CSSE4630 Assignment 1: Array Bounds Analysis

Mark Utting, ITEE, UQ

Version 1.2

1 Introduction

A software development company with a strong emphasis on cybersecurity is employing you to develop a static analysis for array bounds checking.

They have two main goals for the analysis:

- Security: they want to be able to guarantee that every program they write is free from array bounds errors. The compiler for their programming language already generates code that checks the array bounds at runtime, but they want to give higher security assurances than just this. They want the static analysis to be able to prove that some array accesses are definitely safe (within bounds) and all remaining array accesses will generate warnings that will checked by a code review team to ensure that those array accesses are also within bounds. It is desirable for the static analysis to be able to put most of the array accesses into the first category (provably safe), in order to minimise work for the code review team.
- Efficiency: Runtime checking of array bounds accesses has a significant overhead, so the company plans to modify their compiler to omit the bounds checking code for the cases where the static analysis can prove that the array access is within bounds.

Your task is to design and implement a suitable static analysis, for the TIP (Tiny Imperative Language) language, extended with mutable arrays.

2 TIP with Array Creation and Reading [20 Marks]

The company uses a safe subset of TIP (no pointers, records, or dynamic function calls), extended with integer arrays, where each array has a fixed size that is known at compile time:

- the expression $newarray\ N$ creates an array of size N (which must be a constant non-negative integer value) and fills it with zeroes.
- the expression $A!E$ is an array read, and introduces the bounds-checking requirement that $0 \leq E < N$, where N is the length of A.
- the language of types is extended with the type $array(N)$, where N is a non-negative integer constant (the size of the array). So the complete syntax for types is: $\tau ::= int|\&\tau|array(N)|(\tau,\ldots,\tau) \rightarrow \tau$.

Your Task: extend the TIP language implementation by adding the new kinds of expressions, as described above. You do NOT need to extend the interpreter or the typechecker to handle arrays, at this stage. However, you will need to extend the TIP implementation in the following ways:

- add a new keyword `newarray` in `TipParser.scala`, and also add that keyword to the *keywords* set.
- define a new type of AST expression called `ANewArray` that extends `AExpr`.
- extend the TIP parser, to parse array creation expressions: `newarray N`, where `N` is an integer constant, and return an AST node of type `ANewArray`.
- extend `DepthFirstAstVisitor.visitChildren` to handle your `ANewArray` expressions, by just visiting the N part.
- define a new binary operator for array indexing expressions, `A!E`. This should be a subclass of `BinaryOperator`, so that you can use this operator inside `ABinaryOp`.
- extend the TIP parser, to parse array index expressions, with the same precedence as multiplication and division. (Start by adding these just to right-hand-side expressions — you can add them to the left-hand-side of assignments later.) Note that the TIP parser uses the *parboiled2* parser combinators, which are documented on their GitHub site: `https://github.com/sirthias/parboiled2`

Test your implementation by analysing some of the array test programs that are supplied. After you have implemented the `newarray` expression, you should be able to parse `array_new.tip`. After you have implemented the `A!E` array reads as well, you should be able to parse `array_read1.tip`.

```
tip tests/array_new.tip
tip tests/array_read1.tip
```


3 Static Bounds Checking for Array READS [30 marks]

Your Task: Design and implement a static analysis (based on interval analysis) that checks each array read to determine if the index is provably within bounds, or whether it needs a runtime bounds check and a warning about the bounds (which means that the code review team must review that access).

As you design your bounds analysis, record your extra interval rules in a file `BoundsAnalysis.txt` (or `BoundsAnalysis.docx` if you prefer). These rules will be in addition to the *eval* and constraint rules given in Section 6.1 of the textbook. You must submit this file as part of your solution.

Note that you may find it helpful here to review the work that you did in Workshop 6, where you experimented with interval analysis and implemented widening. You will need at least a basic form of widening to be able to run an interval analysis.

After or during the final pass of your static analysis, your analysis should print a report to standard output, with one line for each array read, in the following format. For example, if you run the command:

```
tip -interval wlrw tests/array_read1.tip
```

it should report that all array bounds are safe. If you do the same command with `array_read2.tip` your array bounds checking should include the following bounds-checking output lines (shown in blue) as the result of your analysis:

```
[info] Running tip.Tip -interval wlrw tests/array_read2.tip
[info] Normalized analysis of tests/array_read2.tip written to out/array_read2.tip__normalized.tip
Array bounds analysis results:
  ArrayRead:4:11: safe
  ArrayRead:5:15: safe
```

```
  ArrayRead:6:15: WARNING: index may be outside bounds
[info] Interval analysis of tests/array_read2.tip written to out/array_read2.tip__interval.dot
```

Hints: the abstract lattice analysis of expressions is done within the `ValueAnalysis` superclass of `IntervalAnalysis`, which is not specific to intervals. To record your analysis results in that class, but make them available in the `IntervalAnalysis` class, I suggest that you add a dummy method in the `ValueAnalysis` class, for recording warnings:

```
    def saveWarning(loc: String, msg: String): Unit = ???  // for array bound checking.
```

An example call to this might be: `saveWarning("ArrayRead:" + exp.loc.toString, "safe")`, where `exp` is the expression we are analysing — so `exp.loc` gives the line and column number of that expression.

