Applied Static Analysis

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

I would like to thank Florian Kübler and Dominik Helm for creating this exercise.

If you have questions regarding OPAL don’t hesitate to ask them in the OPAL Gitter channel!

Fixed Point Computations Framework

You should use the provided template. That project is preconfigured to use the latest snapshot version of OPAL and already contains the initial configuration. The template for this exercise can be found at: https://github.com/stg-tud/apsa/tree/master/2019/7-ModularAnalyses/Exercise/template

If the provided template does not compile, you may have an outdated version of OPAL. Please, make sure that you have downloaded the latest snapshot of OPAL. The easiest way to do so is to call "sbt cleanCache" and "sbt cleanLocal". If this doesn't help go to the your local .ivy cache and delete all jars relates to OPAL– they can be found in the respective sub folder. After that, when you compile the template, you should see that OPAL downloads the latest versions from the SNAPSHOT repository!

> ⚠️ Always ensure that you use the latest snapshot version. You can clean the latest (snapshot) version that you have downloaded using the command `sbt cleanCache cleanLocal` in your project's root folder.

An integrated JavaDoc of the latest snapshot version of OPAL that spans all sub-projects can be found at: www.opal-project.de

For further details regarding the development of static analysis using OPAL see the OPAL tutorial.

Declared Methods

Note that OPAL distinguishes between `Method` s and `DeclaredMethod` s.
A `Method` represents an entity existing in the bytecode of the project. It can be either a concrete, abstract or native method.
In contrast to this, a `DeclaredMethod` represents a reference by the code, including library methods not present in the code.
Additionally, a DeclaredMethod puts the method into the context of a class, i.e. if a class inherits a method, there are two `DeclaredMethod` objects, one for the parent and one for the child class, but only one `Method` object for the concrete implementation found in the bytecode.

There is just one type of `Method` , but it may or may not contain code (`body`).

A `DeclaredMethod` may be one of the following:

- `DefinedMethod` : Is backed by a `Method` (its `definedMethod`) and a `declaringClass` (either the parent or the child class from the example above).
- `VirtualDeclaredMethod` : A reference for which no `Method` object exists, i.e. a method from a library not present in the project.
- `MultipleDefinedMethods` : Represents some corner cases the JVM may choose from several methods on invocations.

OPAL uses `Method` s whereever the concrete representation is needed (e.g. for the three-address code), whereas it uses `DeclaredMethod` s for the targets of calls.

For this exercise you may ignore `VirtualDeclaredMethod` and `MultipleDefinedMethods` .

Allocation Free Methods

Develop an **interprocedural** analysis using the fixed point computation framework which determines whether a method (`DeclaredMethod`) or its (transitive) callees may allocate any objects or arrays.
The respective property is already available within OPAL (`org.opalj.br.fpcf.properties.AllocationFreeness`).

These methods are of interesst when determining whether a method is mathematically pure.
Furthermore, methods without allocations may not lead to `OutOfMemory` exceptions.

You should use the property store to retrieve the callees of the method under analysis (see `org.opalj.br.fpcf.cg.properties.Callees#callSites`). Furthermore, you can also get the method's three-address code from the property store.
The template is configured in a way, such that the results for `Callees` and `TACAI` are already final.

You may ignore the allocation of implicitly thrown exceptions.

Method with allocations:

```java
public Object foo() {
  return new Object();
}
```

Method with allocations in callees:

```java
public void bar() {
  foo();
}
```

Method without allocations:

```java
public Object baz() {
  return null;
}
```

Method without allocations even in callees:

```java
public void foobar() {
  baz();
}
```

> To run the analysis against the provided test case, use `run -cp=<Path>/AllocationFreeness.class` .

Note that for historical reasons, OPAL uses a lattice's *top* value to denote the most precise value and the *bottom* value for the sound over-approximation.
Thus, the lattice order is reversed in comparison to the lecture.
For an `EOptionP`, `ub` is the current, precise approximation.

Tasks

1. Test your analysis using the class `AllocationFreeness.class`. It should not produce any false positives.
2. Run your analysis against the JDK.
3. (optional) Also handle allocations of implicitly thrown exceptions. No solution will be provided.

Hints

Please try to solve the exercise before taking a look at the hints.

1. To retrieve the method's three-address code, use `ps(method.definedMethod, TACAI.key)`. Note that the property is already final, thus you can use `asFinal.p` to retrieve the property value.

2. Try to return final results as soon as possible.

3. To retrieve the method's callees, use `ps(method, Callees.key)`.

4. When retrieving the `AllocationFreeness` of another method, it might be the case that there is not even an intermediate value available yet. Thus, remember to handle `EPK`s.

5. Take a look at the `unapply` methods in `EOptionP`. They can be used in pattern matching of `EOptionP`s or `EPS`s.

6. For dependency management you should use a `Set[EOptionP[DeclaredMethod, AllocationFreeness]]`.

7. When receiving an update for a dependency that is still not final, the respective `EOptionP` must be replaced in the set of dependencies.

8. The set of dependencies must never contain final values.