ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Reasoning about Nondeterministic Collections

Bachelor Thesis

Bogdan Gadzhylov

April 12, 2021

Advisors: Prof. Dr. Peter Müller, Dr. Malte Schwerhoff
Department of Computer Science, ETH Zürich

Abstract

The Java platform provides the Java collections framework, which allows to represent and manipulate groups of objects. Different implementations ensure different properties and have different advantages and disadvantages. In particular, some collection implementations, according to the documentation, do not guarantee deterministic traversal-order of their elements. This can potentially lead to nondeterministic output, which could cause confusion and complicate testing of the software. In order to generate deterministic output, users can avoid nondeterministic collections altogether or they can sort the data before generating output. Both strategies, however, are potentially costly in terms of performance.

In this thesis we developed a tool for a small subset of Java that can find nondeterministic collection traversals using static analysis, and warn users about potentially nondeterministic output. Our tool is using the Soot framework in order to perform abstract interpretation on the Jimple intermediate representation. We evaluated our tool by running its determinism analysis on example programs of different complexity and comparing with expected results.

Contents

Chapter 1

Introduction

Reasoning about properties of a program is generally a difficult task due to high complexity of modern software systems. One very important property is determinism. Intuitively, a program is said to be deterministic, if for a given state and a fixed input, it always yields the same observable effect. This property is often desired at a point in the program where output is generated. It could be very confusing for users, if rerunning the program without changing any input would result in different output.

In this thesis, we focus only on nondeterminism introduced by the Java collections framework [4]. The main goal of this thesis is to develop a tool that detects potentially nondeterministic output using static analysis, and produces warnings for the user. For this, we will define different types of nondeterminism and instantiate the abstract interpretation technique to perform a determinism analysis. We will implement the tool using the Soot framework [10].

1.1 Background

1.1.1 Java collections framework

The Java collections framework is a commonly used framework that allows to represent and manipulate groups of objects, independently of implementation details. Figure 1.1 shows an overview of the framework where the different interfaces are represented in red and the implementations in green.

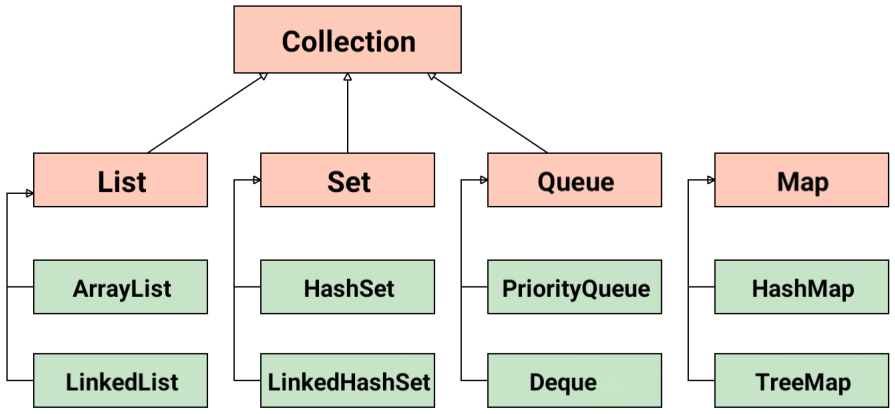


Figure 1.1: Example interfaces (red) and implementing classes (green) of the Java collections framework

The `Set` interface, for example, could be implemented as a `HashSet` [1] or as a `LinkedHashSet` [2]. Different implementations ensure different properties and have different strong and weak points. In particular, the `LinkedHashSet` implementation uses a doubly-linked list to guarantee deterministic traversal-order, whereas the `HashSet` implementation does not guarantee that the traversal-order will remain constant over time. This happens because the `HashSet` implementation relies on the `hashCode()` function, which in its turn does not guarantee the same integer for the same object across multiple executions [5]. But there are also other collection implementations that do not guarantee any particular traversal-order, e.g. `PriorityQueue` [3].

1.1.2 Soot framework

Soot [10] was originally developed by the the Sable Research Group from McGill University as a Java optimisation framework, but is now widely used to analyse, instrument, optimise and visualise Java and Android applications.

One of the main benefits of Soot is that it provides four different intermediate representations for analysis purposes. Each of the intermediate representations has different levels of abstraction that give different benefits [11]. The most important intermediate representation is Jimple. Soot generates Jimple code from the Java bytecode of the analysed programs.

Jimple

Jimple is a typed intermediate representation of Java source code based on three-address code [8]. This means that in each assignment statement there is only one address on the left side and at most two addresses on the right

side of the assignment. Hence, complicated statements with nested expressions are split up into multiple simple statements using additional local variables to store intermediate results. This makes it a lot easier to analyse Jimple code instead of Java source code. Moreover, Jimple has 15 different operations in total, as opposed to over 200 possible operations in Java bytecode, which makes it a lot easier to analyse Jimple code instead of Java bytecode.

The following is an example of a simple Java program and its corresponding Jimple representation:

```
1 public static void main(String[] args){
2
3    Set<String> hs = new HashSet<String>();
4
5    hs.add("abc");
6
7    System.out.println(hs);
8
9 }
```

Listing 1.1: Java source code of a simple program

```
1  public static void main(java.lang.String[]){
2
3      java.util.HashSet $stack2;
4      java.io.PrintStream $stack6;
5      java.lang.String[] args;
6
7      args := @parameter0: java.lang.String[];
8
9      $stack2 = new java.util.HashSet;
10
11     specialinvoke $stack2.<java.util.HashSet: void <init>()>();
12
13     interfaceinvoke $stack2.<java.util.Set: boolean
           add(java.lang.Object)>("abc");
14
15     $stack6 = <java.lang.System: java.io.PrintStream out>;
16
17     virtualinvoke $stack6.<java.io.PrintStream: void
           println(java.lang.Object)>($stack2);
18
19     return;
20 }
```

Listing 1.2: Jimple code corresponding to the program from Listing 1.1

Soot provides a lot of interfaces and classes to manage the analysed Jimple code. Here are the most important components for our analysis:

- `Stmt`: Represents a statement in Jimple. Since Jimple is based on three-address code, we can assume that each statement uses at most three operands. Jimple has 15 different statement types in total, but the most relevant for us are:
  - `DefinitionStmt`: Represents an assignment statement. In Jimple there are two types of assignments:
    - `IdentityStmt`: Parameters of a method and references to `this` are explicitly stored in local variables, using the `:=` operator
    - `AssignStmt`: Represents all other assignments
  - `IfStmt`: Represents an if-statement, and is used together with `GotoStmt` for intraprocedural control-flow
  - `InvokeStmt`: Represents a method invocation statement, and is used together with `ReturnStmt` and `ReturnVoidStmt` for interprocedural control-flow
- `Value`: Represents a single datum. The most important examples of values are:
  - `Local`: Represents a local variable. The name of a Jimple local variable starts with a `$` if this variable was created artificially during the "Jimplification" and was not present as a local variable in the Java source code itself.
  - `Constant`: Represents a constant value. It includes usual numerical, boolean and String literals, but also the null pointer.
  - `Expr`: Represents an expression. Most important expressions are:
    - `BinopExpr`: Represents an expression with two operands
    - `UnopExpr`: Represents an expression with one operand
    - `CastExpr`: Represents a cast expression
    - `InstanceOfExpr`: Represents a check via `instanceof` operator
    - `InvokeExpr`: Represents an invocation of a method inside an expression
    - `NewExpr`: Represents an instantiation via the `new` operator

SPARK

SPARK [19] is a flexible framework based on Soot that provides a lot of possibilities to perform different points-to analyses. A points-to analysis allows to statically determine which pointer variables may point to which objects during program execution. We will use this information to get possible concrete types that a collection instance might have during execution.

