s = \bot_s \\
\text{Top:}\quad & \langle b, s\rangle = \top_p \iff b = \top_b \wedge s = \top_s \\
\text{Join:}\quad & \langle b_1, s_1\rangle \sqcup_p \langle b_2, s_2\rangle = \\
& \quad \text{let } b^{\sqcup} = b_1 \sqcup_b b_2 \\
& \quad \text{let } s^{\sqcup} = \lambda \mathbf{x}.s'(\mathbf{x}) \cup s''(\mathbf{x}) \cup s'''(\mathbf{x}) \\
& \qquad \text{where } s' = \lambda \mathbf{x}.s_1(\mathbf{x}) \cap s_2(\mathbf{x}) \\
& \qquad \text{and } s'' = \lambda \mathbf{x}.\{\mathbf{y} \in s_1(\mathbf{x}) \mid \sup(b_2(\mathbf{x})) < \inf(b_2(\mathbf{y}))\} \\
& \qquad \text{and } s''' = \lambda \mathbf{x}.\{\mathbf{y} \in s_2(\mathbf{x}) \mid \sup(b_1(\mathbf{x})) < \inf(b_1(\mathbf{y}))\} \\
& \quad \text{in } \langle b^{\sqcup}, s^{\sqcup}\rangle \\
\text{Meet:}\quad & \langle b_1, s_1\rangle \sqcap_p \langle b_2, s_2\rangle = \langle b_1 \sqcap_b b_2, s_1 \sqcap_s s_2\rangle \\
\text{Widening:}\quad & \langle b_1, s_1\rangle \nabla_p \langle b_2, s_2\rangle = \langle b_1 \nabla_b b_2, s_1 \nabla_s s_2\rangle
\end{aligned}$$

representation allows for highly efficient implementations.

In essence, the fewer constraints we have, the less information is available. Consequently, the ordering within this domain is determined by pointwise superset inclusion. The bottom element contains a contradiction, represented as $x < y \wedge y < x$, while the top element corresponds to a lack of information, symbolized by the empty set.

The join operation entails a pointwise set intersection, ensuring that only relations holding in both incoming branches are retained. Conversely, the meet operation involves a pointwise set union, preserving relations from either the left or right branch. Finally, widening follows the standard definition, retaining constraints that remain stable in successive iterations.

2.3 Pentagons

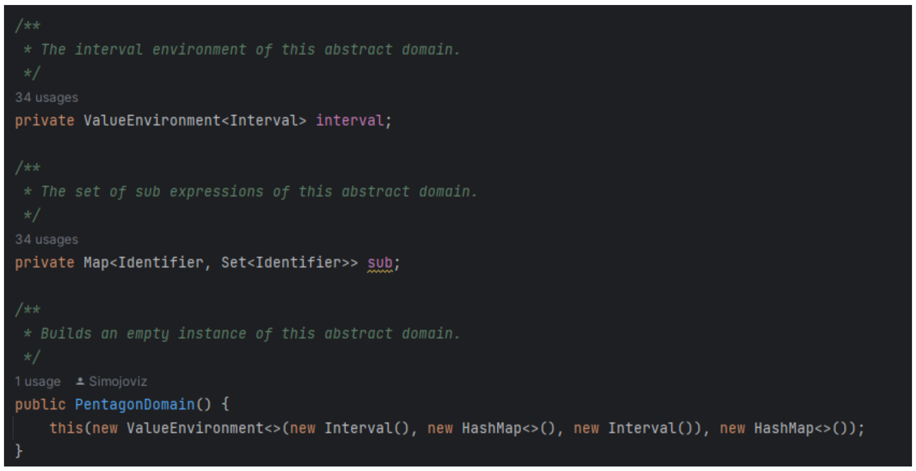
As previously mentioned, elements within the Pentagon domain take the form of $x \in [a, b] \wedge x < y$, where x and y represent variables, and a and b belong to an underlying numerical set, such as Z or Q, which includes extensions for both $-\infty$ and $+\infty$. A pentagon provides both lower and upper bounds for each variable, making it as precise as intervals. However, it distinguishes itself by also maintaining strict inequalities among variables, thus enabling a limited form of symbolic reasoning.

One initial approach might be to combine the numerical properties captured by Intv with the symbolic ones captured by Sub by considering the Cartesian product $Intv \times Sub$. However, this approach essentially runs the two analyses in parallel without any exchange of information between the two domains. A more effective solution involves performing the reduced Cartesian product $Intv \bigotimes Sub$. In essence, the reduced Cartesian product amalgamates pairs that share the same concrete meaning within the product lattice.

3 Implementation

It's important to clarify that certain assumptions underpin our development process. Our primary objective was to create an analysis capable of determining the safety of array accesses. To achieve this, we initially focused on implementing a solution designed to work with the binary search example from the paper. Additionally, we included all possible implementations up to binary expressions. This includes assumptions and assignments, which, as we'll explore later, encompass straightforward operations like negation, addition, subtraction, and division involving numbers and constants.

