SoPA(Soot-based Pointer Analysis)

Members

刘俊豪: Add flow-sensitivity, context-sensitivity; Support various invokes.

肖有为: Add field-sensitivity; Tests; Write the report.

Directories

```
SOPA
├──code
|   └── Local testcases
├──src/main/java/sopa
|   └── Source code for our points-to analysis implementation
├──.gitignore
├──readme.md
└──...
```

Features

We implement an Anderson-style points-to analysis algorithm base on the SOOT framework.
The `MyTransform` class extends the SceneTransformer. It serves as the entrance to both the analysis algorithm and the exception handler.
Our points-to analysis algorithm is implemented as the class `Algorithm`, which extends the `ForwardFlowAnalysis` and works in a forward data-flow analysis manner.
Once any exception is thrown out during the analysis process, the `MyTransform.internalTransform` method catches it and calls `PessiAlg` class to handle it.
`PessiAlg` generates the most sound and preservative analysis solution for the given program, which is naive and so omitted here.

Our algorithm supports a flow-sensitive, field-sensitive, and context-sensitive points-to analysis.

We develop a data-flow analysis algorithm to support flow-sensitivity. The Semilattice element is designed to be a set,
where each element `<x, {id0,id1,...,idk}>` represents the possible object numbers for the value `x` in the Jimple IR.
When merging Semilattice elements, the object number sets for the same value take their concurrent set.
The transition function works as follows.

- `Benchmark.alloc(id)`: store the id for the following `new` operation.
- `Benchmark.test(tid, var)`: get the possible object numbers for `var` from the `inset`, add them to the `analysisResults`.
- `InvokeStmt`: process various invokes in different ways, as described in *Section Invokes*.
- `DefinitionStmt`: kill, gen, consider the field-sensitivity, as described in *Section Field-sensitivity*.
- `CastStmt`: similar as a `DefinittionStmt`.
- `ReturnStmt`: update a returnSet, which is used in Invoke processing.
- `IfStmt, GotoStmt, BreakpointStmt,NopStmt, SwitchStmt...`: ignore these statements.
- Otherwise: throw an exception for meeting unknown statements.

Field-sensitivity

For field-sensitivity, we put `<k.f, {}>` into the sets, where `k` is the allocated number for any object. For objects without an allocated number, we allocate a distinct negative number.

When solving `DefinitionStmt`, if the rhs is a FieldRef(`x.f`), we look up the possible object numbers (for example, 1 and 2) for `x`.
Then we loop up the possible numbers for `1.x` and `2.x`, and we put all of them into the Gen set.
On the other hand, if the rhs is a normal value (Local). We just put the possible numbers for it into the Gen set.

If the lhs is a FieldRef(`y.f`), we need to modify the elements involving all the possible numbers for `k.f`s, where `k`s are possible numbers for `y`. Similarly, if lhs is normal values, just consider its own element.

Other corner situations are omitted here.

Invoke

Now the algorithm can handle function calling without recursion partly.
Each time the given program calling a function, our algorithm will generate a new `Algorithm` Object to calculate the result of the called function, and use store `callstack` detect recursion.

The algorithm can handle `SpecialInvoke`, `Staticinvoke`, `VirtualInvoke` and `InterfaceInvoke`.

For `SpecialInvoke`, it is an ordinary private method in a class, and the algorithm handle it with `enterInvoke` method. `enterInvoke` will add the parameters and `%this` field for this method into the new `Algorithm` Object's `entrySet` .Also all other variables that may be used in the `SpecialInvoke` will be put into `entrySet`, and will put the merge the result of the `SpeacialInvoke` with `outset` to support context-sensitivity.

For `VirtualInvoke` and `InterfaceInvoke` , these two may call a function that is not declared in the current class but in its superclass or subclass, so method `enterVirtualInvoke` will be used to handle them. `enterVirtualInvoke` will find all the function that this Invoke expression will call to based on the object that the base object may point to. For each possible call do the same thing like `enterInvoke` and merge all the result.

For `StaticInvoke`, it will solved by `enterStaticInvoke`. Since `StaticInvoke` has no `%this` field, `enterStaticInvoke` will not process `%this` field and do other things like `enterInvoke`.

The methods that initiate Static variables would run before main method, so the algorithm try to find them out and process it before start analysing main method.

How to use it

Export jar package & Run with jar

```
mvn package
java -jar ./target/pta-1.0-SNAPSHOT-jar-with-dependencies.jar {src} {Package.ClassName}
```

Run directly

```
mvn clean && mvn compile && mvn exec:java "-Dexec.mainClass=sopa.MyPointerAnalysis" "-
Dexec.args={src} {Package.ClassName}"
```