The Static Analysis Framework OPAL

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

Introduction

OPAL is a bytecode…

- **analysis**
- transformation
- generation

platform written in Scala.

In this tutorial we are primary concerned with Bytecode analysis.

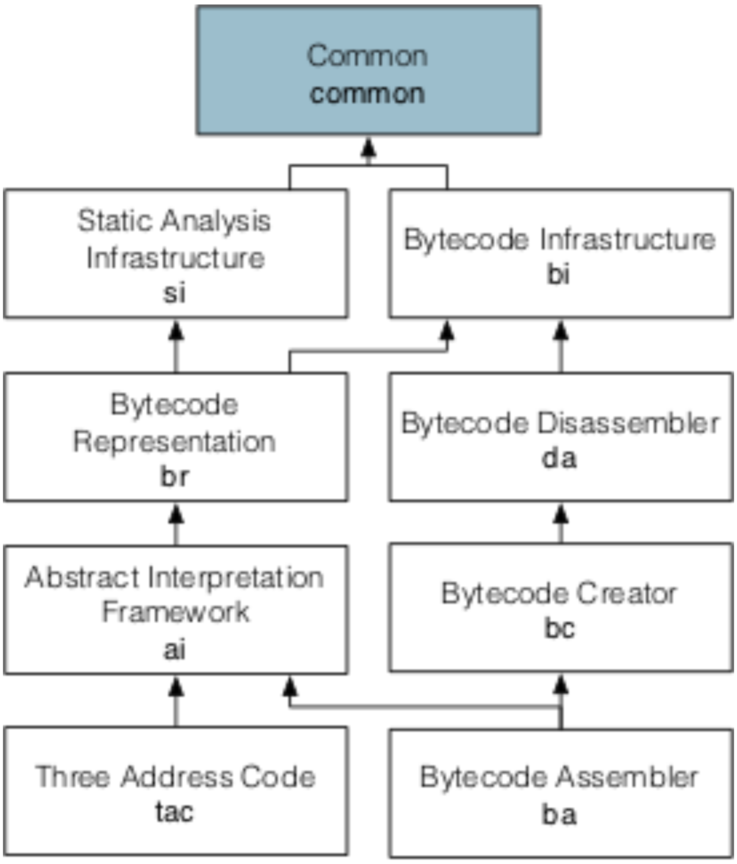
Requirements

- Java 8
- Scala 2.12.8
- The build tool sbt

Java 8 is in particular required if you want to use Visual Studio Code for writing your analysis. Do not forget to install the excellent Scala Metals plug-in.

Scala is actually not strictly required, because it will be download by sbt if required. However, to do some Scala hacking and prototyping, a running Scala installation is helpful.

Basic Components



- Common contains general algorithms and data-structures that are not specific to (Java (Bytecode)) analyses. E.g., logging functionality, parallelization related functionality, specialized data-structures.
- Static Analysis Infrastructure is a framework to facilitate the development of modular, parallelized static analyses. It is also not specific to (Java (Bytecode)) analysis.
- Bytecode Infrastructure is a generic framework to efficiently parse Java Bytecode.
- Bytecode Disassembler provides a 1:1 object-oriented representation of a Java `.class` file. It is primarily useful when a precise understanding of every single byte of a class file is required.
- The Bytecode Creator framework provides the functionality to serialize a class file in the representation provided by the Bytecode Disassembler to a byte array; the bytecode creator framework is used to create/manipulate class files.
- The Bytecode Assembler framework provides a small eDSL to generate Java Bytecode. The target of the assembler is the representation provided by the Bytecode Disassembler which is then serialized using the Bytecode Creator framework.
- The Bytecode Representation framework provides a high-level object-oriented representation of Java Bytecode which is well suited for simple analyses which do not require the tracking of data-flow information.
    - The constant-pool is resolved to facilitate pattern matching
    - Standard Java and Scala Invokedynamic instructions are rewritten
    - Control-flows are normalized

- Abstract Interpretation Framework is a very lightweight framework for the abstract interpretation; currently it is primarily useful for intra-procedural analyses.

- The three-address code framework provides a high level register-based intermediate representation. It provides two basic representations:
  - TACNaive which is an untyped three-address code like representation of Java bytecode
  - TACAI which is typed, SSA-like three-address code representation. This is the primary representation used by analysis.

Analysis Template

```
object Main extends DefaultOneStepAnalysis {

  def doAnalyze(
        p: Project[URL], // The main entry point
    params: Seq[String], // App-specific command-line parameters
    isInterrupted: () ⇒ Boolean // Called test if the analysis should be aborted
  ): BasicReport = {

    // Here goes the analysis...

    "Done"
  }

}
```

You can get the template project using GIT:
`git clone --depth 1 git@bitbucket.org:OPAL-Project/myopalproject.git Project`

The default template predefines several parameters which affect the instantiation of the `Project`. Most notably, `-cp=<Folder or Jar File which contains the class files you want to analyze>`.

The template can directly be started using `sbt run`.

org.opalj.br.analyses.Project

- Primary representation of a Java project:
    - Defines methods to access a project’s class files, methods and fields.
    - Access to a project’s class hierarchy.
    - Provides functionality to resolve method calls/field accesses.
    - Central entry point to get further project-wide information on demand.

