Introduction to the OPAL Framework

Applied Static Analysis

Dr. Michael Eichberg (Organizer)
Johannes Lerch, Ben Hermann, Sebastian Proksch, Karim Ali Ph.D.

Software Technology Group

TECHNISCHE UNIVERSITÄT DARMSTADT

Architecture

Abstract Interpretation Framework

Analyses that are concerned with a program’s control and data-flow (and which provides reasonable abstractions thereof)

“resolved” Bytecode Representation

Object-oriented representation of Java class files

Well suited for lightweight static analyses (e.g., to analyze an application’s static structure, to find instances of bug patterns.)

Bytecode Infrastructure

Generic infrastructure for parsing Java class files.

The Bytecode Representation

Abstract Interpretation Framework

Analyses that are concerned with a program’s control and data-flow (and which provides reasonable abstractions thereof)

“resolved” Bytecode Representation

Lightweight static analyses (e.g., to analyze an application’s static structure, to find instances of bug patterns.)

Bytecode Infrastructure

Low-level “analyses” directly on top the bytecode representation (e.g., to do bytecode verification, to create other representations of Java bytecode, to find instances of bug patterns (sequence of bytecode instructions).)

The package: `org.opalj.br`

- Classes used for the representation of Java class files.
  Basically, each major element of a class file (Method Declaration, Field Declaration, each “Attribute”,…) is represented using one class.
  (The class hierarchy as well as method and field names are well-aligned with the JVM specification.)

- By default all information is represented.
  (Except of the BootstrapMethodTable, which is resolved; the information is directly attached to the respective Invokedynamic instructions.)

- Traversal of class files is primarily facilitated by Pattern Matching.
  (Signatures also support the traversal using a Visitor)

The package: `org.opalj.br`

- Main classes:
  - `ClassFile`
    (Recall that the class `java.lang.Object` does not have a superclass.)
    - `Method`
      (Representation of a Method)
      - `MethodDescriptor`
        (Parameter types and return type.)
      - `Code`
        (If the method is non-native and non-abstract.)
        The instructions array may contain “null” values.
        Defines many helper methods to facilitate the traversal of the instructions array.
    - `Field`

The package: `org.opalj.br`

- Main classes:
  - `Type`
    Types can and should be compared using reference comparison (**`eq`**).
    All types are associated with a unique id.
    - `ReferenceType`
      - `ObjectType`
        The name uses the JVM's binary notation.
        (I.e., `java/lang/Object` and not ~~`java.lang.Object`~~)
        Common ObjectTypes are predefined.
      - `ArrayType`

The package: `org.opalj.br.instructions`

- Representation of a method's instructions.
  The *type hierarchy* and many *extractors* facilitate the matching of byte code sequences/facilitate the abstraction over different types of byte code instructions.

- To match sequences study the class "Code" as a foundation.

Example - Matching Instructions
The package: org.opalj.br.instructions

```
val jre = "/Library/Java/JavaVirtualMachines/jdk1.8.0_77.jdk/Contents/Home/jre/lib"
val cfs = org.opalj.br.reader.Java8Framework.ClassFiles(new java.io.File(jre))

for {
     classFile ← cfs.map(_._1)
     method @ org.opalj.br.MethodWithBody(body) ← classFile.methods
     method ← body.collectFirstWithIndex {
         case (
             pc,
             org.opalj.br.instructions.INVOKESPECIAL(
               org.opalj.br.ObjectType.Object,
               "<init>",
               org.opalj.br.MethodDescriptor(Seq(),org.opalj.br.VoidType)
             )
         ) ⇒ method
    }
 } yield { method }
```

Finds all methods that call the default constructor of java.lang.Object.

The package: org.opalj.br.reader

- Classes to read in Java Class Files. Main classes:
  - `Java8Framework`
    Represents “all” information.
    (Attributes that are not supported are discarded.)
  - `Java8LibraryFramework`
    Only represents the information that describes the public interface.
- In most cases, however, it is more useful to load class files (implicitly) using a `Project`.

The package: [org.opalj.br.analyses]

- Commonly useful analyses or classes that support the development of analyses.

- `Project`
  Primary abstraction of a Java Project.
  Serves as a container for global information (e.g. the call graph, the class hierarchy).
  Enables the navigation from `Methods`, `Fields` and `ObjectTypes` to `ClassFiles`.

- `ClassHierarchy`
  Representation of the class hierarchy. Support methods to calculate least upper type bounds and to resolve method and field references. Supports partial class hierarchies.

- `(DefaultOneStep)AnalysisExecutor`
  Template class that facilitates the development of new analyses.