SPARK comes together with Soot and can be turned on by setting the corresponding Soot options. SPARK does not only compute points-to information but is also used for call graph construction. Even though SPARK computes context-insensitive points-to information, it is still quite precise. The reason for this is that, unfortunately, context-sensitive points-to information is quite expensive to compute and even trivial Java programs could take a long time. Hence, in order to be able to analyse even large programs in a short time, we will use SPARK's points-to information in our determinism analysis.

The points-to information is accessible through Soot's `PointsToAnalysis` interface. This interface provides a method `reachingObjects(Local l)`, which returns a `PointsToSet`, a set of abstract objects the Jimple local `l` may point to. We could then call the method `possibleTypes()` on a `PointsToSet` instance and get a set of `Types` of the objects in the `PointsToSet`. This essentially gives us a set of concrete `Types` that the `Local l` could have at runtime. For example, a `Local l` of `Type java.util.Set` could have a set of possible concrete types `[java.util.HashSet, java.util.TreeSet]`.
The `PointsToSet` interface also has a method `hasNonEmptyIntersection(PointsToSet other)`, which can be used for alias detection.

1.1.3 Abstract Interpretation

Abstract interpretation [15] is a static analysis technique that is used to overapproximate all possible behaviours of a program. The main idea of abstract interpretation is to represent possibly infinitely many concrete program states with finitely many abstract program states. For this, an abstract domain is defined, which is then used to represent an abstract program state. Usually, the abstract state of a program, at a program point, is a mapping from all the variables of a program to some abstract values from the abstract domain. The abstract domain is chosen according to the kind of property one wants to analyse. The abstract domain should be a complete lattice. All the possible behaviours of the program are captured using abstract transformers. Abstract transformers define how the abstract state changes when an arbitrary statement is executed. Abstract transformers are based on the analysis one wants to perform and are defined once for the whole programming language. This means that abstract transformers must cover all possible statements that a program might have, such that the analysis can run on

any example program. Abstract transformers must also be sound, i.e. the abstract state change must represent all possible concrete state changes
When abstract domain, abstract program state and abstract transformers are defined, the analysed program can be executed in an abstract way. The execution will proceed until all the abstract states at all program points do not change anymore, i.e. a fixed point is reached.
To ensure termination of the analysis, it is necessary to define how the abstract state should be overapproximated in case the abstract state keeps changing without reaching a fixed point. This is usually done through widening [16].
After the abstract execution we can then use the abstract states for each program point to reason about the analysed property.

1.2 Subset of Java

Java is an expressive and complex programming language with a huge amount of different features and possibilities. This makes it impossible, in the scope of a bachelor's thesis, to create a reasonably precise static analysis tool that covers all possible Java programs. Hence, we will describe the subset of Java programs that our tool will be able to analyse. This allows us to get a working determinism analysis without having to cover all the numerous details.

We will describe the subset of Java programs that we consider in a grammar on the next page.

Legend for the grammar shown next:

- P = Program
- MC = Main Class
- CD = Class Declaration
- VD = Variable Declaration
- MD = Method Declaration
- S = Statement
- E = Expression
- T = Type
- ID = Identifier
- LIT = Java literals, including integers, floats, booleans, Strings and `null`
- op = Basic binary Java operators for arithmetical, relational and logical operations

Grammar

P ::= **MC** $\overline{\textbf{CD}}$

MC ::= `class` **ID** {`public static void main ( String [ ]` *args*){**S**}}

CD ::= `class` **ID** `extends` **ID**{$\overline{\textbf{VD}}$ $\overline{\textbf{MD}}$}

VD ::= **T ID** ;

MD ::= `public` **T ID** `(` $\overline{\textbf{T ID}}$ `)` {$\overline{\textbf{VD}}$ $\overline{\textbf{S}}$ `return` **E** ; }

S ::= { $\overline{\textbf{S}}$ }
| `if(`**E**`)` **S** `else` **S**
| `while(`**E**`)` **S**
| `for(`**S**;**E**;**S**`)` **S**
| **VD** ;
| **ID** = **E** ;
| **ID**`(` $\overline{\textbf{E}}$ `)` ;

E ::= **E** *op* **E**
| `!` **E**
| `(` **T** `)` **E**
| `(` **E** `)`
| **ID**`(` $\overline{\textbf{E}}$ `)`
| `new` **T** `(` $\overline{\textbf{E}}$ `)`
| **ID**
| `this`
| *LIT*

T ::= `String` | `int` | `double` | `boolean` | **ID**

ID ::= legal Java identifiers

Notes:

- Overline denotes repetition, zero or more times. For a better overview we avoided commas in the repetition.
- For simplicity we represent method calls as **ID**`(` $\overline{\textbf{E}}$ `)`. But we also allow method calls like `System.out.println()`, `Collections.sort()` or `var.foo()`, for a variable `var`.

Not all constraints can be expressed in a grammar, and even those that can may not be immediately noticed, so we will explicitly mention some constraints here.

Constraints

- We do not include arrays and fields.
  - *Why?* Arrays and fields would introduce a whole new complexity layer to our analysis. In general, it is a challenging task to also reason about the heap in a static analysis.
- We do not include recursion.
  - *Why?* Our determinism analysis will be interprocedural, i.e. we will analyse the source code of the called methods (if it is not a Java library method and if we have the source code for it). Both direct and indirect recursion would introduce loops into our analysis, which would make our analysis run until there is no more memory to maintain the analysis information. To ensure termination we would need to stop analysing methods which we already analysed and make some sound assumptions on the effect of these methods. But since a method call could theoretically change the whole abstract state of a program at a particular point, we would need to assume the most general abstract state and basically lose all the previously computed information.
    Another option could be to build procedure summaries for each method, but this would again add a whole complexity layer [17].
- We do not analyse dynamically-dispatched method calls.
  - *Why?* To statically find out which concrete method implementation could be called during program execution is also a challenging task in itself. One could also analyse all possible implementations and compute an overapproximation of the effect of the method call, but this could result in poor performance and usability of our tool. For methods that we do not analyse we make assumptions on the effect of the method call and allow users to configure these assumptions.

- We allow methods to only have one `return` statement at the end.
  - *Why?* For simplicity. Theoretically, any program could be rewritten to this form by just introducing a local variable in the beginning of the method. This local variable would store potentially different return values, which are finally returned at the very end of the method.
- We do not include exceptions.
  - *Why?* Also for simplicity. We want to keep the main focus of our work on the actual determinism analysis.
- We only consider sequential Java programs.
  - *Why?* Parallelism would introduce another complexity layer to the analysis.

