Nullness Analysis of Java Bytecode via Supercompilation over Abstract Values*

Ilya G. Klyuchnikov

JetBrains; Keldysh Institute of Applied Mathematics of RAS

Abstract. Code inspections in the upcoming release of IntelliJ IDEA take into account how binary Java libraries used in a project deal with null references. For this purpose Java libraries are annotated with results of nullness analysis under the hood. The paper reveals one of several non-trivial technical aspects of nullness analysis of Java binaries performed by IDEA: supercompilation over abstract values. A case study project *Kanva-micro* – a tool for inference of `@NotNull` method parameters – is used to illustrate this aspect step-by-step. A method parameter is annotated by *Kanva-micro* as `@NotNull` if the method cannot complete normally when `null` is passed to this parameter. The source code of *Kanva-micro*'s core is provided and explained in details. The paper may also serve as a tutorial on using supercompilation methods for program analysis.

1 Introduction

This paper starts a series of tutorial papers explaining details of how nullness analysis of Java bytecode is implemented in the upcoming IntelliJ IDEA 14 release. The papers are organized around two self-sufficient ready-to-run tutorial projects:

- The *Kanva-micro* project [8] focuses on the essence of the method (supercompilation over abstract values) at the cost of simplifications.
- The *Faba* project [4] is about how this method may be implemented in a production system.

The current paper describes the *Kanva-micro* project step-by-step.

1.1 Nulls in Java, richer type systems and the problem of interoperability

The majority of mainstream programming languages (including Java programming language) allow null references (`null` in Java). Dereference of `null` results into a runtime error. Tony Hoare, the creator of `null`, has stated that the `null` was "the billion dollar mistake" in the language design.

* Supported by Russian Foundation for Basic Research grant No. 12-01-00972-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

Ironically, compile-time checks performed by compilers of statically typed languages are to guarantee the absence of runtime errors if code compiles, and null pointer errors are not covered by these checks in major mainstream languages.

Dereference of `null` results into `NullPoiterException` in Java.

There are practical alternatives to enjoy null safety when programming for JVM:

- Migrate to alternative JVM language with nullable types such as Kotlin [9] or Ceylon [2].
- Enrich Java code with nullity annotations like `@NotNull` and `@Nullable` and use additional tools to check such annotations. There are several tools understanding nullity annotations: Eclipse Java compiler with additional null analysis [3], null analysis inspections in IntelliJ IDEA [6], the Checker framework [10].

Anyway, a Java programmer enjoys null safety only when working in a new richer type system. When there is a need to use existing Java libraries, the problem of `null` references appears on the boundary of two worlds. There is no clear practical way to make use of Java libraries *completely* null safe. However, it is possible to infer *some* nullity information automatically to make use of Java libraries safer.

1.2 The task

Some null analysis tools, including IntelliJ IDEA and the Checker framework, allow to use external annotations that are stored separately from library bytecode. The Kotlin compiler allows to specify external annotations as well. So inference of such annotations is of practical usage.

Kanva-micro [8] focuses on inference of `@NotNull` annotations for method parameters. A method parameter is annotated as `@NotNull` if *in any situation* when `null` is passed to this parameter the method cannot complete normally. Practically, it means that an author of this library method doesn't expect `null` to be passed to this parameter. From a client's point of view this is the same as the author put explicit `@NotNull` annotation in original source code.

The task of *Kanva-micro* is to automatically infer such external annotations for Java libraries. The phrase "*in any situation*" in the previous paragraph is significant: inferred annotations should not forbid to use a library. A `@NotNull` annotation is considered incorrect if under *some* conditions when `null` is passed to this parameter, the method completes normally.

There are no problems with the following `@NotNull` annotation: all possible executions result in `NullPointerException` when `view` is `null`.

```
1  void loadConfig(@NotNull View view) {
2    File f = getConfigFile();
3    if (f != null) {
4        view.loadConfigFromFile(f);
5    } else {
```

```
6         view.loadDefaultConfig();
7     }
8   }
```

However, the next `@NotNull` annotation is incorrect because this method completes normally when `getConfigFile()` returns `null` and `view` is `null`.

```
1   void saveConfig(@NotNull View view) {
2       File f = getConfigFile();
3       if (f != null) {
4           view.saveConfigToFile(f);
5       }
6   }
```

1.3 Handling of asserts

Many real-world Java libraries check that parameter is not `null` in the following way:

```
1   if (array == null) {
2       throw new IllegalArgumentException("array␣is␣null");
3   }
```

Sometimes such checks may be implicit:

```
1   if (!(object instanceof Serializable)) {
2       throw new IllegalArgumentException("object␣is␣not␣Serializable");
3   }
```

So to handle all these checks as well and infer `@NotNull` for corresponding parameters is desirable.