It's worth noting that we leveraged pre-existing implementations for interval environments available in LiSa, directing our efforts toward the logic underpinning strict upper bounds.



```java
/**
 * The interval environment of this abstract domain.
 */
34 usages
private ValueEnvironment<Interval> interval;

/**
 * The set of sub expressions of this abstract domain.
 */
34 usages
private Map<Identifier, Set<Identifier>> sub;

/**
 * Builds an empty instance of this abstract domain.
 */
1 usage  Simojoviz
public PentagonDomain() {
    this(new ValueEnvironment<>(new Interval(), new HashMap<>(), new Interval()), new HashMap<>());
}
```

3.1 Fields

In the context of the Pentagons domain class, we require two specific fields:

- **interval**: This field is an object of the pre-implemented type `ValueEnvironment<Interval>` which handles the interval aspect of pentagons.
- **sub**: Here, we employ a map to implement the strict upper bound aspect of pentagons. The keys within this map represent variables from the analyzed program, each associated with sets of variables that are strictly bigger.

3.2 recomputePentagon

After some operations on the pentagons we may retrieve some new information about the strict upper bound part using the new retrieved interval, and this method recomputes the sub using the new interval. To do so it checks if for example the upper bound of the variable x is strictly lower than the lower bound of the variable y, in this case we can say that $x < y$. In the last part of the method is implemented the transitivity property of the sub, so if we have that $x < y$ and $y < z$ we can say that $x < z$.

3.3 lub

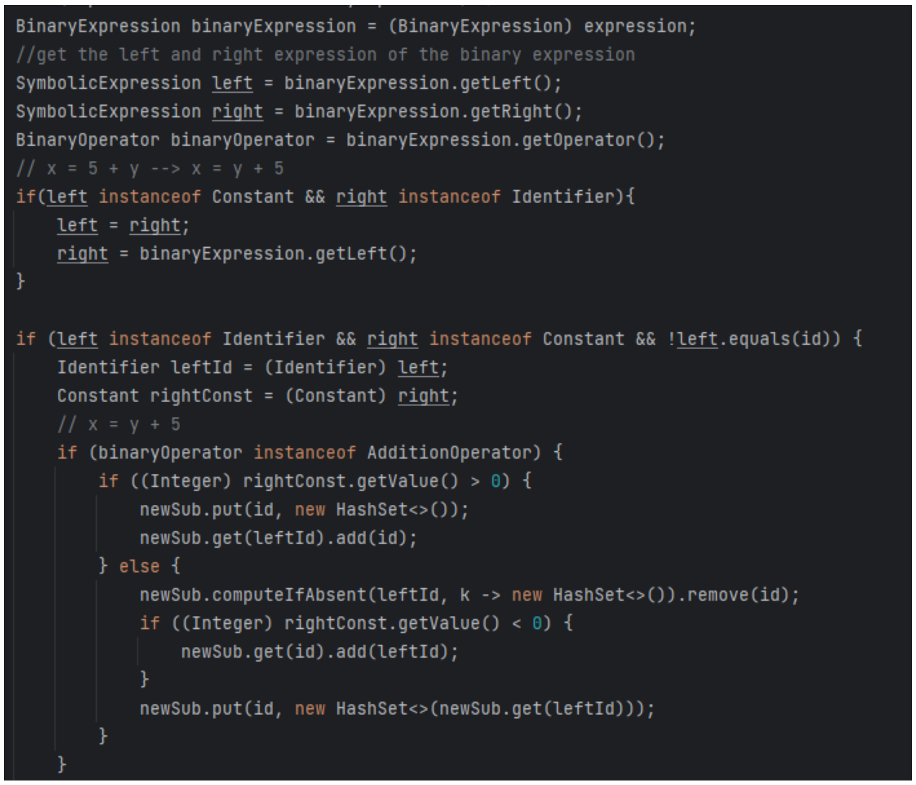
This method computes the least upper bound of two pentagons. This is done by doing the least upper bound of the two interval environments and for the strict upper bound if $x < y$ on both pentagons we add $x < y$ on the strict upper bound of the new pentagon, otherwise if $x < y$ only on one pentagon we check if that is satisfied by the interval of the other pentagon, checking if the upper bound of x is less than the lower bound of y. At the end of the method, a new pentagon is created with the interval and sub just computed, and we return the pentagon obtained by calling the *recomputePentagon* on that new pentagon.

