17-355 Final Project Report: Interprocedural Range Analysis for Array Bounds Checking in Java

Priya Varra

May 11, 2020

Abstract

Range analysis tracks intervals of possible values of variables in programs and has many practical applications in program analysis. In this project, I implemented and tested a context-sensitive interprocedural range analysis for the purpose of array bounds checking in Java. The majority of the project was dedicated to implementing and testing intraprocedural range analysis and experimenting with various widening methods before extending the analysis to a context-sensitive interprocedural analysis.

1 Introduction

Range analysis is a dataflow analysis that tracks the minimum and maximum possible values that integer variables might hold during the execution of a program. Knowing this information about each variable at various points in a program is incredibly useful for detecting a variety of bugs and issues. For instance, range analysis can be used to detect integer overflow, it can be used to eliminate dead code by checking if variables degenerate to constants, and it can be used for branch prediction [1]. For this project, I implemented a context-sensitive interprocedural range analysis for another application called array bounds checking which is the detection of invalid array accesses. The target of my analysis was programs in Java, or more specifically, the Jimple intermediate representation of Java provided by the Soot framework.

2 Design & Implementation

2.1 Abstract State & Definition

To implement context-sensitive interprocedural range analysis, I started by first implementing intraprocedural range analysis, which is used by the interprocedural analysis algorithm when analyzing each individual procedure within a program. For the intraprocedural analysis, I began by referencing the description of interval analysis, which is the same as range analysis, in the Widening Operators and Collecting Semantics notes. As seen in figure 1, the description provides the lattice (L, $\sqsubseteq$) , the abstraction function (α), the join operation ($\sqcup$), and the initial dataflow analysis assumption (σ_0) for individual variables in a program.

$$
\begin{array}{rcl}
L & = & \mathbb{Z}_\infty \times \mathbb{Z}_\infty \qquad \text{where } \mathbb{Z}_\infty = \mathbb{Z} \cup \{-\infty, \infty\} \\
[l_1, h_1] \sqsubseteq [l_2, h_2] & \textit{iff} & l_2 \leqslant_\infty l_1 \wedge h_1 \leqslant_\infty h_2 \\
[l_1, h_1] \sqcup [l_2, h_2] & = & [min_\infty(l_1, l_2), max_\infty(h_1, h_2)] \\
\top & = & [-\infty, \infty] \\
\bot & = & [\infty, -\infty] \\
\sigma_0 & = & \top \\
\alpha(x) & = & [x, x]
\end{array}
$$

Figure 1: Range Analysis Abstraction [2]

For each variable, its abstract value is a pair of elements where the left element is a lower bound and the right element is an upper bound on the value that the variable might hold at a point in the program. Theoretically, each element is a member of $Z \cup \{\infty, -\infty\}$; however, since Java uses 32-bit integers, for this analysis, Z is restricted to the range of integers from -2,147,483,648 to 2,147,483,647. Given two abstract value ranges R_1 and R_2, we know R_1 is at least as precise as R_2 if the interval represented by R_1 is contained inside of the interval represented by R_2. When joining two abstract value ranges, the new lower bound becomes the minimum of the lower bounds of the two ranges, and likewise, the new upper bound becomes the maximum of the upper bounds of the two ranges. The abstraction function maps a concrete integer to an abstract value range where both the lower and upper bound are the integer itself. Finally, the initial dataflow analysis assumption for each variable is $\top$, ($[-\infty, \infty]$) which indicates that the variable can have any integer value [2].

The description above is for an individual variable, but for this analysis, we are interested in tracking the range of values for all variables in a program. We accomplish this by lifting the previously defined lattice over all of the variables in a program. More formally, the abstract state at a program point is a map that contains each variable in the program and its corresponding abstract value range. To join two abstract states, we apply the previously defined join operator point-wise on the abstract value ranges for

each variable in the program, and one abstract state is now at least as precise as another abstract state if, for every variable in the program, the abstract value range from the first abstract state is at least as precise as the abstract value from the second abstract state. For the initial data flow analysis assumption, we create an abstract state where every variable is mapped to $\top$.