1.4 Inference of a subset of `@NotNull` annotations

The task to infer *all* correct `@NotNull` annotations in general case is undecidable. The root cause of undecidability of this task is undecidability of the subtask to detect whether a certain execution path in a program is reachable.

So, the practical task is to infer as much correct `@NotNull` annotations for method parameters as possible.

Kanva-micro assumes that all branches of conditionals that do not directly depend on nullity of parameters are reachable. Under this assumption annotations inferred by *Kanva-micro* are sound (since a superset of all possible execution paths is considered). But some annotations are lost, so *Kanva-micro* infers a subset of all correct annotations.

1.5 Algorithm presentation

Kanva-micro relies heavily on ASM library which is "an all purpose Java bytecode manipulation and analysis framework" [1, 12]. ASM library is de-facto a standard tool for Java bytecode processing in many production projects. The main part of *Kanva-micro* is quite high-level, since all low-level boilerpate may be gracefully delegated to ASM library. It turns out that it is clearer and easier to present the technical details of *Kanva-micro* in well-established ASM terms rather than creating a special complicated formalism for simple things. So *all* technical details of the core logic of *Kanva-micro* are described just in listings.

1.6 Outline

Section 2 describes the core ingredients of the algorithm and also provides a necessary background about Java bytecode internals, section 3 goes into main technical details of the implementation, section 4 discusses experimental results of annotating some Java libraries, the cost of simplifications made in *Kanva-micro* and sketches how *Faba* overcomes these simplifications and section 5 mentions related work.

2 Algorithm

Kanva-micro performs intra-procedural analysis: each method is analyzed separately. Also *Kanva-micro* considers a general case: a Java library may be already partially annotated.

The technical side of the problem can be easily formulated without going into all subtle details of JVM semantics (these details are easily abstracted away). This section deals with Java bytecode from the point of view of nullness analysis and describes the algorithm of *Kanva-micro* informally.

2.1 A crash course of Java bytecode

Java source code comprises of a set of classes, each class in turn comprises of a set of fields and methods. A method in some sense is a "unit of execution". Java virtual machine is a stack virtual machine. A chain of method invocations is organized traditionally via call stack (composed of frames). Execution of a method is associated with a frame which has a storage for variables defined in this methods and also an operand stack. On a method invocation a current frame is pushed on the call stack (with a return address), a new frame is created and initialized with respect to passed arguments, a control is passed to instructions of called method. When execution of the method is completed, the previous frame is popped from the stack, a return value is put on the operand stack and control is transferred to restored return address.

There are more than 200 Java bytecode instructions, however, only a relatively small subset of them may cause null pointer error. Here is a list of such bytecode instructions with descriptions when there may be a `NPE` (`NullPointerException`) during execution of this instruction.

- `GETFIELD`, `PUTFIELD` – load/store a field of an object. Error if the corresponding object (an owner of the field) is `null`.
- `ARRAYLENGTH` – get the length of an array. Error if the corresponding array is `null`.
- `IALOAD`, `LALOAD`, `FALOAD`, `DALOAD`, `AALOAD`, `BALOAD`, `CALOAD`, `SALOAD` – load an int, long, float, double, reference, boolean, char, short value from an array. Error if the corresponding array is `null`.

- `IASTORE`, `LASTORE`, `FASTORE`, `DASTORE`, `AASTORE`, `BASTORE`, `CASTORE`, `SASTORE` – store an int, long, float, double, reference, boolean, char, short value in an array. Error if the corresponding array is `null`.
- `MONITORENTER` – synchronization by entering monitor of an object. Error if the corresponding object (monitor's owner) is `null`.
- `INVOKESTATIC` – call of a static class method (with arguments). Error if a `null` argument is passed into a parameter annotated as `@NotNull`.
- `INVOKEVIRTUAL`, `INVOKESPECIAL`, `INVOKEINTERFACE` – call of an instance method of an object. Error if the corresponding object is `null` or a `null` argument is passed into a parameter annotated as `@NotNull`.
- `ATHROW` – throw an exception. The case when an exceptions is thrown as a result of comparison of a parameters with `null` corresponds to assertions.