Chapter 2

Definitions of Determinism

In this chapter we will provide our definitions of different types of determinism, show examples to help understand the definitions and describe the implications that these definitions have.

2.1 Notation

Before we define different types of determinism, we introduce some notation:

- Let **P** denote our program that we analyse. **P** is fixed.
- Let **e** denote an arbitrary execution of our program **P**.
- Let **I** denote all of the input that the program gets during one execution. As input we consider not only the statically known arguments, but also any information that is arriving dynamically during the execution, from outer sources. This helps us to focus on nondeterminism introduced by the Java collections framework.
  I is fixed.
- Let **p** denote a program location in the program **P**.

2.2 Definitions

In this thesis we distinguish between two types of determinism: *traversal-order determinism* and *content determinism*. Here we provide intuitive definitions of these concepts. Since we are interested in finding potentially nondeterministic output, we define determinism by comparing the output in different executions.

Definition 2.1 *(Traversal-order determinism). A collection instance is deterministic in its traversal-order, at the point* **p** *in the program* **P**, *if for all possible executions* **e** *that have the same input* **I** *outputting the collection instance at the point* **p** *would yield the same result. This means that if we introduced an output operation on this collection instance at point* **p**, *it would always have exactly the same observable effect. Otherwise, the collection instance is nondeterministic in its traversal-order at the point* **p**.

The following is a simple example of a `HashSet` instance, which is nondeterministic in its traversal-order at the point in the program where output is produced, i.e. at line 7.

```
1 Set<String> hs = new HashSet<String>();
2
3 hs.add(" a ");
4 hs.add(" b ");
5 hs.add(" c ");
6
7 System.out.println(hs);
```

Listing 2.1: The `HashSet` instance `hs` is nondeterministic in its traversal-order at line 7

According to the documentation of the `HashSet` class [1], the traversal-order of its elements is not guaranteed to remain constant, i.e. it could be different in different executions with the same input.
Note that, according to our definition, `hs` is also nondeterministic in its traversal-order at lines 5 and 6, since we could introduce another print statement there and potentially see different output in different executions.

One might think that traversal-order nondeterminism is inherent to some collection classes whose documentation does not guarantee a particular order, but this is not the case, as the next example shows.

```
1 // let hs be defined and filled as in Listing 2.1
2
3 List<String> list = new LinkedList<String>(hs);
4
5 System.out.println(list);
```

Listing 2.2: The `LinkedList` instance `list` is nondeterministic in its traversal-order at line 5

Since the order of the elements in `hs` is nondeterministic, the order of the elements in `list` is also nondeterministic. Even though the `LinkedList` class inherently provides a strict order of the elements, the actual `LinkedList` instance `list` is nondeterministic in its traversal-order at line 5, since the output could be different in different executions.

Note: In some cases it is possible to recover traversal-order determinism. For example, if we sorted the collection instance `list` at line 4 of Listing 2.2, then the traversal-order would become deterministic, thus always producing the same output at line 5. Recovering content determinism, on the other hand, seems highly unlikely to happen in practice.

Definition 2.2 *(Content determinism). A collection instance is deterministic in its content, at the point* **p** *in the program* **P**, *if for all possible executions* **e** *that have the same input* **I**, *the collection instance has the same content at the point* **p**. *With "same content" we mean that the multisets of values of the contained objects are equal, so the traversal-order does not matter.*

Going back to previous examples, we can see that both the `hs` from Listing 2.1 and the `list` from Listing 2.2 are deterministic in their content at the points where output is produced.

The following example shows a collection instance that is nondeterministic in its content.

```
1  Set<String> hs = new HashSet<String>();
2
3  hs.add(" a ");
4  hs.add(" b ");
5  hs.add(" c ");
6
7  List<String> list = new LinkedList<String>();
8
9  Iterator<String> it = hs.iterator();
10
11 list.add(it.next());
12 list.add(it.next());
13
14 System.out.println(list);
```

Listing 2.3: The `LinkedList` instance `list` is nondeterministic in its content at line 14

In Listing 2.3 the `list` is filled with two elements from `hs`. But since the traversal-order of the elements in `hs` is nondeterministic, different elements could be added to `list` in different executions with the same input.

Note that sorting does not change the content of a collection and would not help to restore determinism in this case. In general, restoring content

determinism of a collection would include overwriting all of the elements with some deterministic values, which seems rather unlikely to happen in practice. Note also that the `list` from Listing 2.3 is nondeterministic in its traversal-order at line 14 according to our definitions.

Definition 2.3 *(Nondeterministic collection instance). For simplicity, we say that a collection instance is nondeterministic if it is nondeterministic in its traversal-order or content.*

So far we have defined determinism only as a property of a collection instance at a particular point in the program. But since the ultimate goal of this project is to develop a tool that warns users about potentially nondeterministic output, we also need to define determinism of the output.

Definition 2.4 *(Potentially nondeterministic output). Output is potentially nondeterminsitic if it is generated from a nondeterministic collection instance. In general, any user-observable behaviour could be viewed as output. For simplicity, we consider the standard `System.out.println()` to be the canonical output function.*

Going back to previous examples we can see that in Listing 2.1, Listing 2.2 and Listing 2.3 there is potentially nondeterministic output.

Finally, we want to also reason about determinism of other elements that are not collection instances. This will be useful for our determinism analysis. For example, in order to see that the `list` from Listing 2.3 is nondeterministic in its content at line 14, we have to reason about the determinism of the iterator `it` and the elements returned by `it.next()`.

Remark 2.5 *We consider elements that are not collection instances, i.e. do not come from the Java collections framework, as collections with only one element. This allows us to apply our determinism definitions to all elements in our program. This also means that the determinism of such elements boils down to content determinism, since there is only one element in these collections.*

Remark 2.6 *Iterators are a special case. Determinism of an iterator depends on the determinism of the collection instance it is attached to. If this collection instance is deterministic in its traversal-order, then also the iterator is deterministic. If the collection instance is nondeterministic in its traversal-order, then also the iterator is nondeterministic in its traversal-order. Calling `next()` on this iterator would then return an element with nondeterministic content.*

2.3 Implications of the Definitions

Choosing definitions this way has an important consequence:

- If a collection instance is nondeterministic in its content, it is automatically nondeterministic in its traversal-order, since we defined traversal-order determinism in terms of identical output. Equivalently, if a collection instance is deterministic in its traversal-order, it must also be deterministic in its content. Traversal-order is a stronger property in this perspective.

We also address some open questions here.

Question 1: What if an output point is reached within nondeterministic control flow?

This would mean that not all executions with the same inputs will reach that output point. The following example illustrates such a case:

```
1  //HashSet with elements 1,2,3,4,5
2  HashSet<Integer> hs = new HashSet<Integer>(Arrays.asList(1,2,3,4,5));
3
4  int n = hs.iterator().next();
5
6  LinkedList<String> list = Arrays.asList("a", "b", "c");
7
8  if(n > 3) {
9     System.out.println(list);
10 }
```

Listing 2.4: Output point (line 9) lies within a nondeterministic control flow because of the condition `n > 3` and the fact that `n` is nondeterministic in its content

Note that in Listing 2.4 we use that `hs.iterator().next()` is nondeterministic to introduce nondeterministic control flow. Even though the entire input of our program might be completely identical, it could happen that some executions reach the output point at line 9 and some do not. This would lead to different overall output of the program.