2.2 Flow Functions

Once I had implemented the abstract state and analysis definition, I moved on to developing and implementing flow functions for the analysis. Specifically, I created flow functions for addition, subtraction, multiplication, division, negation, and if statements. For the arithmetic operations, I referenced both the provided flow function for addition in the Widening Operators and Collecting Semantics notes as well as the existing method of interval arithmetic to develop the flow functions detailed below.

For all arithmetic expressions of the form $x = y\ op\ z$, if $\sigma(y) = \bot \vee \sigma(z) = \bot$, then $f_R[[x := y\ op\ z]](\sigma) = \sigma$.

Otherwise,

$$f_R[[x := y + z]](\sigma) = \sigma[x \mapsto [l, h]]$$
$$\text{where } l = \sigma(y).low +_\infty \sigma(z).low$$
$$\text{and } h = \sigma(y).high +_\infty \sigma(z).high$$

$$f_R[[x := y - z]](\sigma) = \sigma[x \mapsto [l, h]]$$
$$\text{where } l = \sigma(y).low -_\infty \sigma(z).high$$
$$\text{and } h = \sigma(y).high -_\infty \sigma(z).low$$

For multiplication and division let,

$$a = \sigma(y).low\ op_\infty\ \sigma(z).low$$
$$b = \sigma(y).low\ op_\infty\ \sigma(z).high$$
$$c = \sigma(y).high\ op_\infty\ \sigma(z).low$$
$$d = \sigma(y).high\ op_\infty\ \sigma(z).high$$

Then,

$$f_R[[x := y * z]](\sigma) = \sigma[x \mapsto [l, h]]$$
$$\text{where } l = min(a, b, c, d)$$
$$\text{and } h = max(a, b, c, d)$$

$$f_R[[x := y/z]](\sigma) = \sigma[x \mapsto \top]$$
$$\text{if } 0 \in \sigma(y) \vee 0 \in \sigma(z)$$

$$f_R[[x := y/z]](\sigma) = \sigma[x \mapsto [l, h]]$$
$$\text{where } l = min(a, b, c, d)$$
$$\text{and } h = max(a, b, c, d)$$

For unary negation,

$$f_R[[x := -y]](\sigma) = \sigma$$
$$\text{if } \sigma(y) = \top \vee \sigma(y) = \bot$$

$$f_R[[x := -y]](\sigma) = \sigma[x \mapsto [l, h]]$$
$$\text{where } l = -\sigma(y).high$$
$$\text{and } h = -\sigma(y).low$$

For this analysis, I used the ForwardBranchedFlowAnalysis class in Soot which allowed me to propagate different abstract states down the true and false branches of if statements. By default, the if statement flow functions pass on the input abstract state without modification to both branches. However, if the condition of the if statement compares a variable with an integer constant, then the flow functions utilize the additional information provided by the comparison to send appropriate abstract states to each of the true and false branches. For this case, the flow function uses the concept from the Widening Operators and Collecting Semantics notes of propagating $\bot$ when the range of the variable is infeasible given the condition in the if statement [2]. The flow functions for the true branch are outlined below where c is an integer constant.

$$f_R[[if\ x = c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto \bot]$$
$$\text{if } c \notin \sigma(x)$$

$$f_R[[if\ x = c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto \alpha(x)]$$
$$\text{if } c \in \sigma(x)$$

$$f_R[[if\ x \neq c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto \bot]$$
$$\text{if } \sigma(x) = \alpha(c)$$

$$f_R[[if\ x\ \neq c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto [l, h]]$$
$$\text{where } l = c + 1 \text{ if } \sigma(x).low = c$$
$$\text{else } l = \sigma(x).low$$
$$\text{and } h = c - 1 \text{ if } \sigma(x).high = c$$
$$\text{else } h = \sigma(x).high$$

$$f_R[[if\ x\ \leq c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto \bot]$$
$$\text{if } \sigma(x).low > c$$

$$f_R[[if\ x\ \leq c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto [l, h]]$$
$$\text{where } l = \sigma(x).low$$
$$\text{and } h = c \text{ if } c \in \sigma(x)$$
$$\text{else } h = \sigma(x).high$$

$$f_R[[if\ x\ \geq c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto \bot]$$
$$\text{if } \sigma(x).high < c$$