2.2 The core idea - inspection of process graph

Kanva-micro performs analysis for each method parameter of a reference type. Thus, for a method with three reference parameters three independent analyses will be run. The simple core idea is to consider *all* possible executions paths inside the method assuming that a parameter of interest is `null`. If *none* of these executions paths completes normally, the parameter is annotated as `@NotNull`. We are going to build a process tree [13] and inspect each branch of this tree for errors caused by `null` value of the parameter of interest. *Kanva-micro* doesn't try to build a perfect process tree – trees built by *Kanva-micro* may have unreachable branches. However, the existence of such branches doesn't affect the soundness of inference, but simplifies inference a lot.

It is sufficient to abstract away concrete values of local variables and operands in the operand stack in the current frame (corresponding to execution of a method being analyzed) and consider just two abstract values:

- `ParamValue` – a value passed into parameter of interest.
- `BasicValue` – any value (there is no information whether it corresponds to a parameter or not).

Obviously, `ParamValue` $\subseteq$ `BasicValue`. With ASM library it is easy to get a control flow graph for a method. To build all branches of the process tree it is enough to explicitly *unfold* all possible paths in this control flow graph. Of course, if there are cycles in the original control flow graph, then process tree will be infinite in general case. However, there is a simple *folding* strategy to fold such process tree into a finite process graph. For the purposes of inference it will be enough just to inspect resulting process graph.

2.3 Configurations and folding strategy

Nodes in a process tree are labeled with *configurations*. *Kanva-micro* represents a configuration as a pair of a program point and a store of abstract values. More precisely, the configuration is a pair `(insnIndex, frame)`:

- `insnIndex` – an index of a current instruction, each method is represented as a finite sequence of bytecode instructions,
- `frame` – a store (list) of local variables and stack operands, where abstract values are of two kinds: `ParamValue` and `BasicValue`. For each program point the size of the store is fixed and known in advance.

A nice fact is that a number of all possible configurations of a method process tree is finite. So, there is a natural folding strategy: during construction of the process graph to fold a current configuration to a more general configuration in the history of the current branch (when a current configuration is an instance of some previous configuration). Folding is performed when $c \subseteq c'$, where c is a current configuration and c' is a previous configuration. The relation $c \subseteq c'$ holds when instruction indices are the same and corresponding values (stored in slots with the same index i) are related as $v_i \subseteq v'_i$. Moreover, there is no need to construct a traditional back folding edge. So, when opportunity for folding is detected, development of the current branch of a process tree is stopped and the current node is just marked as a "cycle" leaf. Taken this into account, in what follows terms *process tree*, *process graph* and *graph of configurations* are used interchangeably.

A process graph (or a graph of configurations) built in this way is similar in a spirit to one built during supercompilation [18]. There are two main differences from traditional supercompilation:

- Driving (unfolding of a control flow graph) is done over abstract values.
- No residual program is generated, but the constructed process graph is used for a quite specific task: approximation of a method execution in the perspective of a possible dereference of a `null` parameter.

2.4 Tracking dereferences, keeping only interesting branches

The process graph is constructed to answer the following question:

> Let a certain parameter be `null`, do all possible executions of the method result in errors caused by this `null`?

If the answer is "yes", it is correct to annotate this parameter as `@NotNull`.

So, if there is a conditional of the form

```
1  if (param == null) {
2      ...
3  } else {
4      ...
5  }
```

there is no interest in the `else`-branch and no development of such branch is done in the constructed process tree. A subtree in a process tree corresponding to `then`-branch is a *null-aware subtree* (the intuition is that a programmer consider a case when a parameter is `null` explicitly).

During driving step, when an instruction is executed over abstract values, it is possible to detect situations listed in subsection 2.1 when dereference of `ParamValue` happens. Such transitions are said to be *dereferencing transitions*.

2.5 Approximating method execution

So, using described folding strategy, tracking dereferences of a parameter and keeping only interesting branches a *finite* process tree is developed. Additional information used for nullness analysis is stored in nodes and edges:

- Some leaves are marked as *cycle* leaves.
- Some edges are marked as *dereferencing* ones.
- Some subtrees are marked as *null-aware* ones.

Based on this labeling information, another labels ("nullness labels") describing a method behavior are produced in a bottom-up manner. First, leaves of the process tree are labeled with following values:

- `RETURN` – a leaf contains a return instruction and no dereferencing edge was taken on the path from the root.
- `NPE` – a dereferencing edge was taken on the path from the root, or a leaf's configuration points to a `ATHROW` instruction and this leaf belongs to a null-aware subtree.
- `ERROR` – a leaf's configuration corresponds to a `ATHROW` instruction but this leaf doesn't belong to a null-aware subtree.
- `CYCLE` – a leaf is a cycle leaf.