To deal with this we would need to statically determine whether each branch condition of our program is deterministic for the same input, but this would require reasoning about all possible nondeterminisms, which is a much more difficult task than reasoning about nondeterministic collections. Instead we make a decision:

- **Decision 1:** Given two executions $\mathbf{e_1}, \mathbf{e_2}$, input $\mathbf{I}$ and output point $\mathbf{p}$, in our analysis we assume that we reach $\mathbf{p}$ in both executions. This means, to analyse determinism of the output, we only consider executions that

are comparable at the point **p**, i.e. both reach point **p** and both have the same input **I**. Hence we do not consider the overall output of the program. The overall output size could vary between different executions, but for the same overall output it should be deterministic.

With this decision our analysis for the program in Listing 2.4 should not result in a warning, since the output only uses a `LinkedList` instance, which is deterministic in its content and traversal-order at the output point.

Question 2: What if we had an output point in a loop?

In a loop, a program point **p** could be reached multiple times and thus some collections could be deterministic in one iteration at the point **p** and nondeterministic in another iteration at the point **p**.

```
1  //HashSet with elements "a", "b"
2  HashSet<String> hs = new HashSet<String>(Arrays.asList("a", "b"));
3
4  LinkedList<String> list = Arrays.asList("c", "d");
5
6  for(int i = 0; i < 5; i++){
7      if(i == 3){
8          list.add(hs.iterator().next());
9      }
10     System.out.println(list);
11 }
```

Listing 2.5: Output point (line 10) lies in a loop and is deterministic in some iterations and nondeterministic in other iterations

To deal with this we make another decision:

- **Decision 2:** We overapproximate in our analysis and say that if a collection is nondeterministic at the point **p** at least once, then it is considered nondeterministic at the point **p** overall.

With this decision our analysis for the program in Listing 2.5 should result in a warning, since the `list` has nondeterministic content in some iterations.

Chapter 3

Abstract Interpretation

In this chapter we will describe how we instantiate the abstract interpretation technique to perform our determinism analysis. We will define the abstract domain, abstract transformers and abstract state for our analysis.

3.1 Abstract Domain

For our determinism analysis we choose the abstract domain as

$$(S, \sqsubseteq, \sqcap, \sqcup)$$

with $S := \{\bot, D_o D_c, N_o D_c, N_o N_c = \top\}$, where

D_o stands for deterministic traversal-order,

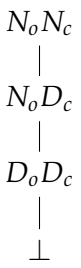
D_c stands for deterministic content,

N_o stands for nondeterministic traversal-order,

N_c stands for nondeterministic content,

according to our definitions of (non)determinism. This abstract domain is the set of possible abstract values that our abstract objects could have during abstract interpretation. An abstract object is either a concrete object abstracted using its allocation site or a variable of primitive type. If an abstract object has the abstract value $D_o D_c$, it means that it is deterministic in both traversal-order and content. This holds for other abstract values analogously. $\bot$ represents the abstract value of abstract objects that were not yet initialised. $\top$ is the most general and imprecise abstract value an abstract object could have, which, according to our definitions, is equivalent to $N_o N_c$.

The partial order between the elements ($\sqsubseteq$), as well as the least upper bound ($\sqcup$) and the greatest lower ($\sqcap$) bound can be seen in the following Hasse diagram:



Note: We do not consider D_oN_c as a possible value in our abstract domain, since our definitions imply $D_o \implies D_c$, thus making D_oN_c unreachable.

3.2 Abstract State

Having defined the abstract domain for our abstract interpretation, we need to describe what an abstract state at a program point looks like. For this, let us introduce some definitions:

- Let O be the set of all abstract objects of our program. We abstract concrete objects using their allocation sites. We abstract variables of primitive types using their names. We consider variables of primitive types as abstract objects in order to be able to represent their abstract values. For example, if we get the first element of a `HashSet` of integers and store it in a variable of type `int`, we want that variable to have the abstract value N_oN_c, since a `HashSet` is nondeterministic in its traversal-order.
- Let V be the set of all variables of our program.
- Let p be an arbitrary program point.

An abstract state at a program point p in our abstract interpretation consists of three mappings:

$$m_1 : V \to \mathcal{P}(O)$$

$$m_2 : O \to S$$

$$m_3 : V \to S$$

- m_1 is a mapping from variables to sets of abstract objects they could point to. This mapping is given by the points-to analysis. Since we also consider variables of primitive types as abstract objects, we make these variables point to themselves.

- m_2 is a mapping from abstract objects to abstract values. This mapping actually contains the determinism information and is computed during our analysis.
- m_3 is a mapping from all variables to abstract values. This mapping has the following meaning:
  Let $x \in V$ be an arbitrary variable. $m_1(x)$ is the set of abstract objects x could point to. Since a variable either points to an object or is of primitive type and m_1 maps variables of primitive types to themselves, we know that $m_1(x)$ is non-empty.
  Let $possibleAbstractValues(x) := \{m_2(o) \mid o \in m_1(x)\}$.
  m_3 maps variable x to $\sqcup possibleAbstractValues(x)$, i.e. to the least upper bound of the abstract values of all abstract objects that x could point to. m_3 is computed by our determinism analysis using m_2 and m_1.

At the beginning of our abstract execution the abstract state at each program point is as follows: m_2 maps every abstract object to $\bot$ and m_3 maps every variable to $\bot$. m_1 is given by the points-to analysis at each program point and is not changed by our determinism analysis. During abstract execution the abstract state will change from program point to program point according to our abstract transformers, which we define in the next section. After abstract execution, our abstract state will represent an overapproximation of all possible concrete states at each program point.

3.3 Abstract Transformers

Since our tool will analyse Jimple programs, we will define our abstract transformers for the Jimple intermediate representation. But first we will introduce some notation.

3.3.1 Notation

m_1, m_2, m_3 : mappings of an abstract state at a program point (defined as in the previous section)

x : a local variable

v : an abstract value, i.e. an element from S

$m[x \to v]$: denotes a mapping obtained by replacing the abstract value of x in m with v

$stmt$: denotes a statement in Jimple

$[\![stmt]\!](m_2, m_3) = (m_2, m_3[x \to v])$: denotes an abstract transformer, i.e. how the mapping m_3 changes after the effect of $stmt$ takes place. In this case the mapping m_3 will change such that the new abstract value of x will be v.

$defaultValue(t)$: denotes a function that returns the default abstract value for a variable or an object of type t. If, at a program point, a new object is created, we will assign a default abstract value to it. For example, if our abstract interpretation encounters a statement like
`x = new java.util.HashSet`, we will assign $defaultValue(java.util.HashSet)$ to the abstract object with this allocation site. In this case, $defaultValue(java.util.HashSet) = N_oD_c$, since a `HashSet` is nondeterministic in its traversal-order by documentation.
In general, we consider the default abstract value to be N_oD_c for the following types:

- `java.util.Set`
- `java.util.Map`
- `java.util.HashSet`
- `java.util.HashMap`
- `java.util.Hashtable`
- `java.util.IdentityHashMap`
- `java.util.PriorityQueue`

For all other types, including primitive types, we consider the default abstract value to be D_oD_c.