$$f_R[[if\ x\ \geq c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto [l,h]]$$
$$\text{where } l = c \text{ if } c \in \sigma(x)$$
$$\text{else } l = \sigma(x).low$$
$$\text{and } h = \sigma(x).high$$

$$f_R[[if\ x < c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto \bot]$$
$$\text{if } \sigma(x) = \alpha(x) \vee \sigma(x).low > c$$

$$f_R[[if\ x < c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto [l,h]]$$
$$\text{where } l = \sigma(x).low$$
$$\text{and } h = c - 1 \text{ if } c \in \sigma(x)$$
$$\text{else } h = \sigma(x).high$$

$$f_R[[if\ x > c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto \bot]$$
$$\text{if } \sigma(x) = \alpha(x) \vee \sigma(x).high < c$$

$$f_R[[if\ x > c\ goto\ n]]_T(\sigma) = \sigma[x \mapsto [l,h]]$$
$$\text{where } l = c + 1 \text{ if } c \in \sigma(x)$$
$$\text{else } l = \sigma(x).low$$
$$\text{and } h = \sigma(x).high$$

For the flow functions for the false branch, the conditional operator in the if statement is first logically negated before calling the corresponding flow function for the true branch.

2.3 Widening

After designing and implementing the flow functions, my next step was implementing and applying widening. For a dataflow analysis, in order to ensure the termination of the analysis, one of the requirements is that the lattice for the analysis has finite height.

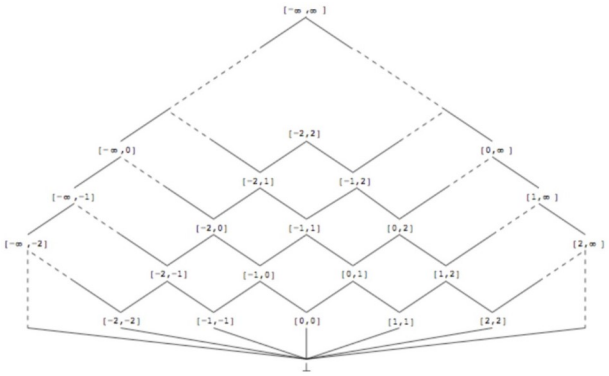

Figure 2: Range Analysis Lattice [1]

Range analysis, however, theoretically has a lattice of infinite height, and even when the set of integers in the lattice is restricted to those that can be represented by 32-bit integers, the height of the lattice is still large enough that the analysis may take too long to terminate for practical use [1]. This issue can be addressed by applying widening which is the concept of jumping up the lattice so that it can no longer grow when the possibility arises of climbing up an extremely large and/or infinite ascending chain during the analysis. In general, widening results in a loss of precision, and by using different methods of widening and selectively applying it during the analysis, the amount of precision that is lost can vary. For my analysis, I implemented three different widening methods: simple widening, delayed widening, and widening with constants. For all three methods, I only applied them at program points corresponding to the heads of loops -as suggested in the Widening Operator and Collecting semantics notes [2]. I found the loop heads in a program by using the LoopFinder class from the Soot framework.

Simple Widening

The first method of widening that I implemented was simple widening which is defined in the Widening Operators and Collecting Semantics notes and can be seen in the figure below.

$$W(\bot, l_{current}) = l_{current}$$

$$W([l_1,h_1],[l_2,h_2]) = [min_W(l_1,l_2), max_W(h_1,h_2)]$$

$$\begin{array}{ll} \text{where } min_W(l_1,l_2) = l_1 & \text{if } l_1 \leqslant l_2 \\ \text{and } min_W(l_1,l_2) = -\infty & \text{otherwise} \\ \text{where } max_W(h_1,h_2) = h_1 & \text{if } h_1 \geqslant h_2 \\ \text{and } max_W(h_1,h_2) = \infty & \text{otherwise} \end{array}$$

Figure 3: Simple Widening Definition [2]

For this method of widening, as soon as we enter an ascending chain, we immediately jump to the top of the lattice. This means that if we encounter increasing upper limits on the last two ranges for a variable at a loop head, we assume an upper bound of ∞, and if we encounter decreasing lower limits on the last two ranges for a variable at a loop head, then we assume a lower bound of $-\infty$.