Next, nullness labels for other nodes are inferred from labels of its children. If a node has a single child node, then label is just propagated from the child to the parent. If a node has more child nodes then child labels are combined according to the following table:

	RETURN	NPE	ERROR	CYCLE
RETURN	RETURN	RETURN	RETURN	RETURN
NPE	RETURN	NPE	NPE	NPE
ERROR	RETURN	NPE	ERROR	ERROR
CYCLE	RETURN	NPE	ERROR	CYCLE

Finally, the root node is labeled. If it is labeled with `NPE`, then the corresponding parameter is annotated as `@NotNull`.

A nullness label in the root node is in a sense an approximation of method execution with the following meaning:

- `RETURN` – there is a possible execution path which completes normally and no proof that a given parameter is dereferenced on this path was found.
- `NPE` – all possible execution paths result in an exception and there is at least one path when this exception is caused by `null` value of parameter.
- `ERROR` – all possible execution paths result in an exception but there is no information whether or not such error is caused by `null` value of parameter.
- `CYCLE` – just a loop.

The reason why `NPE` and `ERROR` labels are distinguished is that there may be a method which just throws an exception without checking parameters like in the following code:

```
1  public void log(String msg) {
2      throw new UnsupportedOperationException();
3  }
```

```
1  package kanva.analysis
2
3  import org.objectweb.asm.tree.*
4  import org.objectweb.asm.tree.analysis.*
5  import kanva.declarations.*
6  import kanva.graphs.*
7
8  fun buildCFG(method: Method, methodNode: MethodNode): Graph<Int> =
9    ControlFlowBuilder().buildCFG(method, methodNode)
10
11 private class ControlFlowBuilder(): Analyzer<BasicValue>(BasicInterpreter()) {
12   private class CfgBuilder: GraphBuilder<Int, Int, Graph<Int>>(true) {
13     override fun newNode(data: Int) = Node<Int>(data)
14     override fun newGraph() = Graph<Int>(true)
15   }
16
17   private var builder = CfgBuilder()
18
19   fun buildCFG(method: Method, methodNode: MethodNode): Graph<Int> {
20     builder = CfgBuilder()
21     analyze(method.declaringClass.internal, methodNode)
22     return builder.graph
23   }
24
25   override protected fun newControlFlowEdge(insn: Int, successor: Int) {
26     val fromNode = builder.getOrCreateNode(insn)
27     val toNode = builder.getOrCreateNode(successor)
28     builder.getOrCreateEdge(fromNode, toNode)
29   }
30 }
```

Fig. 1. Construction of a control-flow graph

2.6 Correctness

Correctness of inference is almost obvious – a parameter is assumed to be `null` and *all* possible execution paths are considered. A parameter is annotated as `@NotNull` only if all execution paths result in an exception, which, in turn, satisfies the requirement that the method cannot complete normally when parameter is `null`.

3 Implementation

Initially this task has arisen in the context of development of the Kotlin programming language pursuing safer interoperability of Kotlin and Java. So, *Kanva-micro* is coded in Kotlin. The implementation is rather concise since many lower-level things are delegated to ASM library [12].

Technically, the full cycle of annotating a Java library consists of following stages:

1. *Context construction.* Context is a list of all signatures, their bytecode in ASM representation and a storage for inferred annotations. At the next steps inferred annotations are put in the context. Inferred annotations can be fetched from the context by a method signature.

```
class ParamValue(tp: Type?): BasicValue(tp)
class InstanceOfCheckValue(tp: Type?): BasicValue(tp)
class Configuration(val insnIndex: Int, val frame: Frame<BasicValue>)
fun startConfiguration(
    method: Method, methodNode: MethodNode,paramIndex: Int
): Configuration {
    val frame = Frame<BasicValue>(methodNode.maxLocals, methodNode.maxStack)
    val returnType = Type.getReturnType(methodNode.desc)
    val returnValue =
      if (returnType == Type.VOID_TYPE) null else BasicValue(returnType)
    frame.setReturn(returnValue)
    val args = Type.getArgumentTypes(methodNode.desc)
    var local = 0
    if (!method.access.isStatic()) {
        val thisValue=
            BasicValue(Type.getObjectType(method.declaringClass.internal))
        frame.setLocal(local++, thisValue)
    }
    for (i in 0..args.size - 1) {
        val value =
          if (i == paramIndex) ParamValue(args[i]) else BasicValue(args[i])
        frame.setLocal(local++, value)
        if (args[i].getSize() == 2)
          frame.setLocal(local++, BasicValue.UNINITIALIZED_VALUE)
    }
    while (local < methodNode.maxLocals)
        frame.setLocal(local++, BasicValue.UNINITIALIZED_VALUE)
    return Configuration(0, frame)
}
```