3.3.2 Abstract Transformers

We will now define abstract transformers, case by case, for each possible Jimple statement that we could encounter during abstract interpretation.

We only describe how a statement changes the mappings m_2 and m_3 of an abstract state from one program point to the next program point, according to the control flow graph. m_1 is already given by the points-to analysis and is not modified by our analysis. The overall program state changes by merging the new mappings at each program point with the old mappings at that point. We merge two mappings by taking the least upper bound of the abstract values they map to. Example:

- Let map_1 and map_2 be two mappings and let $map_1(x) = a$ and $map_2(x) = b$ for some abstract values $a, b \in S$ and a variable or object x. Let map_{12} be the merged mapping. Then $map_{12}(x) = a \sqcup b$, where $\sqcup$ is the least upper bound according to our abstract domain definition.

DefinitionStmt

In Jimple a `DefinitionStmt` can be either an `IdentityStmt` or an `AssignStmt`.

- `IdentityStmt`: In Jimple, references to the `this`-object and references to method parameters are explicitly stored in Jimple local variables.
  - $[\![x$ `:= @this:` $type]\!](m_2, m_3) =$

    $(m_2, m_3[x \rightarrow m_2(thisObject)])$,

    where *type* is a placeholder for the type of the `this`-object and *thisObject* is a placeholder for the abstract `this`-object. So after the effect of the statement takes place in our abstract interpretation, the local variable x will be mapped to the current abstract value of the actual `this`-object.
  - $[\![x$ `:= @parameter:` $type]\!](m_2, m_3) =$

    $(m_2, m_3[x \rightarrow \sqcup m_2(parameter)])$,

    where *type* also stands for the type of the parameter and *parameter* stands for the actual parameter. Since *parameter* could point to different abstract objects, we want x to map to the least upper bound of the abstract values of all abstract objects *parameter* could point to.

    Note: If we encounter these statements in the `main` method, we have no previously computed information about the `this`-object and the parameters. In this case we will map x to *defaultValue(type)*, i.e.:
  - $[\![x$ `:= @this:` $type]\!](m_2, m_3) =$

    $(m_2, m_3[x \rightarrow defaultValue(type)])$
  - $[\![x$ `:= @parameter:` $type]\!](m_2, m_3) =$

    $(m_2, m_3[x \rightarrow defaultValue(type)])$

- `AssignStmt`:
  - $[\![x = r]\!](m_2, m_3) = (m_2, m_3[x \to m_3(r)])$, if r is a `Local`
  - $[\![x = const]\!](m_2, m_3) = (m_2, m_3[x \to D_o D_c])$, if $const$ is a `Constant`
  - $[\![x = \texttt{new}\ type]\!](m_2, m_3) =$

    $(m_2[obj \to defaultValue(type)], m_3[x \to defaultValue(type)])$,

    where obj is the abstract object corresponding to this allocation site. In Jimple, Java's object instantiation is split up into two different statements: the first statement simply introduces a Jimple local variable to reference the newly allocated object and the second statement calls the constructor method on that object. For example, the Java statement `hs = new HashSet<String>()` will be represented as two Jimple statements:

    1. `hs = new java.util.HashSet`
    2. `specialinvoke hs.<java.util.HashSet:void <init>()>()`

    In this abstract transformer we only handle the first Jimple statement. The effect of the second statement is handled in `InvokeStmt`.
  - $[\![x = binopExpr(op_1, op_2)]\!](m_2, m_3) =$

    $(m_2, m_3[x \to m_3(op_1) \sqcup m_3(op_2)])$,

    where $binopExpr$ is a binary operation expression using two operands op_1 and op_2. A binary operation is an arithmetical, relational or logical operation on two operands. Since these operations neither introduce any additional nondeterminism nor restore determinism, the result of these operations can only be nondeterministic if the operands are nondeterministic, hence the least upper bound of their abstract values.

    There are cases where this abstract transformer is imprecise, e.g. when we compare two variables that point to the same object:
    Let `hs` be a variable pointing to a `HashSet`. Then x = `(hs == hs)` will make x nondeterministic. But intuitively this expression should be deterministic, since it always evaluates to `true`. So in this example, our abstract transformer is sound but imprecise. However, there are examples where this transformer is sound and precise:
    Let x be an int-typed variable that stores the first element of a `HashSet` of integers. Since x is nondeterministic in its content, a statement like $z = (x + 5)$ should make z nondeterministic as well.

– $[\![x = unopExpr(op_1)]\!](m_2, m_3) = (m_2, m_3[x \to m_3(op_1)])$,

where $unopExpr$ is a unary operation expression using only one operand op_1. A unary operation, e.g. a logical negation, does neither introduce any additional nondeterminism nor restore determinism. Thus, the result of a unary operation is nondeterministic if and only if the operand is nondeterministic.

– $[\![x = (type)\ r]\!](m_2, m_3) = (m_2, m_3[x \to m_3(r)])$,

where r is a `Local` and $(type)$ is an expression that casts r to $type$. Casting neither introduces nondeterminism nor restores determinism, hence we can just propagate the abstract value of the casted operand.

Method Invocation: When there is a method invocation inside an assignment statement, we distinguish between three possible cases:

a) We have a specification of the effect of the method call. This specification will be stored in a JSON file and users will be able to configure it. Here is an example:
Let the assignment statement be of the form
`a = b.foo(c,d)`. A possible specification for the `foo` method could be: If `c` or `d` have the abstract value N_oD_c, then `b` and `a` will both have the abstract value N_oN_c after the method call. So the specification tells what abstract values will abstract objects (that participated in that method call) have after the method call, conditioned on their abstract values before the call. Since `b,c` and `d` could potentially point to different abstract objects, we will apply the specification for each possible combination of abstract objects. We then update each abstract object to the least upper bound of abstract values the specification prescribed to that abstract object.

b) We do not have a specification, but we have an analysable method body. In this case we will continue our analysis interprocedurally. For this, we will prepare the abstract entry-state to start the analysis of the method body with the information we have already computed.
Let the assignment statement be again of the form
`a = b.foo(c,d)`. The entry-state will contain the abstract values of the arguments (`c` and `d`) and the abstract value of the object on which the method was called if it exists (`b` in this case). Other abstract objects and variables from the method body of the called method will be initialised to $\bot$. With this

entry-state we will then analyse the method body. When we reach the return statement of that method body we end our interprocedural analysis and transfer the abstract values of the arguments (`c` and `d`), the base-object (`b`) and the returned value (`a`) back to the caller's site of our analysis.

c) We neither have a specification, nor a method body to analyse. Since Soot generates Jimple code from Java bytecode, not having a method body most probably means that the method is native. In this case we allow users to decide between two options: either to assume the worst case or the best case.
Let the assignment statement be again of the form `a = b.foo(c,d)`. If we assume the worst case, then `a,b,c,d` will all have the abstract value $N_o N_c$ after the method call. If we assume the best-case, then `b,c,d` will preserve their abstract values and `a` will get the abstract value $D_o D_c$ after the method call.