Then in the `IntervalAnalysis` class, override this dummy method with a more useful implementation. Here is a Scala trait (like an abstract superclass) that you can inherit into `IntervalAnalysis` to override the default `saveWarning` method, save just the most-recent message for each location string, and then print out all the messages when needed.

```
/**
 * A mixin for remembering the latest message for each location.
 */
trait FinalWarnings extends ValueAnalysisMisc {
  val warnings = collection.mutable.LinkedHashMap[String, String]()

  override def saveWarning(key: String, msg: String): Unit = {
    warnings(key) = msg
  }

  def printWarnings() = {
    println("Array bounds analysis results:")
    for ((key,msg) <- warnings) {
      println("  " + key + ": " + msg)
    }
  }
}
```


In the various interval analysis classes in `IntervalAnalysis.scala`, you should also override the `analyze` method to call `printWarnings` after the whole analysis is finished. For example:

```
    override def analyze(): lattice.Element = {
      val result = super.analyze()
      printWarnings()
      result
    }
```

Once you have finished implementing your static bounds analyser, run it on the following simple test programs. For each test program *P.tip*, save the output of your run into a file `out/P.tip.out`. These will be part of your submission, to demonstrate how well your bounds checker works.

For example, run the following commands at this stage:

```
tip -interval wlrw tests/array_new.tip >out/array_new.tip.out
tip -interval wlrw tests/array_read1.tip >out/array_read1.tip.out
tip -interval wlrw tests/array_read2.tip >out/array_read2.tip.out
```

4 Static Bounds Checking for Array WRITES [30 marks]

Your Task: Extend your static analysis to also check each array write to determine if the index is provably within bounds, or whether it needs a runtime bounds check and a warning about the bounds (which means that the code review team must review that access).

First you will need to extend the TIP language to allow array index expressions on the left-hand-side of assignments:

- the assignment $A!E = E2$ is an array write, and introduces the bounds-checking requirement that $0 \leq E < N$, where N is the length of A.

Document your additional array-update bounds-checking rules in your `BoundsAnalysis.txt` file (or `BoundsAnalysis.docx`). These rules will be in addition to the *eval* and constraint rules given in Section 6.1 of the textbook. Do not forget to submit this file as part of your solution.

Hints:

- extend the TIP parser, to parse array index expressions on the left side of assignments.
- since the left-hand-side of an assignment is an `Assignable` object, you will need to change the definition of `ABinaryOp` so that it is a subclass of `Assignable`.
- you should also update `TipNormalizer` and `AstPrinter` to handle these array index expressions in the left side of assignments.
- then extend your interval analysis to check the array bounds of these array writes.

After implementing your bounds analysis of array writes, try running the command:


```
tip -interval wlrw tests/array_write1.tip
```

You should see output that includes the following bounds-checking output lines (shown in blue):


```
[info] Running tip.Tip -interval wlrw tests/array_write1.tip
[info] Normalized analysis of tests/array_write1.tip written to out/array_write1.tip__normalized.tip
Array bounds analysis results:
  ArrayWrite:4:7: safe
  ArrayWrite:5:7: safe
  ArrayRead:6:13: safe
[info] Interval analysis of tests/array_write1.tip written to out/array_write1.tip__interval.dot
```

Once you have finished implementing your static bounds analyser, run it on the following simple test programs. For each test program *P.tip*, save the output of your run into a file `out/P.tip.out`. These will be part of your submission, to demonstrate how well your bounds checker works.

For example, run the following commands at this stage (in addition to the programs you tested previously).

```
tip -interval wlrw tests/array_write1.tip >out/array_write1.tip.out
tip -interval wlrw tests/array_write2.tip >out/array_write2.tip.out
tip -interval wlrw tests/array_write3.tip >out/array_write3.tip.out
```

5 Customer Responses

You test your static analysis thoroughly and then release it to your customers. They are happy — for a week or so!

Then they report that their programmers are complaining that the bounds analysis is reporting too many false positives. Their review teams are spending too much time reviewing potential out-of-bounds reports that turn out to be obviously within bounds. They want a more precise analysis with fewer false positives!