Fig. 2. Construction of a start configuration

2. *Construction of dependency graph, calculation of strongly connected components.* What described in the previous section is just one iteration of the inference cycle. Annotations of different methods may depend on each other, since inference of annotations for a given method relies on annotations for methods called from the current method. So, annotating is an iterative process. To minimize the number of iterations, the graph of dependencies between methods is constructed, strongly connected components are calculated and then sorted in reverse topological order.
3. *Iterative inference within a single component.* All members of a component are put in a queue. Then members are pulled from this queue one by one and **the described algorithm is run for each of not yet annotated parameters**. If a new annotation is inferred, dependent methods are added into the queue. Obviously, this process converges.

Steps 1 and 2 are rather trivial and implemented in a standard way. An interested reader may consult the full source code for details. However, a single iteration of inference is rather interesting from a technical point of view. And this part heavily relies on ASM library.

First, a method's control flow graph is built. ASM provides a number of utilities for bytecode analyses. One of such utilities is `Analyzer`. `Analyzer` performs basic bytecode analyses given a semantic bytecode interpreter. Also ASM library provides a simple interpreter `BasicInterpreter`. `Analyzer` and

```
class ParamSpyInterpreter(val context: Context): BasicInterpreter() {
  var dereferenced = false
  fun reset() {
    dereferenced = false
  }

  public override fun unaryOperation(
   insn: AbstractInsnNode, value: BasicValue
  ): BasicValue? {
    if (value is ParamValue)
      when (insn.getOpcode()) {
        GETFIELD, ARRAYLENGTH, MONITORENTER ->
          dereferenced = true
        CHECKCAST ->
          return ParamValue(Type.getObjectType((insn as TypeInsnNode).desc))
        INSTANCEOF ->
          return InstanceOfCheckValue(Type.INT_TYPE)
      }
    return super.unaryOperation(insn, value);
  }

  public override fun binaryOperation(
    insn: AbstractInsnNode, v1: BasicValue, v2: BasicValue
  ): BasicValue? {
    if (v1 is ParamValue)
      when (insn.getOpcode()) {
        IALOAD, LALOAD, FALOAD, DALOAD, AALOAD,
        BALOAD, CALOAD, SALOAD, PUTFIELD ->
          dereferenced = true
      }
    return super.binaryOperation(insn, v1, v2)
  }

  public override fun ternaryOperation(
    insn: AbstractInsnNode, v1: BasicValue, v2: BasicValue, v3: BasicValue
  ): BasicValue? {
    if (v1 is ParamValue)
      when (insn.getOpcode()) {
        IASTORE, LASTORE, FASTORE, DASTORE,
        AASTORE, BASTORE, CASTORE, SASTORE ->
          dereferenced = true
      }
   return super.ternaryOperation(insn, v1, v2, v3)
  }

  public override fun naryOperation(
    insn: AbstractInsnNode, values: List<BasicValue>
  ): BasicValue? {
    if (insn.getOpcode() != INVOKESTATIC)
      dereferenced = values.first() is ParamValue
    if (insn is MethodInsnNode) {
      val method = context.findMethodByMethodInsnNode(insn)
      if (method != null && method.isStable())
        for (pos in context.findNotNullParamPositions(method))
          dereferenced = dereferenced || values[pos.index] is ParamValue
    }
    return super.naryOperation(insn, values);
  }
}
```

Fig. 3. Semantic interpreter for driving and tracking dereference of `ParamValue`