InvokeStmt

- We have already described the possible effects of a method call in the `AssignStmt` case. An `InvokeStmt` is essentially the same, but without an assignment, i.e. not considering the returned value of the method.

Other Statements

- For all other possible statements we propagate the abstract state without changes, i.e.
  - $[\![stmt]\!](m_2, m_3) = (m_2, m_3)$,

    where $stmt \in \{$`IfStmt,GotoStmt,ReturnStmt,ReturnVoidStmt`$\}$.

3.4 Reaching a Fixed Point

To complete the instantiation of abstract interpretation we also need to make sure that the abstract interpretation will terminate, i.e. we will reach a fixed point. A fixed point, in this context, will be a program state that, after applying abstract transformers to it, will not change anymore.

For our instantiation of abstract interpretation it holds that:

- Our abstract domain is finite. This follows trivially from the definition.
- Our abstract domain is a complete lattice. This can easily be seen from the Hasse diagram of our abstract domain.
- Each time after applying abstract transformers we join the resulting program state with the old program state. We join by taking the least upper bound of the abstract values of the variables and objects at a program point.
- We currently do not handle Java programs with recursion, as we already mentioned in section 1.2. Hence, the only potential source of non-termination are loops. But since we abstract objects using their allocation site, there will be finitely many abstract objects, as there are finitely many lines of code in the analysed program. Therefore, the abstract program state can not grow infinitely. We also do not need any widening in our analysis, since our abstract domain is finite and only contains four elements.

From these facts we can follow that our abstract interpretation will terminate.

Chapter 4

Implementation

In this chapter we describe how we implemented our determinism analysis, which challenges we faced and how we solved them. For the development we used the Eclipse IDE 2020-09 [6] with Java SE 8 [7] and Soot version 4.1.0 [13].

4.1 Monotone Framework

Soot provides different implementations of data flow analyses that are based on the monotone framework [18]. The monotone framework is a conceptual framework that allows to instantiate different data flow analyses in five steps:

1. Decide whether the analysis should be a forward or a backward analysis.
    - In our case, the analysis is definitely a forward analysis. To reason about determinism of an object we might need the information about determinism of other objects that were introduced earlier in the program. To implement this we extend Soot's `ForwardBranchedFlowAnalysis<A>` class.
2. Decide what domain should be used to represent the information in the data flow analysis.
    - In our case, the information of the data flow analysis is represented by our abstract program state as described in section 3.2. We implemented a `StateWrapper` class that contains a map from variables and abstract objects to their abstract values. Instances of this class are then propagated by our forward data flow analysis according to the control flow of the program. Our analysis is thus of type `ForwardBranchedFlowAnalysis<StateWrapper>`.

3. Decide how a statement changes the analysis information.
    - Our abstract transformers in section 3.3.2 describe this. In the implementation we override the `flowThrough()` method of the `ForwardBranchedFlowAnalysis` class. In this method we define how the outflowing analysis information changes, based on the inflowing analysis information and the current statement.
4. Decide how inflowing analysis information should be merged.
    - When there are different inflowing abstract states, we need to decide how to combine them. In a must-analysis one usually takes the intersection and in a may-analysis the union. In our case we want the union, since we overapproximate all possible program states at each program point. In the implementation we override the `merge()` method of the `ForwardBranchedFlowAnalysis` class, such that we return a `StateWrapper` that joins the two inflowing `StateWrappers`. In the joined `StateWrapper` the abstract value of each abstract object is the least upper bound of its abstract values from the two inflowing `StateWrappers`.
5. Decide how the analysis should be initialised.
    - We initialise our analysis with a `StateWrapper entryState`. Its mapping maps all the variables and abstract objects to $\bot$ in the very beginning of the analysis. When we encounter a method call and we have the body of that method, we can start a new analysis recursively and initialise its `entryState` with the already computed information.

4.2 Generating Nondeterminism Warnings

After implementing the actual determinism analysis, we can run it on example programs and get the overapproximated determinism information at each program point. Since we are interested in warning users about potentially nondeterministic output, we can check if the output was created using a nondeterministic argument. As already mentioned in Definition 2.4, we consider `System.out.println()` to be our canonical output function.

In our warnings we tell the users at which line in the source code a potentially nondeterministic output could happen, what the abstract value of the outputted argument is and at what line in the source code this argument became nondeterministic, i.e. where the last bad change to its abstract value happened. Here is an example:

```
1  public class Example {
2      public static void main(String[] args){
3          Set<String> hs = new HashSet<String>();
4
5          hs.add(" a ");
6          hs.add(" b ");
7
8          List<String> list = new LinkedList<String>();
9
10         list.add(" c ");
11         list.add(hs.iterator().next());
12         list.add(" d ");
13
14         System.out.println(list);
15     }
16 }
```

Listing 4.1: A simple program that has potentially nondeterministic output at line 14. The `list` is nondeterministic in its content, since we added a nondeterministic element to it at line 11.

```
1 WARNING: Your output in method
2 <Example: void main(java.lang.String[])>
3 at line 14 is potentially nondeterministic.
4 Argument of println() is N_N
5 Last bad change happened at line 11
```

Listing 4.2: Warning created by our tool after analysing the program from Listing 4.1

4.3 Programs outside our Java Subset

As we stated in section 1.2, our tool only analyses Java programs from our predefined Java subset. Our implementation, however, does not explicitly reject Java programs outside that subset. In the following, we briefly describe how our tool will behave when the analysed program does not fulfill some constraints:

- If the analysed program contains a recursive function of which we do not have a specification, but we have an analysable method body, our tool will try to analyse it and will not terminate. If we have a specification for such a function, our tool will just normally continue the analysis as if it was an ordinary non-recursive function. If we neither have a specification nor an analysable method body of the recursive function, our tool will also continue the analysis assuming the most general possible effect of that function, as it does for non-recursive functions.

- If the analysed program contains a dynamically-dispatched method call, our tool will assume the most general possible effect of this method call, as if we had no specification and no analysable method body. Hence, we will remain sound but imprecise in these cases.
- If the analysed program contains arrays or fields, our tool will not include them into the abstract state and will not track their abstract values.
- Other unhandled features will be ignored during analysis.

4.4 Challenges

In this section we briefly describe what challenges we faced during the development of our tool and how we dealt with them.