They would also like to have a static typechecker for the extended TIP language.

You have limited time, so you have to prioritise which of these two requests to work on.

6 Option A: Path-Sensitive Bounds Analysis [20 marks]

NOTE: this part of the assignment is more difficult than the previous parts. Attempt it only after you have completed the previous steps.

After investigating the complaints from your customer, you find that the above bounds analysis works well for straight-line code, but does not handle programs with if-else branches and loops in a very precise way.

Try your analysis on `tests/array_if1.tip`. You will find that it reports possible out-of-bounds errors (false positives), even though the program actually uses if-else statements to ensure that the array accesses are within bounds. This lack of accuracy is because the effects of the control flow are not included in your analysis.

Your Task: add *assertions* into the (internal) TIP language and incorporate them into your bounds analysis so that the analysis is more accurate when programs contain branches and loops.

Steps:

- Add an *assert* statement to the language, as a subclass of `AStmtInNestedBlock`.
- You will also need to add this assert statement into `AstPrinters`, `DepthFirstAstVisitor` and `TipNormalizers`. Hint: I looked at the usages of `AErrorStmt` and used those usages as a guideline for the assert case.
- You do not have to add it to the parser, since this assert statement will be for internal use only.
- You need to insert *assert* statements after each condition check, where the flow of control changes in `while` and `if` statements. See Section 7.1 of the textbook for details. Hint: a nice place to do this is in `TipNormalizers.normalizeStmtInNestedBlock`.
- Extend your bounds analysis to handle assertions, by implementing the rules for `assert(X>E)` and `assert(E>X)` that are discussed on page 84 of the textbook.

After implementing your bounds analysis of array writes, try running the command:

```
tip -normalizereturns -interval wlrw tests/array_if1.tip
```

Note that you must include the **-normalizereturns** flag from now on, to ask TIP to run your normalizer. (By default, TIP does no normalization).

You should see output that includes the following bounds-checking output lines (shown in blue):

```
[info] Running tip.Tip -normalizereturns -interval wlrw tests/array_if1.tip
[info] Normalized analysis of tests/array_if1.tip written to out/array_if1.tip__normalized.tip
Array bounds analysis results:
  ArrayWrite:10:11: safe
[info] Interval analysis of tests/array_if1.tip written to out/array_if1.tip__interval.dot
```

Once you have finished implementing your path-sensitive static bounds analyser, run it on ALL the test programs in the `tests` folder. For each test program *P.tip*, save the output of your run into a file `out/P.tip.out`. These will be part of your submission, to demonstrate how well your bounds checker works.

For example, here is a bash command to run them all:

```
#!/usr/bin/env bash
for T in $(cd tests; ls *.tip)
do
    echo $T
    ./tip -normalizereturns -interval wlrw tests/$T >out/$T.out
done
```

7 Option B: Typechecking Arrays [20 marks]

Your customers would like to have a static typechecker for the TIP language, extended with arrays, and have offered to pay for the development of the typechecker.

Your Task: implment a static typechecker for TIP, including the extra type constraints associated with arrays.

Note: This 'Typechecking' task is an *alternative* task to the previous 'Assertions' task. You can attempt either task. If you attempt both, then your mark for these two sections will be $max(A, T)$, where A is your mark from the 'Assertions' task and T is your mark from the 'Typechecking' task.

You will first need to implement the typechecking rules for standard TIP programs in `TypeAnalysis.scala`, since they not fully implemented.

Then you can add the new $array(N)$ type as a subclass of `TipType` in `Types.scala`, and implement the additional typechecking rules for arrays.

See Section 3.2 of the textbook for the standard TIP type constraints. Here are the additional array-specific typechecking constraints for our custom TIP extension.

$$\begin{array}{rl} A!E & [\![A]\!] = array(N) \wedge [\![E]\!] = int \wedge [\![A!E]\!] = int \\ newarray\ N & [\![N]\!] = int \wedge [\![newarray\ N]\!] = array(N) \end{array}$$

8 Submission

This assignment is due on Friday of Week 10 (16-Oct-20 6pm Queensland time).

Submit your whole repository on Blackboard, in a single *.zip file. Make sure you include:

- the 'src' folder, containing all the Scala source code, including your modifications.
- your `BoundsAnalysis.txt` file (or `BoundsAnalysis.docx`) that documents the bounds-checking rules that you have designed. You can also explain any design decisions or limitations in that file, if needed.
- the 'out' folder, containing all your final `out/array*.tip.out` files and all the `*.dot` files generated by your analysis. These show the final output from each run of your bounds analyser on the various test programs.