Delayed Widening

The second method of widening that I implemented was delayed widening. For this method, instead of applying a widening operator every time a loop head is visited, the application of the operator is delayed by waiting until the number of visits to the loop head surpasses a predetermined threshold [4]. This may result in increased precision because by waiting to apply the operator, the variable may come to a fixpoint on its own, disregarding the need to jump to $-\infty$ or ∞. For my implementation of delayed widening, I used the simple widening operator defined above after exceeding the delay threshold.

Widening with Constants

The third and final method of widening that I implemented was widening with constants which is also defined in the Widening Operators and Collecting Semantics notes and can be seen in the figure below.

$$
\begin{array}{lll}
W(\bot, l_{current}) & = l_{current} & \\
W([l_1, h_1], [l_2, h_2]) & = [min_K(l_1, l_2), max_K(h_1, h_2)] & \\
 & \text{where } min_K(l_1, l_2) = l_1 & \text{if } l_1 \leqslant l_2 \\
 & \text{and } min_K(l_1, l_2) = max(\{k \in K | k \leqslant l_2\}) & \text{otherwise} \\
 & \text{where } max_K(h_1, h_2) = h_1 & \text{if } h_1 \geqslant h_2 \\
 & \text{and } max_K(h_1, h_2) = min(\{k \in K | k \geqslant h_2\} & \text{otherwise}
\end{array}
$$

Figure 4: Widening with Constants Definition [2]

This method of widening utilizes constants found in the program to infer when variables might reach a fixpoint. Specifically K in the definition above is a set containing constants extracted from the program in addition to $-\infty$ and ∞, and each time the widening operator is applied, if an ascending chain is encountered, instead of immediately jumping to $-\infty$ or ∞, it jumps to either the next smallest constant or the next largest constant in K. As a result, this method of widening gives the abstract value for a variable an additional chance to converge to a fixpoint before giving up precision. For my implementation, I chose to extract constant integer values from loop heads and arithmetic expressions in each program to create the set K [2].

2.4 Array Bounds Checking

For my project, the main goal of implementing range analysis was to utilize the results to perform array bounds checking. To accomplish this, I created and reported four error/warning messages:

NEGATIVE_INDEX_ERROR
POSSIBLE_NEGATIVE_INDEX_WARNING
OUT_OF_BOUNDS_INDEX_ERROR
POSSIBLE_OUT_OF_BOUNDS_INDEX_WARNING

While performing the range analysis, I kept track of each array in the program and their corresponding initialized sizes, and at each program point, similar to the integer sign analysis done for homework 5, I stored the abstract state of the variables for each program point. Then, once the range analysis finished, I examined the stored abstract state for each program point corresponding to an array indexed by a variable. If the abstract value range for the index variable was strictly negative, I reported NEGATIVE_INDEX_ERROR at that line. If the abstract value range for the index variable contained both negative and non-negative values, I reported POSSIBLE_NEGATIVE_INDEX_WARNING at that line. If the abstract value range for the index variable was strictly greater than or equal to the size of the array, I reported OUT_OF_BOUNDS_INDEX_ERROR at that line. And finally, if the abstract value range for the index variable contained values that were both less than and greater than or equal to the size of the array, I reported POSSIBLE_OUT_OF_BOUNDS_INDEX_WARNING.

2.5 Interprocedural Analysis

After I implemented and tested the above steps for intraprocedural range analysis, I finally extended it to a context-sensitive interprocedural analysis. From homework 5, I already had an implementation of the context-sensitive interprocedural analysis algorithm from the Interprocedural Analysis notes. To adapt it for this analysis, I replaced the intraprocedural subroutine with my intraprocedural range analysis implementation. For the context-sensitive component of the analysis, call strings were used to differentiate contexts with the call strings being cut off after two call sites [3].