1. The first challenge was to get familiar with the Soot framework. Soot is a powerful and complex framework and in order to understand how it works I benefitted from various helpful resources, some of which I will list here:
   - Soot Survivor's Guide [11]. This source really helped me to get an overview of how Soot works and helped me to get started.
   - PLDI03 Soot tutorial [9]. This is the official Soot tutorial given by the developers at the PLDI conference.
   - Soot Wiki [14]. Also helps to get a first overview and links to other useful sources (some of which are unfortunately outdated).
   - Soot Mailing List [12] together with the issues section on Soot's GitHub repository [10] help to find an answer by either searching already asked questions or asking a new question.
2. When creating warnings for the users we initially wanted to tell users the original name of the nondeterministic argument of the outputting function. But despite setting the Soot option `use-original-names` to `true` during Jimple code creation, only a few variables preserved their original source code names. The reason for this is that Jimple is based on a three-address-code. Thus, generating Jimple code will inevitably break up complex expressions and introduce new Jimple variables to hold the intermediate results. These Jimple variables will also be given to the outputting function as arguments and there is no mapping to the original names. This was a problem, since we wanted to create a useful warning for the users. Fortunately, setting the `keep-line-number` option to `true` allows to get the source code line number of every Jimple statement, which suffices to locate problematic output statements.

Chapter 5

Evaluation

In this chapter we describe how our tool performs on example programs of different complexity. Since we can only analyse a subset of possible Java programs, we created our own test suite. We manually analysed these programs to find potentially nondeterministic output, according to our definitions, and compared the results of the tool with our expected results.

5.1 Test Suite

Example programs from our test suite are not complete applications from the real world, but more like different unit tests that cover different features of our Java subset and different cases of possible nondeterministic behaviour. We start our analysis in the `main` method of the entry class of a program and analyse methods of other classes when they are used in that program. We also tried to cover the most important collection implementations from the Java collections framework, which are:

- `List`: `LinkedList`, `ArrayList`, `Stack`, `Vector`
- `Set`: `HashSet`, `LinkedHashSet`, `TreeSet`
- `Queue`: `PriorityQueue`
  - `Deque`: `ArrayDeque`
- `Map`: `HashMap`, `Hashtable`, `LinkedHashMap`, `TreeMap`

Table 5.1 shows the results of our analysis for each example.

Legend for the table shown next:

- Ex = Identifying number of the example
- LoC = Number of lines in the source code of the corresponding example
- Stmts = Number of Jimple statements that were visited during our determinism analysis. Our tool performs abstract interpretation on the Jimple code of the analysed example and Stmts shows how many steps our abstract interpretation had. Hence, if a statement is visited multiple times it is also counted multiple times.
- S_1 = Tells if our analysis was sound on that example, if we assumed the worst case for the effect of unspecified and non-analysable methods.
- P_1 = Tells if our analysis was precise on that example, if we assumed the worst case for the effect of unspecified and non-analysable methods.
- S_2 = Tells if our analysis was sound on that example, if we assumed the best case for the effect of unspecified and non-analysable methods.
- P_2 = Tells if our analysis was precise on that example, if we assumed the best case for the effect of unspecified and non-analysable methods.
- T = Time needed to analyse this example assuming the worst case for the effect of unspecified and non-analysable methods (in seconds).

Table 5.1: Results

Ex	**LoC**	**Stmts**	**S_1**	**P_1**	**S_2**	**P_2**	**T**
1	26	9	yes	yes	yes	yes	7 s
2	35	17	yes	yes	yes	yes	7.6 s
3	27	9	yes	yes	yes	yes	8 s
4	40	17	yes	yes	yes	yes	7.2 s
5	37	12	yes	yes	yes	yes	7.1 s
6	33	11	yes	yes	yes	yes	7.4 s
7	31	22	yes	no	yes	yes	7.5 s
8	74	45	yes	no	yes	no	7.5 s
9	36	22	yes	yes	yes	yes	7.3 s
10	32	17	yes	yes	yes	yes	6.7 s
11	39	70	yes	no	yes	yes	6.9 s
12	43	24	yes	yes	yes	yes	6.9 s
13	35	19	yes	no	yes	no	7 s
14	86	194	yes	no	yes	yes	7.8 s
15	65	83	yes	no	yes	yes	7.3 s
16	58	43	yes	yes	yes	yes	5.9 s
17	35	22	yes	yes	yes	yes	6 s
18	66	29	yes	yes	yes	yes	5.6 s
19	43	18	yes	yes	yes	yes	5.5 s
20	37	18	yes	yes	yes	yes	5.7 s
21	40	34	yes	yes	yes	yes	6.9 s
22	34	24	yes	yes	yes	yes	6.8 s
23	52	25	yes	yes	yes	yes	7.2 s

Code coverage: Running our tool on all these examples resulted in a 93.8% code coverage of our determinism analysis code.

5.2 Precision

In this section we want to show one example where our tool is precise and one example where our tool is imprecise in order to provide a quick glance into what our tool is capable and not capable of doing.

The following is an example where our tool is precise:

```
1  public class Example {
2      public static void main(String[] args){
3          Set<String> s;
4
5          Scanner in = new Scanner(System.in);
6
7          if(in.nextInt() > 5) {
8           s = new LinkedHashSet<String>();
9          } else {
10          s = new TreeSet<String>();
11         }
12
13         s.add(" a ");
14         s.add(" b ");
15
16         foo(s);
17     }
18     public static void foo(Set<String> s) {
19       System.out.println(s);
20    }
21 }
```

Listing 5.1: Example program for which our tool is precise and does not create a warning

Thanks to the points-to analysis our tool can figure out that the `Set s` is either a `LinkedHashSet` or a `TreeSet`, both of which are deterministic in the traversal-order. Hence, the printed `Set s` is deterministic in its traversal-order and no warning is created at line 19 of the program in Listing 5.1. Without a reasonably precise points-to analysis, we would have assumed the worst case and would produce a warning at line 19, since a `Set` could also be a `HashSet` and thus nondeterministic.

The following is an example where our tool is imprecise, but still sound:

```
1  Set<String> hs = new HashSet<String>();
2
3  hs.add(" a ");
4  hs.add(" b ");
5
6  LinkedList<String> list = new LinkedList<String>();
7
8  list.add(hs.iterator().next());
9
10 if(list.get(0).equals(" a ")) {
11     System.out.println(list.get(0));
12 }
```

Listing 5.2: Example program for which our tool is imprecise

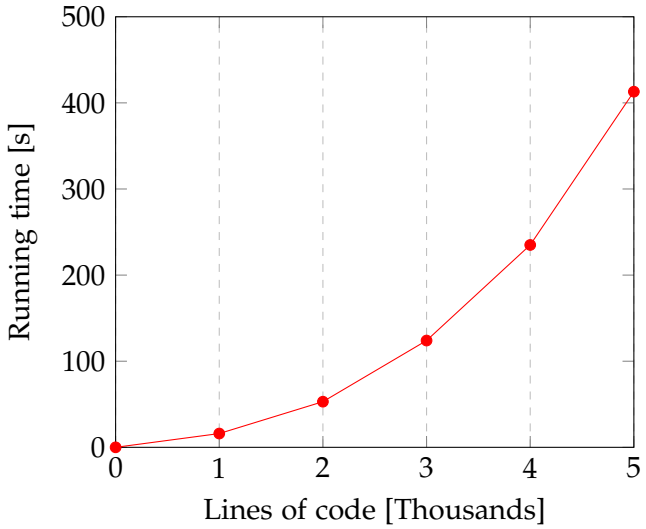
Our tool will produce a warning for the program from Listing 5.2. It will say that a nondeterministic output is possible at line 11, since a nondeterministic element was added to the `list` at line 8. However, according to our definitions, the output at line 11 is deterministic because of the if-condition. Since our tool does not track the path conditions and does not track the possible content of the collections, it will be imprecise in these cases.

