UNIVERSITÉ DU LUXEMBOURG

A Simple Abstract Interpreter

Lattice Theory for Parallel Programming

Goals

- ✯ Understanding the basics of abstract interpretation.
- ✯ Understanding how to analyzing the program.
- ✯ Implementing a simple abstract interpreter.
- ✯ This project accounts for 10% of the total grade.
- ✯ Don't share your code.
- ✯ This is a solo project.
- ✯ **Deadline**: Before the January exam.

1 Explanations

We provide a skeleton of the project at `https://github.com/ptal/lattice-theory-parallel-programming-lu/tree/main/4-Abstract_Interpreter`. The goal of this assignment is to verify the programs given in `tests`. Each program in `tests` comes with an additional challenge that helps you to improve your static analyzer incrementally. There are two kinds of "bugs" that you are tasked to verify:

- *Assertions*: Those are given by the user as `assert(condition);` and in the AST as a `POST_CON` node.
- *General properties*: Those are not explicitly given by the user and represent family of bugs such as *overflow* and *division-per-zero*.

It is useless to try to code directly the whole abstract interpreter. We proceed incrementally, only supporting program construction when the test program requires it. To statically evaluate a program, you are going to need three additional classes:

- *Abstract interpreter* with a function `eval(AST)` which traverse the AST and search for bugs.
- *Interval* which is a class modelling an interval $[\ell, u]$ where ℓ is a lower bound and u an upper bound. In this class, you will have a few methods for:
  - The lattice operations, especially the join operation.
  - Computing arithmetic and comparison operations (the $\mathbf{E}_I^\sharp[\![.]\!]$ and $\mathbf{C}_I^\sharp[\![.]\!]$ seen in class).

- *Interval Store* which is a class modelling a map $X \to I$ between variables and intervals. It relies on the interval class to perform arithmetic and comparison operations. In addition, it has a notion of variables. You can use `std::map<std::string, Interval>` to represent the map[1].

Until we manage loops, we do not need to write a fixpoint loop as shown on the slides "Abstract Least Fixpoint", because by simply traversing the AST once we directly reach the fixpoint. It means we do not even need to have a notion of *program locations*. Given the following program (`easy3.c`):

```
1  int a;
2  int b;
3  void main() {
4    /*!npk b between 0 and 1 */
5    if(b == 0) {
6      a = 1;
7    }
8    else {
9      a = 2;
10   }
11   assert(a <= 2);
12   assert(a >= 1);
13 }
```

Here are the steps performed by the abstract interpreter's `eval` on this example:

1. We start the static analyzer with the interval store at $\top$.
2. Once we reach the pre-condition `!npk b between 0 and 1`, we can restrict the value of `b` to be $[0, 1]$.
3. For the if statement, we must evaluate both branches separately using a copy of the interval store S_1 and S_2.
4. For the if-branch, we verify that `b == 0` can be true in the current interval store, and we enter the branch by restricting $b = [0, 0]$.
5. For the else-branch, we verify that `!(b == 0)` can be true in the current interval store, and we enter the branch by restricting $b = [1, 1]$.
6. In each branch, we restrict the value of a.
7. After evaluating both branches, we have two stores S_1 and S_2 and we must join them $S_1 \sqcup S_2$.
8. When we reach the assertion, we verify if it is satisfied.
9. If it is not satisfied, you print an error message and continue the evaluation.

Exercise 1 – Getting Started (8 points)

1. Follow the instructions in the `README.md` to compile the project and run it on a few examples we have provided in `tests`.
2. The program is represented by an *abstract syntax tree* (AST). If you don't know what it is, read `https://en.wikipedia.org/wiki/Abstract_syntax_tree` and the implementation in `ast.hpp`. For simplicity, we use only one kind of AST to represent both the expressions and the statements. Don't hesitate to write your own program to understand how they are represented by the AST. Note that we only support a very small subset of C and some instructions are desugared directly in the parser. For instance, `i++` is transformed into `i = i + 1;` so we have less constructions to manage during the static analysis.

[1] In practical implementation, we use `std::vector<Interval>` and a mapping `std::map<std::string, int>` for the connection between the variable's name and the variable's index in the store, but you do not need to do that here.

3. The files `easy1.c` to `easy9.c` are testing assignment, arithmetic operation and comparison. Consider them one by one until you verify all the assertions.
4. The files `ifelse1.c` to `ifelse3.c` are testing the if construction. Consider them one by one until you verify all the assertions.

Exercise 2 – Testing General Properties (4 points)

1. Add to the abstract interpreter an "automatic test" to verify that all division are correct, in the sense that division-per-zero cannot happen. If, according to the abstract semantics, it can happen, you must raise a warning.
2. Same question with overflow. You can use `int64_t` to represent the bounds of the intervals (we suppose only 32-bits integers are used in the programs tested).

Exercise 3 – Equational Form (4 points)

1. Restructure the abstract interpreter (in a new file, keep the previous interpreter) to be in equational form as shown in the slides. You need to have one store of intervals per location in the program.
2. Implement an abstract fixpoint computation engine as shown in the slides.

Exercise 4 – Loop Construction (4 points)

1. Using the previous equational form, add support for the loop statement. Test your program on bounded loops.
2. Add widening operation to ensure termination. Test your program on unbounded loops.