3.4 lessOrEqual

The implementation of the lessOrEqual method is straightforward, as we have employed the definition provided in the paper and translated the expression's logic into Java boolean statements. Initially, we must confirm whether the intervals satisfy the "less or equal" condition, utilizing the existing implementation within the ValueEnvironment class. Subsequently, we must validate an additional condition, specifically:

$$\forall x \in s_2 \forall y \in s_2(x).y \in s_1(x) \vee sup(b_1(x)) < inf(b_1(y))$$

Prior to this verification, we ensure that both subs are initialized for every variable present in the other. This precautionary step prevents errors when accessing non-existent values.



```java
BinaryExpression binaryExpression = (BinaryExpression) expression;
//get the left and right expression of the binary expression
SymbolicExpression left = binaryExpression.getLeft();
SymbolicExpression right = binaryExpression.getRight();
BinaryOperator binaryOperator = binaryExpression.getOperator();
// x = 5 + y --> x = y + 5
if(left instanceof Constant && right instanceof Identifier){
    left = right;
    right = binaryExpression.getLeft();
}

if (left instanceof Identifier && right instanceof Constant && !left.equals(id)) {
    Identifier leftId = (Identifier) left;
    Constant rightConst = (Constant) right;
    // x = y + 5
    if (binaryOperator instanceof AdditionOperator) {
        if ((Integer) rightConst.getValue() > 0) {
            newSub.put(id, new HashSet<>());
            newSub.get(leftId).add(id);
        } else {
            newSub.computeIfAbsent(leftId, k -> new HashSet<>()).remove(id);
            if ((Integer) rightConst.getValue() < 0) {
                newSub.get(id).add(leftId);
            }
            newSub.put(id, new HashSet<>(newSub.get(leftId)));
        }
    }
```

3.5 assign

The assign method holds significant importance as it governs the evolution of the domain following an assignment. It's crucial to note that when an assignment involves a constant, all information about that variable within the subdomains is lost. Consequently, we must remove it from the other subs and empty its own sub. Subsequently, by employing the recompute pentagon, we can reconstruct all sub-instances based on the updated interval environment.

We also conduct verifications for variable assignments of the form $x = y + const$. By checking if the constant is greater than or equal to zero, we can discern whether x is greater than y or vice versa.

In the case of assignments like $x = y[operator]z$, where there are numerous possibilities to consider, we opt to let the sub be determined exclusively using intervals through the recompute pentagon method.

Additionally, it's worth noting that for testing the code presented in the paper, specifically the binary search, we have included code to verify that the result of the mean of two positive values is smaller than each of the two, expressed as $x = (y + z)/2 \rightarrow y >= x \& z \geq x$.

3.6 assume

The assume method holds particular significance, especially in its strong connection to the sub domain, which is of primary interest since it requires manual implementation. Similar to previous methods, we encompass all operations up to binary expressions. To enhance generality, we also consider all possible negations, which is easily achieved as negation is a unary expression, and the removeNegation method facilitates code reuse.

A noteworthy aspect to highlight is that when we encounter an expression such as $x \leq y$, we cannot definitively assert that $y > x$, but we can be certain that all elements greater than y will also be greater than x. Leveraging this concept, we have crafted code to handle all possible logical binary expressions

3.7 satisfies

The satisfies method is quite straightforward, primarily involving the verification of relationships between variables. For expressions that include constants, their satisfiability is determined through interval calculations.

When it comes to sub domains, we follow a process of assessing whether a specific inequality holds within the current sub. For instance, the satisfaction of $x < y$ depends on the presence of y in the subdomain of x. We employ this logic to implement fundamental tests for binary expressions.

4 Results and conclusion

To assess the written implementation, we proceed by crafting test cases and analyzing the outcomes. It is essential to reiterate our primary objective, which is to validate array access.

In our testing, we initially devised numerous test cases, but ultimately retained five key functions representing a wide range of potential computations:

- `subtraction`: This function verifies basic operations such as assignment and subtraction to ensure the correct functioning of both environments and sub.
- `branches`: Here, we aim to test the merging operation using a simple if-else statement.
- `array_access`: In this test, we utilized multiple variables to assess the analysis's capability to handle more than four variables effectively. Additionally, we created an array and populated it with numbers corresponding to their indexes. A notable observation was that, at the point of array access, the index within its sub contained the array's length and was positive. This observation assures us that access errors are impossible.
- `binary_search_with_length_defined`: In this phase, we examined the binary search algorithm outlined in the paper. In this initial version, we provided information about the array's length. This additional information significantly enhances the intervals, aiding the sub in confirming that the index is always less than the array's length during array access. Notably, we simulated array access by performing `num4 = array + index`. Since `array` is undefined, adding a value results in an undefined value. While this is analogous to `num4 = array[index]`, we refrained from using the latter because, when the array is passed as a function parameter, the `array[index]` operation yields "bottom" and disrupts subsequent analysis.
- `binary_search`: In this final test, we deliberately withheld information about the array's actual length. Instead, we leveraged a particularly useful insight: the sub of the mean of two values contains the intersection of the sub present in both values. This principle can be expressed as "given $x = (y + z)/2$, the $sub(x)$ will contain $sub(y) \cap sub(z)$". This concept empowers us to provide the analysis with all the necessary information to yield the expected results. Specifically, we conclude that the index is less than the length and is positive when accessing the array.

In conclusion, this analysis is a potent tool and not overly complex to implement. It furnishes us with a wealth of information about variable values and their relationships, proving invaluable for various scenarios, particularly in addressing array access issues.

All our results have exceeded expectations. A potential avenue for future work involves enhancing the sub by incorporating more operations, including complex expressions, into the analysis.