`BasicInterpreter` are used by *Kanva-micro* to construct a method's control flow graph. How this is done is shown in a listing in Figure 1. The function

```
class NullParamSpeculator(val methodContext: MethodContext, val pIdx: Int) {
  val method = methodContext.method
  val cfg = methodContext.cfg
  val methodNode = methodContext.methodNode
  val interpreter = ParamSpyInterpreter(methodContext.ctx)
  fun shouldBeNotNull(): Boolean = speculate() == Result.NPE
  fun speculate(): Result = speculate(
    startConfiguration(method, methodNode, pIdx), listOf(),false, false
  )

  fun speculate(
    conf: Configuration, history: List<Configuration>,
    alreadyDereferenced: Boolean, nullPath: Boolean
  ): Result {
    val insnIndex = conf.insnIndex
    val frame = conf.frame
    if (history.any{it.insnIndex==insnIndex && isInstanceOf(frame, it.frame)})
      return Result.CYCLE
    val cfgNode = cfg.findNode(insnIndex)!!
    val insnNode = methodNode.instructions[insnIndex]
    val (nextFrame, dereferencedHere) = execute(frame, insnNode)
    val nextConfs =
      cfgNode.successors.map{Configuration(it.insnIndex, nextFrame)}
    val nextHistory = history + conf
    val dereferenced = alreadyDereferenced || dereferencedHere
    val opCode = insnNode.getOpcode()
    return when {
      opCode.isReturn() && dereferenced -> Result.NPE
      opCode.isReturn()                 -> Result.RETURN
      opCode.isThrow() && dereferenced  -> Result.NPE
      opCode.isThrow() && nullPath      -> Result.NPE
      opCode.isThrow()                  -> Result.ERROR
      opCode == IFNONNULL && Frame(frame).pop() is ParamValue ->
        speculate(nextConfs.first(), nextHistory, dereferenced, true)
      opCode == IFNULL && Frame(frame).pop() is ParamValue ->
        speculate(nextConfs.last(), nextHistory, dereferenced, true)
      opCode == IFEQ && Frame(frame).pop() is InstanceOfCheckValue ->
        speculate(nextConfs.last(), nextHistory, dereferenced, true)
      opCode == IFNE && Frame(frame).pop() is InstanceOfCheckValue ->
        speculate(nextConfs.first(), nextHistory, dereferenced, true)
      else ->
        nextConfs.map{
          speculate(it, nextHistory, dereferenced, nullPath)
        } reduce{ r1, r2 -> r1 join r2}
    }
  }

  fun execute(
    frame: Frame<BasicValue>, insnNode: AbstractInsnNode
  ): Pair<Frame<BasicValue>, Boolean> = when (insnNode.getType()) {
    AbstractInsnNode.LABEL, AbstractInsnNode.LINE, AbstractInsnNode.FRAME ->
      Pair(frame, false)
    else -> {
      val nextFrame = Frame(frame)
      interpreter.reset()
      nextFrame.execute(insnNode, interpreter)
      Pair(nextFrame, interpreter.dereferenced)
    }
  }
}
```

Fig. 4. Inference of `@NotNull` annotation

`buildCFG` builds a directed graph whose nodes are labeled with indices of instructions of a method and edges correspond to transitions between instructions.

`BasicValue` introduced in subsection 2.2 is already implemented in ASM. The class `Frame` provided by ASM corresponds to a frame holding abstract values. `BasicInterpreter` already implements execution of bytecode instructions over `BasicValue`s is the desired way. *Kanva-micro* extends `BasicInterpreter` in order to distinguish between `BasicValue` and `ParamValue`. Notions of `ParamValue`s and configurations are depicted in a listing in Figure 2. Class `InstanceOfCheckValue` is for tracking `instanceof` checks. The function `startConfiguration` presented in Figure 2 creates a start configuration (placed in the root node of a process tree) for a given method and an index of a parameter being analyzed. The main logic of `startConfiguration` is that all values in frame except a given parameter are initialized with `BasicValue`.

`BasicInterpreter` provided by ASM already has almost everything needed for driving. The missed parts are:

- Tracking of dereferencing of `ParamValue`.
- Handling of `instanceof` checks of `ParamValue`.
- Knowledge about already inferred annotations (to detect dereferencing).
- Propagation of `ParamValue` during class casting.

All these parts are implemented in class `ParamSpyInterpreter` shown in Figure 3. The most interested lines are 52-56: if the current parameter of interest is passed as an argument to another parameter (of some method) already annotated as `@NotNull`, it is handled in the same way as dereferencing of the current parameter.

The main analysis is implemented in the class `NullParamSpeculator` shown in Figure 4. `NullParamSpeculator` holds a `methodContext`, which contains everything needed for inference, and an index of a parameter being annotated. The method `shouldBeNotNull` returns `true` if an approximation of method execution is `NPE`. A process tree is not constructed explicitly here, since it is enough to get a nullness label for the root configuration. The call to `speculate(conf, history, alreadyDereferenced, nullPath)` results in one of `RETURN`, `NPE`, `ERROR`, `CYCLE` nullness labels. The call arguments are:

- `conf` – the current configuration (for which nullness label should be calculated),
- `history` – a list of already encountered configurations (for folding),
- `alreadyDereferenced` – whether dereferencing was already detected on a path from the root to current configuration,
- `nullPath` – if `nullPath` is `true`, it means that current configuration belongs to a null-aware subtree.

