Exercises

Sign Analysis

Points	Submission due	Status
0 / 30	Jul 1, 2025 17:00	No graded result

Tasks:

Sign Analysis

In this second graded assignment, you will implement a sign analysis. A sign analysis is a data-flow analysis that propagates the signs $(-, 0, +)$ of the (integer) number values through the program. This allows, for example, to check for division by zero or array accesses using negative indices. The analysis must be implemented in Java and it will target Java programs. You will use the ASM byte-code instrumentation framework to instrument the sign analysis.

General Task Description

The Analysis

For our implementation, we will only consider a subset of Java: we are only interested in the `int` values of the programs that we analyse. This means, the analysed programs can contain arbitrary code but we only reason about the `int` values. We build an abstraction of potential values for our analysis; these values are the sign values, each number can have one of $(-, 0, +)$. We utilise a lattice to represent these abstract values and their relationships.

Our analysis will receive a class and a method of this class to be analysed as an input. The analysis will check for division-by-zero errors and negative array indices within the root method of the analysis. Depending on the analysis results, your implementation shall either output that no violations have been found or the kind of violation together with the corresponding line number where the violation ocurred. In case our analysed root method calls other methods that return an integer value, we will have to derive the sign value of this returned value. Thus, our analysis is considered inter-procedural as we follow method calls to infer the sign values of their returned value. However, we will ignore violations that occur in called methods and only report violations that might occur in the analysed root method.

Since our analysis is static, it can only approximate certain scenarios. Consider the following source code snippet:

```java
int subtract(int b) {
    return b - 42;
}
```

Let us assume that we know `b` can only be a positive number. Then the result of the `subtract` method can either be positive, zero, or negative, depending on the concrete value of `b`. Since we do not know the concrete value in the abstract domain of signs, we have to assume that the result can be any value of our abstract domain.

To deal with this, our analysis distinguishes between two types of violation notifications: warnings and errors. A warning is emitted, if the abstract value of an array index *could* be negative or the divisor *could* be zero. An error is emitted, if the abstract value of an array index is *for sure* negative, or the divisor is *proved* to be zero.

Implementation

We provide you a main class and interfaces for a lattice and a transfer relation that shall be the basis of your implementation. The functionality shall be implemented by you. Before you start, read Section 8.2 of the ASM manual on how one can implement their own data-flow analyses with ASM.

Start your implementation by defining the lattice and its elements. Think about which elements are needed to model all possible signs of an integer value. The analysis only needs to support integer (`int`) values, thus you do not need to handle values of other types.

Utilise ASM's `Analyzer` to implement your data-flow analysis. It requires you to instantiate an `Interpreter` that performs the symbolic interpretation during the integrated analysis. You need to implement your own interpreter for the lattice elements you have defined. Have a look at the `BasicInterpreter` of ASM, which is somewhat similar to what the interpreter for the sign analysis needs to do.

As already mentioned, we are only considering `int` values for our analysis. Furthermore, we are only interested in the following four basic arithmetic operations: addition, subtraction, multiplication, and division. You need to implement the semantics of these operations in terms of our abstract domain (the number's sign) in your implementation of the `TransferRelation` interface. The only additional unary operation we want to support is sign negation for integers. You do not need to consider other operations, such as, for example, bit shifts.

The analysis shall be inter-procedural. Hence, if the analysed method calls another method, we need to derive the correct lattice element from the method's return value. To determine this lattice element, you can simply execute another sign analysis procedure on the called function. You can assume that the called method is located within the same class as the root method we are analysing. As a fallback value, e.g. if an exception gets

The `Analyzer` provides you an array of `Frames`, which correspond to the operation stack *before* an instruction is executed. After implementing the `analyse()` method in `SignAnalysisImpl`, iterate through the list of instructions and their corresponding frames via the `extractAnalysisResults()` method and check for the following properties:

- *Division by Zero*, that is, the `idiv` instruction is called and the top-most value on the current frame might be zero. We distinguish two cases: the error case, that is, the division by zero *will* happen, and the warning case, that is, the division by zero *might* happen. This will be reported to the command line.
- *Negative Array Index*, that is, the array index might be negative when attempting to read from or write to an array. Again, we distinguish two cases: the error case, where the index *is* negative and the warning case, where the index *might* be negative. This will also be reported to the command line.

The base implementation provides an enum `AnalysisResult` that has a symbolic constant for each of those four scenarios. A string value is attached to the enum items which will be used for printing the output. We are only interested in violations that occur in the analysed method and ignore violations that might occur in methods that get called within the analysed method.

Additional Remarks:

- Make sure to add the suffix `Test` to all your unit tests and that you place them in the appropriate `test` directory.
- If you, **in any way**, make use of LLMs such as ChatGPT, upload the prompts you send to the LLM together with the answers you obtained in a folder called LLM. Furthermore, ensure to annotate every piece of code you write with the help of LLMs.
- You can trigger the mutation analysis on your machine by executing `mvn clean test pitest:mutationCoverage`, which outputs an `XML` file in `target/pit-reports` that tells you exactly which mutants have survived.
- The lecture will cover the topics *Abstract Interpretation* and *Lattices* on June 17th.

Checks

Checks at the Unit Level

Checks for the transfer relation.

1. ◉ **The evaluate(int) method of the TransferRelation works as expected** No results
2. ◉ **The evaluate(Operator, SignValue) method of the TransferRelation works as expected** No results
3. ◉ **The evaluate(Operator, SignValue, SignValue) method of the TransferRelation works as expected** No results

Checks for the lattice.

1. ◉ **The join works as expected** No results
2. ◉ **The less-or-equal relation works as expected** No results

Checks for the abstract value.

1. ◉ **The join works as expected** No results
2. ◉ **The less-or-equal relation works as expected** No results
3. ◉ **Checking whether a value is zero** No results
4. ◉ **Checking whether a value might be zero** No results
5. ◉ **Checking whether a value is negative** No results
6. ◉ **Checking whether a value might be negative** No results

Checks at the Functional Level

1. ◉ **Correct tool output for functional test add** No results
2. ◉ **Correct tool output for functional test allCases** No results
3. ◉ **Correct tool output for functional test bar** No results
4. ◉ **Correct tool output for functional test div** No results
5. ◉ **Correct tool output for functional test first** No results
6. ◉ **Correct tool output for functional test foo** No results
7. ◉ **Correct tool output for functional test ifelse** No results
8. ◉ **Correct tool output for functional test loop0** No results
9. ◉ **Correct tool output for functional test twoErrors** No results
10. ◉ **Correct tool output for functional test divZeroCall** No results
11. ◉ **Correct tool output for functional test divMaybeZeroCall** No results
12. ◉ **Correct tool output for functional test divZeroIndirectCall** No results
13. ◉ **Correct tool output for functional test negativeArrayAccessCall** No results

Coverage Values

1. ◉ **75% Branch Coverage** No results
2. ◉ **85% Line Coverage** No results
3. ◉ **80% Mutation Score** No results

Exercise details

Release date	Jun 11, 2025 10:00

About Feedback Releases Privacy Imprint