5.3 Performance

To evaluate performance of our tool we ran it on larger Java programs of different size and measured the running time. We represent the size of a Java program by the number of lines in the corresponding source code.

The program on which we tested the performance consists of a nested loop with some if-statements, collection operations and a method call. We then increased the number of lines of code by repeatedly appending this nested loop to the end of the program. The following diagram demonstrates the approximated results obtained by running the tool on Microsoft Windows 10 on a 4-core Intel i7-8550U CPU with 1.80 GHz.

Lines of code	Running time
1000	16 s
2000	52 s
3000	124 s
4000	234 s
5000	413 s



As we can see from the diagram, the running time roughly doubles with each additional thousand lines of code. This happens because the tool keeps track of determinism of all variables and abstract objects of the program. The more distinct variables and abstract objects a program has, the larger are the analysis data-structures and the longer it takes to complete the analysis. Nevertheless, the tool still takes a reasonable amount of time to analyse even large programs.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis we developed a tool that helps programmers to discover potentially nondeterministic output in their Java programs using static analysis. The first challenge was to develop our own definitions of determinism in order to pinpoint the underlying problem. Having defined the problem, we decided to use the abstract interpretation technique to perform our determinism analysis. For this, we have defined the abstract domain and abstract transformers of the analysis. In order to keep the focus of this project on developing a sound determinism analysis and not get lost in numerous features of the Java language we limited our analysis to a subset of Java programs. We then implemented our analysis using the Soot framework and evaluated it on example programs of different complexity.

6.2 Future Work

In general, we see two possible ways of improvement: increasing the expressiveness of the determinism analysis, and improving the usability of the developed tool.

6.2.1 Expressiveness

- Cover arrays and fields. This would be very useful, since a lot of real-world Java programs use arrays and fields. However, it will be a challenging task, since it would include reasoning about the heap of the program. One would also need to consider arrays of collections, collections as fields of some objects and potential aliasing.
- Cover recursive functions. This would also be useful, since a lot of real-world Java programs use recursive functions. To handle recursive

functions in our analysis one would need to somehow break the cycles. One option could be to treat recursive functions as a loop and only continue analysing further down the recursion if the calling context changes. To terminate the analysis, in case the calling context keeps changing, one could perform some sort of widening, as it is done when analysing loops.

- Implement the analysis of dynamically-dispatched methods. One idea could be to analyse the underlying bootstrap method, i.e. the method in which it is decided which concrete implementation of a method will be called. This obviously only works if such a bootstrap method is available. Another option would be to analyse all possible concrete implementations of the dynamically-dispatched method and to over-approximate its effect. This option, however, is potentially very costly performance-wise.
- Include reasoning about determinism of branch conditions. This would make the analysis more precise and help detect potentially nondeterministic overall output of the program. The following example illustrates such a case:

```
1  //HashSet with elements 1,2,3,4,5
2  HashSet<Integer> hs = new
       HashSet<Integer>(Arrays.asList(1,2,3,4,5));
3
4  int n = hs.iterator().next();
5
6  LinkedList<String> list = Arrays.asList("a", "b", "c");
7
8  if(n > 3) {
9     System.out.println(list);
10 }
```

Listing 6.1: Revisited example from Listing 2.4

Recall that our analysis does not produce a warning for this example program, since the output at line 9 only uses a deterministic collection instance `list`. Tracking determinism of a branch condition would allow to also warn users that the overall output of the program could vary in different executions with the same input.

- Cover exceptions.

6.2.2 Usability

- Allow users to configure which types of output should be checked for determinism and extend the tool accordingly. It could be that the user does not care about determinism of the printed output, but does care about determinism of the data sent over a network or written into a database. The tool should then only produce warnings for outputs of specified types.
- Introduce special source-code annotations to allow users to explicitly change the abstract state of the determinism analysis at a particular point in the program. This could improve the precision of the analysis in cases where, e.g. the user applied some custom sorting algorithm on a collection making the traversal-order deterministic, but our analysis could not figure this out.
- Implement an IDE plugin to make the tool more user-friendly and highlight potentially nondeterministic output points in the source code.

Bibliography

[1] Class HashSet. https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html.
(Accessed: 14.03.2021).

[2] Class LinkedHashSet. https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html.
(Accessed: 14.03.2021).

[3] Class PriorityQueue. https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html#iterator--.
(Accessed: 14.03.2021).

[4] Collections Framework Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html.
(Accessed: 14.03.2021).

[5] Default hashCode(). https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--.
(Accessed: 14.03.2021).

[6] Eclipse Version. https://www.eclipse.org/downloads/packages/release/2020-09/r.
(Accessed: 14.03.2021).

[7] Java Version. https://docs.oracle.com/javase/8/docs/.
(Accessed: 14.03.2021).

[8] Jimple. https://en.wikipedia.org/wiki/Soot_(software)#Jimple.
(Accessed: 15.03.2021).

[9] PLDI03 Tutorial. https://www.sable.mcgill.ca/soot/tutorial/pldi03/tutorial.pdf.
(Accessed: 14.03.2021).

[10] Soot. https://github.com/soot-oss/soot.
(Accessed: 14.03.2021).

[11] Soot guide. https://www.brics.dk/SootGuide/sootsurvivorsguide.pdf.
(Accessed: 14.03.2021).

[12] Soot Mailing List. https://groups.google.com/g/soot-list.
(Accessed: 14.03.2021).

[13] Soot Version. https://soot-build.cs.uni-paderborn.de/public/origin/master/soot/soot-master/4.1.0/.
(Accessed: 14.03.2021).

[14] Soot Wiki. https://github.com/soot-oss/soot/wiki.
(Accessed: 14.03.2021).

[15] Cousot, Patrick; Cousot, Radhia (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. https://www.di.ens.fr/~cousot/publications.www/CousotCousot-POPL-77-ACM-p238--252-1977.pdf.
(Accessed: 27.03.2021).

[16] Cousot, Patrick; Cousot, Radhia (1992). Comparing the Galois Connection and Widening / Narrowing Approaches to Abstract Interpretation. https://www.di.ens.fr/~cousot/COUSOTpapers/publications.www/CousotCousot-PLILP-92-LNCS-n631-p269--295-1992.pdf.
(Accessed: 27.03.2021).

[17] Jonathan Aldrich. Lecture Notes: Interprocedural Analysis. https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/resources/interprocedural.pdf.
(Accessed: 16.03.2021).

[18] Kam, John B.; Ullman, Jeffrey D. (1977). Monotone Data Flow Analysis Frameworks. https://homes.luddy.indiana.edu/achauhan/Teaching/B629/2006-Fall/CourseMaterial/1977-acta-kam-monotone.pdf.
(Accessed: 31.03.2021).

[19] Ondřej Lhoták. Spark : a flexible points-to analysis framework for Java. Master's thesis, McGill University, 2003.