Adhoc Analyses
Prototyping Analyses

Using the Scala REPL

The Scala REPL

OPAL is easy to explore using Scala's REPL
OPAL facilitates prototyping (parts of) analyses using the REPL

Get the number of abstract methods defined in the rt.jar

```
pc-eichberg:OPAL eichberg$ sbt
[info] Set current project to OPAL Library (in build file:/Users/eichberg/
Code/OPAL/)
> project OPAL-DeveloperTools
[info] Set current project to Bytecode Representation (in build file:/
Users/eichberg/Code/OPAL/)
> console
[info] Starting scala interpreter...

scala> val cfs = org.opalj.br.reader.Java8Framework.ClassFiles(new
java.io.File("/Library/Java/JavaVirtualMachines/jdk1.8.0_77.jdk/Contents/
Home/jre/lib/rt.jar"))
cfs: Seq[(org.opalj.br.reader.Java8Framework.ClassFile, java.net.URL)]
= ...
scala> cfs.size
res0: Int = XXXXX
scala> cfs.map(cf => cf._1.methods.filter(_.isAbstract).size).sum
res3: Int = XXXXX
```

Get the number of abstract methods defined in abstract classes (not interfaces).

```
scala> val cfs = org.opalj.br.reader.Java8Framework.ClassFiles(new java.io.File("/Library/Java/JavaVirtualMachines/jdk1.8.0_77.jdk/Contents/Home/jre/lib/rt.jar"))

scala> cfs.filter(cf => !cf._1.isInterfaceDeclaration).map(cf => cf._1.methods.filter(_.isAbstract).size).sum
res3: Int = XXXX
```

Get all classes that extend `java.util.AbstractList` (including anonymous classes.)

```
scala> val p = org.opalj.br.analyses.Project(new java.io.File("/Library/Java/JavaVirtualMachines/jdk1.8.0_77.jdk/Contents/Home/jre/lib/rt.jar"))
p: org.opalj.br.analyses.Project[java.net.URL] =
Project(…

scala> p.classHierarchy.allSubtypes(org.opalj.br.ObjectType("java/util/AbstractList"),false).filter(_.packageName.startsWith("java/")).mkString("\n")
res0: String =
ObjectType(java/util/Collections$EmptyList)
…

scala>
```

AnalysisExecutor
The package: org.opalj.br.analyses

```
package org.opalj.ai.tutorial.base

import java.net.URL
import org.opalj._
import org.opalj.br._
import org.opalj.br.analyses._
import org.opalj.ai._

object <AnalysisTemplate> extends DefaultOneStepAnalysis {

    override def doAnalyze(
        theProject: Project[URL],
        parameters: Seq[String],
        isInterrupted: () ⇒ Boolean) = {

        // HERE GOES YOUR ANALYSIS


        BasicReport(theProject.statistics.mkString("\n"))
    }
}
```

The Bytecode Representation

- General Design Decisions

  - All classes are (effectively) immutable.
    (Supports parallelization.)

  - Extensive support for pattern matching.
    If available, *always* use the accessor methods offered by the class that manages the collection and not the collection itself.
    E.g., to find a method with a specific name, use the respective method defined by `Method` and do not use the collection's `find` method; to iterate over the instructions of a method use code's respective methods. The explicitly defined methods generally offer additional functionality or are more efficient due to domain knowledge.

  - There are no "`null`" values.
    (There is *only one exception*: the instructions array)

Example - all clone methods which call super.clone()
The Bytecode Representation

```
import org.opalj.br._
import org.opalj.br.instructions._

for {
    classFile ← classFiles
    superClass ← classFile.superclassType.toList
    ms = classFile.methods
    method @ Method(_, "clone", MethodDescriptor(Seq(), ObjectType.Object)) ← ms
    if method.body.isDefined
    if !method.body.get.instructions.exists {
        case INVOKESPECIAL(
            `superClass`,"clone", MethodDescriptor(Seq(), ObjectType.Object)
             ) ⇒ true;
        case _ ⇒ false;
    }
} yield (classFile /*.thisClass.className*/ , method /*.name*/ )
```

Inspired by a FindBugs analysis.

Example

- An analysis that collects all constructor calls in all methods that create an instance of `java.io.File` using the constructor (`File(String s)`).

  - The analysis returns the triple:

    ```
    ( /*Declaring Class*/ ClassFile,
      /*Caller*/ Method,
      /*PCs of constructor calls*/ Set[PC])
    ```

    I.e., for each method the set of program counters that call the respective constructor is returned.

  - The analysis in executed in parallel.

All calls of the constructor `File(String s)`.

```
val callSites = for {
   cf <- p.allClassFiles.par
    m @ MethodWithBody(body) <- cf.methods
    pcs = body.collectWithIndex {
          case (
              pc,
              INVOKESPECIAL(
                  ObjectType("java/io/File"),
                  "<init>",
                  SingleArgumentMethodDescriptor((ObjectType.String, VoidType)))
              ) => pc
      }
    if pcs.nonEmpty
  } yield (cf, m, pcs)
```