Let's iterate through the code of the `speculate` method line-by-line.

If there is a more general configuration in the history, folding is performed, the corresponding label is `CYCLE`. Otherwise, the current instruction is executed – the `execute` method returns a pair of a next frame and a boolean whether there

was dereferencing of the parameter during instruction execution. If the current instruction is a return or throw instruction, then a nullness label is calculated based on the `dereferenced` and `nullPath` flags. Otherwise, if the current instruction is `IFNULL` or `IFNONNULL` and a value being tested is `ParamValue` (it corresponds to conditionals `if (param == null)` and `if (param != null)`), a corresponding null-aware subtree is processed (the `nullPath` flag to `true`).

The same logic applies to handling of `if (param instanceof SomeClass)` conditional. When `param` is `null`, this check results in `false`. The implementation is a bit verbose since there is no special instruction in Java bytecode for such conditional and this check is compiled into the sequence of two instructions: `INSTANCEOF` and `IFEQ`. The `INSTANCEOF` instruction is handled by `ParamSpyInterpreter`: if an operand is `ParamValue`, then a special `InstanceOfCheckValue` value is produced. The `IFEQ` instruction is handled inside the `speculate` method: when the current instruction is `IFEQ` and an operand on the top of the stack is `InstanceOfCheckValue`, then the `if (param instanceof SomeClass)` construction is recognized and only a branch that corresponds to null parameter is considered. (Handling of the `IFNE` instruction corresponds to `if (!(param instanceof SomeClass))` construction.)

Otherwise, nullness labels for child configurations are calculated and combined. This concludes the discussion of the implementation.

4 Discussion

In a sense, *Kanva-micro* performs domain-specific supercompilation [15] of Java bytecode, abstracting away almost all aspects of operational semantics not associated with nullness analysis. Because of these abstractions, representation of configurations becomes extremely simple – just a bit vector. The interesting fact is that configurations are so generalized in advance, that *no traditional online generalization* is required to ensure termination of supercompilation. But this comes for the price that a constructed process tree of method execution is not perfect in a general case.

4.1 The cost of simplifications

The main point of the *Kanva-micro* project is simplicity, focusing on the essence of the method and ignoring some technical details for the sake of brevity of presentation. However, there are two significant drawbacks that simplifications that make *Kanva-micro* not ready for production use.

Exponential complexity The main drawback of *Kanva-micro* is that this algorithm is of exponential complexity in general case. This complexity may exploded by a method with sequential conditionals.

```
1  if ( ... ) {
2
3  }
```

```
4  if ( ... ) {
5
6  }
7  if ( ... ) {
8
9  }
```

If there are n sequential conditionals in a method, then process tree constructed by *Kanva-micro* will contain 2^n branches in the worst case.

Memory usage The bytecode of a library method is processed by *Kanva-micro* more than one time: the first time when a graph of dependencies between methods is constructed and then during iterative inference of annotations inside a strongly connected component of the graph of dependencies. Loading and parsing the bytecode of a method from scratch every time without additional processing of binaries is problematic, so *Kanva-micro* loads all library bytecode into memory at once in advance. This means that the amount of memory required by *Kanva-micro* is proportional to the size of a library, which is not acceptable from a practical point of view.

4.2 The Faba project

The *Faba* project [4] overcomes mentioned drawbacks by smart handling of the library bytecode. *Faba* processes a binary library in two stages:

1. Indexing a library: the result of indexing is a set of equations over a lattice.
2. Solving equations.

At the first stage each method is processed exactly once. After the bytecode for a method is indexed, it is unloaded. Equations are not memory consuming, so the problem of memory usage disappears.

During indexing a method, *Faba* exploits memoization and sharing facilities. The main observation is that in a sequence of conditionals in *real libraries* the majority of conditionals are irrelevant to nullness analysis (do not test a parameter for nullity). Driving of both branches of "irrelevant" conditions result *in most cases* in the same configurations in two nodes of the process tree, these nodes are joined. In general case *Faba* is also of exponential complexity, but this exponential complexity is not exploded by *real-world libraries*.

Both problems may be tamed in naive but simple ways: the memory usage problem may be solved via unloading the bytecode for a method after its bytecode is processed by the current iteration and loading it from the scratch from the disk. A simple ad-hoc way to mitigate exponential complexity of *Kanva-micro* is just to limit the number of processed configurations. When this limit is reached, analysis for the method stops and infers nothing.