An instance of a `org.opalj.br.analyses.Project` represents the concrete project that will be analyzed. OPAL distinguishes between the code that belongs to the project and the project’s libraries. Depending on the requirements of the analysis libraries can completely be loaded or just the public interface. In the later case, the method bodies are omitted which safe some memory and speeds up the time required to load the classes.

In real projects – in particular when libraries are analyzed - it is often practically unavoidable that the class hierarchs is not complete (e.g., the JDK references classes belonging to `org.eclipse` which are not part of the JDK). OPAL provides extensive support to handle such situations, but it is nevertheless highly recommended to analyze projects which are complete.

OPAL contains growing support for deliberately broken projects to enable the analysis of heavily obfuscated projects.

Iterating over a project’s source elements

Using For-comprehensions

```
val p : SomeProject = ...
val publicMethods = for {
    classFile ← p.allClassFiles//.par
    method ← classFile.methods
    if method.isPublic
  } yield {
    method.toJava
  }
```

To only iterate over the *project’s* class files use `allProjectClassFiles`.

In many cases parallelization can be achieved in two different ways:
- using Scala’s parallel collections
- using OPAL’s native parallelization methods

Iterating over a project’s source elements

Using higher-order functions

```
val r = new java.util.concurrent.ConcurrentLinkedQueue[<ResultType>]()
p.parForeachMethodWithBody(isInterrupted) { mi ⇒
    val m = mi.method
}
import scala.collection.JavaConverters._
r.asScala.mkString("\n")
```

In general, the parallelization provided by OPAL is more efficient because it uses domain specific information to optimize the parallel execution. E.g., OPAL’s `parForeachMethodBody` processes all methods in parallel starting with the longest method(s). This way the parallelization level can be increased; e.g., a random processing order could lead to the situation that the longest (most complex) method os scheduled to be analyzed last. In this case the overall analysis time is then heavily influenced by the time required to analyze that method.

Iterating over the instructions of a method

The most efficient way to iterate over the body of a method is to use on of the respective methods provided by `Code`:

```
m.body.get.collect {
    case i @ INVOKEVIRTUAL(
            ObjectType.Object,
            "toString",
            MethodDescriptor.JustReturnsString
        ) ⇒ i
}
```

Code provides a variety of methods that should suite most needs.

It is also possible to iterate over the underlying instructions array. However, in most cases it is more efficient to use the `iterator` method provided by the `Code` object if you really want to use a for-comprehension. `for {i <- code.iterator}{...}`.

None of the methods is parallelized because in the very vast majority of cases the effort to parallelize the execution far outweighs the performance gains.

CFG for Java Bytecode

Getting the CFG is trivial:

```
iomport org.opalj.br.cfg.CFGFactory
val cfg = CFGFactory(m : Method, classHierarchy : ClassHierarchy)

// A rudimentary class hierarchy is always available:
ClassHierarchy.PreInitializedClassHierarchy
```

The class hierarchy is required to correctly resolve exceptions. If the class hierarchy is not complete, it may happen that a control-flow edge is created to an exception handler that will never handle the respective exception at runtime.

Use the `PreInitializedClassHierarchy` only for testing purposes!

Given the cfg it is then possible to, e.g., iterate over all blocks or to traverse the cfg:

```
val cfg = CFGFactory(m : Method, classHierarchy : ClassHierarchy)
cfg.allBBs // to iterate over all blocks in lexical order

cfg.startBlock // ... the initial start block
```

CFG for Java Bytecode

In OPAL the CFG has four types of nodes:

- (standard) basic block
- exit nodes:
    - normal return node
    - abnormal return node
- catch node

The first block may contain predecessors!

A catch node does not have any instructions.

Performance Measurements

Using ProjectInformationKeys

`ProjectInformationKey` objects are used to get/associate some (immutable) information with a project that should be computed on demand.

When a concrete `ProjectInformationKey` is passed to the `Project` the respective analysis is triggered and the result is cached.

- Notable `ProjectInformationKey`s:
  - `ProjectIndexKey`
  - `` `*TACAIKey ``
  - `StringConstantsInformationkey`
  - `FieldAccessInformationKey`

The project index key enables a reverse lookup of methods and fields given the name (and descriptor) of a method.
The different TACAIKeys provide access to three-address code representation provided by OPAL. A specific analyses should always use only (at most) one of these keys.
The `StringConstantsInformationkey` is an index of all strings found in the source code.
The `FieldAccessInformationKey` provides the information about where a field is read/write accessed. Field accesses via reflection or `Unsafe` are not reflected.

> Analyses that compute information for `ProjectInformationKey`s are expected to be internally parallelized.

Global configuration

- Which kind of messages are logged
- Which kind of transformations are performed
    - Control-flow simplification
    - Resolution of `invokedynamics`

Project specific configuration

- Typesafe config based
- Each project has its own configuration
- The configuration can be adapted when a project is created

OPAL uses Typesafe Config for managing its configuration. That is, the default configuration is stored in the `reference.conf` files which are part of OPAL. To adapt the configuration to specific needs it is possible to specify an application.conf file. That file will overwrite the defaults. Alternatively, when a project is created it is possible to adapt the configuration to specific needs.

Getting the 3-Address Code (TAC(AI))

CFG for Three Address Code