3 Results & Discussion

After writing test cases to check the correctness of the abstract state and flow functions, the widening operators, and the error reporting functionality as well as the context-sensitivity of the interprocedural analysis [1], I focused on finding test cases that would demonstrate the effects of the different widening methods on the precision and time complexity of range analysis. The following program was one simple example that was particularly illustrative [5].

```
int x;
int y;
int[] array = new int[1001];

x = 0;
y = 1;

while(x < 1000) {
    x = x + 1;
    y = 2 * x;
    y = y + 1;
}

/* Point A */

int ignore = array[x];
ignore = array[y];
```

For this code, I ran range analysis using the simple widening operator; delayed widening with thresholds of 10, 100, and 1000; and widening with constants. The resulting abstract states at point A in the program are summarized in the figure on the next page.

As seen in the figure, the simple widening method along with the delayed widening with thresholds of 10 and 100 had the least precise results with the upper

[1] All of the test cases can be found in the src/test/java/inputs directory of the GitHub repository

Widening Method	σ(x)	σ(y)
Simple Widening	[1000, ∞]	[1, ∞]
Delayed Widening, k = 10	[1000, ∞]	[1, ∞]
Delayed Widening, k = 100	[1000, ∞]	[1, ∞]
Delayed Widening, k = 1000	[1000, 1000]	[1, 2001]
Widening with Constants	[1000, 1000]	[1, ∞]

Figure 5: Widening Method Results at Point A

bounds for both x and y being ∞. Delayed widening with a threshold of 1000 had the most precise results with an integer upper bound for both x and y, and the precision from widening with constants was in between with an integer upper bound for x but ∞ as an upper bound for y. For widening with constants, the extracted constants were $K = \{-\infty, 1, 2, 1000, \infty\}$.

These results make sense because this piece of code reaches a fixpoint after 1000 iterations without any widening operators. Thus, for the simple widening and delayed widening with thresholds of 10 and 100, the head of the loop is not visited enough times before the widening operator is applied causing the upper bound to jump to ∞ for both x and y. For the delayed widening with a threshold of 1000, the analysis would have already reached a fixpoint and terminated before the widening operator could be applied, resulting in no loss of precision. For widening with constants, because the analysis extracted the bound of 1000 from the head of the loop, the widening operator allowed the analysis to jump to 1000 instead of ∞ as the upper bound for x and y the first time it was applied. This allows x to come to a fixpoint since it is bounded by the condition in the loop, and one more application of the widening operator then gives up precision on y and causes its upper bound to jump to ∞.

From this, we can see the advantages and disadvantages provided by the different widening methods. For delayed widening, if we are unlucky with picking the threshold, then we achieve the same precision as simple widening but have to run the analysis for more iterations. However, if we are lucky with picking the threshold, then the results of the analysis can be quite precise. The widening with constants method appears to provide a balance between precision and time complexity, because it retained precision with respect to x while running significantly less iterations than the delayed widening with a threshold of 1000.

4 Conclusion

Overall, I was able to successfully implement a baseline range analysis with a variety of widening methods, a context-sensitive interprocedural component, and error reporting for array bounds checking in Java. However, there is considerable potential for further exploration and improvements to the analysis. For instance, I did not test this implementation on nested loops and examining the results of different widening methods on them and adapting the analysis accordingly presents itself as a very interesting direction for future work. In addition, I would like to look further into developing and testing different methods of generating both the thresholds for delayed widening as well as the constants for widening with constants to investigate their effect on the precision of the analysis.

References

[1] Campos, Victor Hugo Sperle, et al. "Speed and precision in range analysis." Brazilian Symposium on Programming Languages. Springer, Berlin, Heidelberg, 2012.

[2] Le Goues, Claire. "Widening Operators and Collecting Semantics." Program Analysis, Carnegie Mellon University. 2020.

[3] Le Goues, Claire. "Interprocedural Analysis." Program Analysis, Carnegie Mellon University. 2020.

[4] Sutre, Grégoire. "Range Analysis with Widening and Narrowing" Software Verification, University of Bordeaux. 2019.

[5] "Widening and Narrowing in Variable Range Analysis." LARA - Lab for Automated Reasoning and Analysis, lara.epfl.ch/w/sav09:widening_in_variable_range_analysis.

Appendix

To run and test both the interprocedural and intraprocedural range analysis implementations, please refer to the README in the GitHub repository containing the source code for this project.