4.3 Experiments and more details

The *Kanva-micro* project [8] provides utilities to annotate popular Java libraries (available as Maven artifacts) in a simple way. The project page also has a set of

experimental result for running inference with different settings. The interesting fact is that limiting the number of processed configurations by a reasonable number (say, by 5000) *Kanva-micro* infers about 95 percent of annotations inferred by *Faba* in comparable time.

An interested reader may also consult the *Kanva-micro*'s wiki for more more technical details related to implementation and experiments.

5 Related work

Initially *Kanva-micro* was developed in the context of JetBrains KAnnotator tool [7]. KAnnotator is based on abstract interpretation and infers different nullness annotations (for method parameters and for method results) in a single pass. So, abstract domains and logic of approximations in KAnnotator is much more complex that of *Kanva-micro*.

On the contrary, *Kanva-micro* is specialized to infer just one type of nullness annotations. The *Faba* project infers not only `@NotNull` annotations for method parameters, but also `@NotNull` annotations for method results and `@Contract` annotations [5]. All *Faba* inferencers are quite similar and based on supercompilation but have very different abstract domains, logic of approximations and logic for sharing configurations.

The pragmatic observation from developing *Kanva-micro* and *Faba* is that it is more practical to have a set of specialized inferencers which run independently and may reuse results of each other via context rather than a tool that runs different analyses together in a single pass.

The main goal of KAnnotator, *Kanva-micro* and *Faba* is to annotate existing Java libraries for safer usage. Inference of annotations happens on bytecode level, no source is required.

Surprisingly, as we can judge from existing literature, this task was not addressed in academia from practical point of view before. The closest existing tool is NIT [14]. NIT infers `@NotNull` and `@Nullable` annotations but these annotations have different semantics. NIT considers Java bytecode as a single application and starts analysis from so called entry points. A `@NotNull` parameter annotation in NIT setting means that during execution of an application `null` will never be passed into this parameter, other annotations have similar semantics – they describe which values may be passed to parameters and returned from methods during executions of a specific application. NIT doesn't consider bytecode at library level. NIT motivation is that such analysis maybe used to detect bugs in an applications. Another possible application of NIT annotations is bytecode optimizations – removing unnessecary checks from bytecode.

Another tool that infers nullness information from bytecode is Julia [17]. Again, this information is inferred with the goal of analysis – the main application is to generate a set of warnings about possible null pointer exceptions.

There is a tool called JACK [11,16] which verifies Java bytecode with respect of `@NotNull` annotations, ensuring that `null` will never be passed to a `@NotNull` variable or parameter.

Note that *Kanva-micro* annotations are semantic-based. There is a lot of works devoted to checking and inferencing nullness annotations in source code, but these annotations have different semantics, since they may forbid some executions paths not resulting in null pointer exception. Also many source-based annotation inferencers require an additional user's input.

References

1. ASM Framework. http://asm.ow2.org/.
2. Ceylon programming language. http://ceylon-lang.org/.
3. Eclipse user guide: using null annotations. http://help.eclipse.org/kepler/topic/org.eclipse.jdt.doc.user/tasks/task-using_null_annotations.htm.
4. Faba, fast bytecode analysis. https://github.com/ilya-klyuchnikov/faba.
5. IntelliJ IDEA 13.1 Help. @Contract Annotations. http://www.jetbrains.com/idea/webhelp/@contract-annotations.html.
6. IntelliJ IDEA How-To, Nullable How-To. https://www.jetbrains.com/idea/documentation/howto.html.
7. KAnnotator. https://github.com/JetBrains/kannotator.
8. Kanva-micro. https://github.com/ilya-klyuchnikov/kanva-micro.
9. Kotlin programming language. http://kotlin.jetbrains.org/.
10. The Checker Framework. Custom pluggable types for Java. http://types.cs.washington.edu/checker-framework/.
11. The Java Annotation Checker (JACK). http://homepages.ecs.vuw.ac.nz/~djp/JACK/.
12. E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.
13. R. Glück and A. Klimov. Occam's razor in metacompuation: the notion of a perfect process tree. In *WSA '93: Proceedings of the Third International Workshop on Static Analysis*, pages 112–123, London, UK, 1993. Springer-Verlag.
14. L. Hubert. A non-null annotation inferencer for java bytecode. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 36–42. ACM, 2008.
15. A. Klimov, I. Klyuchnikov, and S. Romanenko. Automatic verification of counter systems via domain-specific multi-result supercompilation. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.
16. C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Compiler Construction*, pages 229–244. Springer, 2008.
17. F. Spoto. The nullness analyser of julia. In E. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 405–424. Springer Berlin Heidelberg, 2010.
18. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.