All calls of the constructor `File(String s)`.

```
import java.net.URL
import org.opalj._
import org.opalj.br._
import org.opalj.br.analyses._
import org.opalj.br.instructions._
import org.opalj.ai._

object IdentifyResourcesAnalysis extends DefaultOneStepAnalysis {

        override def title: String = "File Object Creation Using Constant Strings"

        override def description: String =
            "Identifies java.io.File object instantiations using constant strings."

        override def doAnalyze(theProject: Project[URL], parameters: Seq[String],
                               isInterrupted: () ⇒ Boolean) = {
            val callSites = for {
               cf <- p.allClassFiles.par
                m @ MethodWithBody(body) <- cf.methods
                pcs = body.collectWithIndex {
                      case (
                          pc,
                          INVOKESPECIAL(
                              ObjectType("java/io/File"),
                              "<init>",
                              SingleArgumentMethodDescriptor((ObjectType.String, VoidType))
                          ) => pc
                   }
                if pcs.nonEmpty
              } yield (cf, m, pcs)

            BasicReport(callSites.map(cs => cs._2.toJava(cs._1)).mkString)
}   }
```

Exercise 1

- Find all simple conditional jump instructions (`if_icmpXX`, `ifXX`) instructions in the JDK that are useless.
(E.g.,
`if(x < 100) { /* we wanted to do something smart, but we forgot it.. */ }`
)

- First clone OPAL (http://bitbucket.org/delors/opal)

- Install SBT (http://www.scala-sbt.org)

- Start the sbt console (using "`sbt`" in OPAL's root folder)

- Change the project: `project OPAL-DeveloperTools`

- Start the scala console: `console`

Architecture

Abstract Interpretation Framework

Analyses that are concerned with a program's control and data-flow (and which provides reasonable abstractions thereof)

"resolved" Bytecode Representation

Object-oriented representation of Java class files

Well suited for lightweight static analyses (e.g., to analyze an application's static structure, to find instances of bug patterns.)

Bytecode Infrastructure

Generic infrastructure for parsing Java class files.

The Abstract Interpretation Framework

- Static analysis framework that is inspired by Model Checking/ Symbolic Execution and **Abstract Interpretation**
- Works directly on top of Java bytecode to reduce overhead
  (I.e., the AI framework does not use an intermediate representation!)
- Complexity of Java bytecode is completely hidden within OPAL
- "Functional Style" to facilitate parallelization/to avoid hard to find bugs (most objects cannot be mutated and those that can be mutated are not expected to be mutated)
- User defined precision (no framework inherent limitations)

Four Different Approaches to Develop Analyses Using the AI Framework can be Distinguished

- by reusing a pre-configured `Domain` to pre-analyze a method and then analyzing the result

- by configuring a `Domain` w.r.t. the analysis task at hand
  (By means of mixin-composition of existing partial domains.)

- by adapting/configuring a `Domain`
  (By means of subclassing a `Domain`.)

- by developing a new `Domain`

A simple dead-code analysis.

…by reusing a pre-configured Domain to pre-analyze a method and then analyzing the result.

E.g. we want to find code such as:

if(x){…} else {…} where we know that x is always either true or false.

A Simple Domain that can be used, e.g., by a dead-code analysis.

```
class AnalysisDomain(
    override val project: Project[java.net.URL],
    val method: Method)
      extends Domain
      with domain.DefaultDomainValueBinding
      with domain.ThrowAllPotentialExceptionsConfiguration
      with domain.l0.DefaultTypeLevelFloatValues
      with domain.l0.DefaultTypeLevelDoubleValues
      with domain.l0.TypeLevelFieldAccessInstructions
      with domain.l0.TypeLevelInvokeInstructions
      with domain.l1.DefaultReferenceValuesBinding
      with domain.l1.DefaultIntegerRangeValues
      with domain.l1.DefaultLongValues
      with domain.l1.LongValuesShiftOperators
      with domain.l1.ConcretePrimitiveValuesConversions
      with domain.DefaultHandlingOfMethodResults
      with domain.IgnoreSynchronization
      with domain.TheProject
      with domain.TheMethod
      with domain.ProjectBasedClassHierarchy
```

A simple dead-code analysis.

```
val results = new java.util.concurrent.ConcurrentLinkedQueue[DeadCode]()
for {
  classFile ← theProject.classFiles.par
  method @ MethodWithBody(code) ← classFile.methods

  result = BaseAI(classFile, method, new AnalysisDomain(theProject, method))

  operandsArray = result.operandsArray

  (ctiPC, instruction, branchTargetPCs) ← code collectWithIndex {
    case (ctiPC, instruction @ ConditionalControlTransferInstruction())
         if operandsArray(ctiPC) != null /* the if is not dead */ ⇒
      (ctiPC, instruction, instruction.nextInstructions(ctiPC, code))
  }
  branchTarget ← branchTargetPCs
  if operandsArray(branchTarget) == null /* this target is dead */
} {
  val operands = operandsArray(ctiPC).take(2) /* get info about the relevant values*/
  results.add(
    DeadCode(classFile, method, ctiPC, code.lineNumber(ctiPC), instruction, operands)
  )
}
```

executed in parallel

thread safe!

Exercise 2

- Find all defs without a use.
  (I.e., find all instructions which push a new value onto the stack, but where the value is not used (in a meaningful manner)).
- Hint: use a *domain* that records the def-use information.
- Run your analysis on top of the JDK.