SAE Project 2: Program Analysis

Project Description

Consider a class `PrinterArray` which represents an array of printers of fixed size (`numPrinters`):

```
public class PrinterArray {
     private int numPrinters;
     public PrinterArray(int numPrinters) {
          this.numPrinters = numPrinters;
     }
     public native void sendJob(int printerIndex);
}
```

Design and implement a program analyzer that checks Java programs for the absence of:

- Division by zero exceptions
- Calls to the method `sendJob` with an argument `printerIndex` out of bounds.

More precisely, given a Java program, the program analyzer verifies that the following conditions hold for all methods declared in the program:

- For any division operation, such as `x/y`, the divisor `y` is not 0.
- For any call of method `sendJob` on an object `PrinterArray` with index `printerIndex`, the index `printerIndex` is in the range `[0, numPrinters-1]`.

Your program analyzer (see template below) must take as input the name of a test class (`TEST_CLASS`). The last two lines of your program analyzer's output must be:

```
TEST_CLASS OUTCOME_DIV_ZERO
TEST_CLASS OUTCOME_OUT_OF_BOUNDS
```

where `OUTCOME_DIV_ZERO` is

- `NO_DIV_ZERO`, if the program is **guaranteed** to have no divisions by zero.
- `MAY_DIV_ZERO`, if the program **cannot be proven** to be divisions by zero free.

and `OUTCOME_OUT_OF_BOUNDS` is

- `NO_OUT_OF_BOUNDS`, if the program is **guaranteed** to have all calls to `sendJob` with indexes `printerIndex` in the range `[0, numPrinters-1]`.
- `MAY_OUT_OF_BOUNDS`, if the program **cannot be proven** to have all calls to `sendJob` with indexes `printerIndex` in the range `[0, numPrinters-1]`.

Example 1

```
public class Test1 {
     public static void foo() {
```

```
        PrinterArray pa = new PrinterArray(5);
        pa.sendJob(2);
    }
}
```

The output of your tool for this program should be:

```
Test1 NO_DIV_ZERO
Test1 NO_OUT_OF_BOUNDS
```

Example 2

```
public class Test2 {
    public static void bar(int n) {
        int a = 5/n;
        PrinterArray pa = new PrinterArray(10);
        if ((a >= 0) && (a < 10)) {
            pa.sendJob(a);
        }
    }
}
```

The output of your tool for this program should be:

```
Test2 MAY_DIV_ZERO
Test2 NO_OUT_OF_BOUNDS
```

Project structure

You can download the template here.

It contains java files that you can extend:

- `ch.ethz.sae.Analysis` - this class performs computation until a fixpoint is reached.
- `ch.ethz.sae.Verifier` - the main class. You check here if the two conditions above are satisfied.
- `Test1, Test2` - examples of applications your verifier should be able to reason about.

As the first step, compile and run the template. The output of the template is initially `MAY_DIV_ZERO` and `MAY_OUT_OF_BOUNDS`, for any test class, which is very imprecise. The goal of the project is to follow the comments in the code, such that the project becomes more precise, soundly outputing `NO_DIV_ZERO` or `NO_OUT_OF_BOUNDS` for test classes where we can **guarantee** it.

Running the code from the template:

- Install APRON
- See build.sh for instructions how to compile.
- See run.sh for instructions how to run from command line.
- There is an eclipse project included in the template. You can use it or create your own project.

Virtual machine

We provide a virtual machine where the whole environment and dependencies for the project are pre-installed. To use the virtual machine, first install: Virtual Box. Next, download the virtual machine: : Ubuntu.ova (3GB). Import the virtual machine and start it with VirtualBox. The username is `sae` and the password is also `sae`. The project folder is in `/home/sae/project`. The virtual machine has Eclipse installed and the project is already set up there. To start eclipse, open a terminal and run `eclipse`. You can also compile and run the project from the command line, using the scripts `build.sh` and `run.sh`, respectively. The `run.sh` script takes a test class name as an argument.

Language Fragment to handle

For this project, you will analyse a fragment of Jimple. This language contains only local integer variables and `PrinterArray` objects.

- Details about the Jimple language can be found here
- The language fragment to handle is:

Jimple Construct	**Meaning**
DefinitionStmt	Definition Statement: here, you only need to handle integer assignments to a local variable. That is, x = y, or x = 5 or x = EXPR, where EXPR is one of the 4 binary expressions below. That is, you need to be able to hande: y = x + 5 or y = x * z.
JMulExpr	Multiplication
JSubExpr	Substraction
JAddExpr	Addition
JDivExpr	Division
JIfStmt	Conditional Statement. You need to handle conditionals where the condition can be any of the binary boolean expressions below. These conditions can again only mention integer local variables or constants, for example: if (x > y) or if (x <= 4), etc.
JEqExpr	==
JGeExpr	>=
JGtExpr	>
JLeExpr	<=
JLtExpr	<
JNeExpr	!=

- Loops are also allowed in the programs. You need to make sure to identify loops so that widening is performed.
- Assignment of pointers of type `PrinterArray` are possible, e.g. `p = q` where `p` and `q` are of type reference. However, those are handled by the pointer analysis and you do not need to worry in the numerical analysis about these.

- When verifying if a method satisfies the conditions, you need to handle calls to the method `sendJob` and calls to the constructor `PrinterArray`.

APRON

APRON is a library for numerical abstract domains. You can find documentation about the APRON framework here. An example file of using APRON is here.

Soot

Your program analyzer is built using Soot, a framework for analyzing Java programs. You can learn more about Soot by reading its tutorial, survivor guide, and javadoc. You can find additional tutorials here. Your program analyzer uses Soot's pointer analysis to determine which variables may point to `PrinterArray` objects (see the `Verifier.java` file in your template).

Implementation tips

- It is enough to analyze each method of the control application separately (e.g. without propagating parameters from one method to another). That is, you only need to analyze one method at a time.
- You can assume the constructor `PrinterArray` takes as argument only positive integer constants (never local variables) less than 10000.
- You can assume that the constructor `PrinterArray` and `sendJob` are never called inside loops.
- It is enough to use the polyhedra domain that APRON provides (Polka) and analyze relations over the local integer variables.
- The methods of the control applications may contain loops and branches. You do not need to handle type casts or types other than the Java `int` type in your analysis.
- If you see an operation for which you are not precise - do not crash, but be less precise or go to top instead so that you remain sound.
- Only local variables need to be tracked for the numerical analysis (no global variables or object fields), but for the heap you need to use the existing pointer analysis of Soot. The template already contains the invocation of the pointer analysis. You just need to make sure to use that information for checking the property.

Deliverables

- The source code of the tool (it should compile with the standard javac compiler of Java 6). The main class must be `ch.ethz.sae.Verifier` as in the template.
- You cannot use any library other than soot-2.5 which we will test with. You need to use the transformers of the APRON polyhedra domain (Polka).
- Feel free to unit test your implemetation, but your program should not depend on JUnit to run. We will not pass JUnit to the classpath when testing.
- Your program will be run using OpenJDK Java 6 under Linux (keep in mind that Java 7 has a slightly different syntax). Also, pointer analysis runs faster under Java 6 than Java 7 (likely, due to fewer classes to analyze).
- There will be a time limit of 10 seconds to verify an application. Each application will consist of at most 10 methods.

- Project deadline **June 8th, 11:00am**

Grading:

- Evaluation of your tool on our own set of programs for which we know if they are valid or not.
- We will evaluate you depending on the correctness of your program and the precision of your solution. You will not get points if your program does not satisfy the requirements. If your solution is unsound (i.e. wrong - says that an unsafe code is safe), or imprecise (says unsafe for code, which is safe) we will penalize it in the points. We will penalize unsoundness much more than imprecision.

Project Assistance

For questions about the project requirements, send an email to the TA mailing list (sae-assistants@lists.inf.ethz.ch)