Reasoning about Nondeterministic Collections

Bachelor Thesis Project Description

Bogdan Gadzhylov
Supervised by Dr. Malte Schwerhoff

October 12, 2020

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. A program is said to be deterministic, if for a given state and a fixed input it always returns the same output. This property is often desired at a point in the program where the program outputs some data. It could be very confusing for users, if rerunning the program without changing any input would result in different output.

High-level programming languages usually provide a collections framework for dealing with multiple objects through a single unit. Those frameworks are very important because they reduce the programming effort, but they could potentially introduce nondeterminism. One example for a framework that can lead to nondeterministic output is the Java collections framework.

The Java collections framework [3] is a commonly used framework, which allows to represent and manipulate groups of objects, independently of implementation details. 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 predictable traversal-order, whereas the `HashSet` implementation does not guarantee that the traversal-order will remain constant over time.
In larger programs it is common to instantiate a collection of a particular type, e.g. `HashSet`, and pass this instance around using its abstract collection type, e.g. `Set` in this case. This would mean that the user of this collection does not know whether or not deterministic traversal-order is guaranteed.

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.

The aim of this project is to research techniques for reasoning about the use of nondeterministic collections, and develop a tool that can find potentially nondeterministic collection traversals and warn users if those can lead to nondeterministic outputs. Users can then decide to use different collection types, or sort data at specific program points, based on the tool's results.

2 Different types of determinism

It is important to distinguish between two types of determinism in collections: traversal-order determinism and content determinism.
Here is an intuitive definition:

- A collection is deterministic in its **traversal-order** if in multiple executions of the same program with the same inputs an order-dependent operation on this collection always has the same observable effect. Typical order-dependent operations are: the foreach-iteration or returning the first element of the collection. Otherwise the collection is nondeterministic in its traversal-order.
- A collection is deterministic in its **content** if in multiple executions of the same program with the same inputs at a particular program point the collection always has the same content. Otherwise the collection is nondeterministic in its content. Content nondeterminism can only occur because of nondeterministic controll flow or because of nondeterministic traversal-order of another collection.

Note: In this document, we only consider determinism in the context of multiple executions of the same program with the same inputs at the same point in the program.

Given these two types of determinism, a collection could be:

- deterministic in its traversal-order and content
- nondeterministic in its traversal-order and deterministic in its content
- deterministic in its traversal-order and nondeterministic in its content
- nondeterministic in its traversal-order and content

Traversal-order nondeterminism Listing 1 shows an occurrence of traversal-order nondeterminism.

```
1 //initiate a HashSet of Strings
2 Set<String> hs = new HashSet<String>();
3
4 //fill the set
5 hs.add(" a ");
6 hs.add(" b ");
7 hs.add(" c ");
```

```
8
9  //call the userFunction with a HashSet argument
10 userFunction(hs);
11 ...
12 //a function that abstracts the implementation of a Set from the user
13 public static void userFunction(Set<String> set) {
14    for(String s : set) {
15       System.out.print(s);
16    }
17 }
```

Listing 1: Printing each element of a `HashSet`

Since the `HashSet` implementation does not guarantee a deterministic traversal-order, it could happen that during multiple executions of the same program with the same inputs we traverse the elements in a different order, which would lead to nondeterministic output.
In this example, at the point where the program creates output, our collection is nondeterministic in its traversal-order, but deterministic in its content.

Note: If we used a `LinkedHashSet` implementation, the collection would be deterministic in its traversal-order and content. The problem is that the concrete type of the collection is not known to the user.
The `userFunction(Set<String> set)` abstracts the concrete type of the collection through an implicit upcast from a `HashSet` to a `Set`.

Content nondeterminism Listing 2 shows an occurrence of content nondeterminism

```
1  //Set hs initialised as in Listing 1 with the same elements
2
3  //creating a list of Strings
4  List<String> list = new LinkedList<String>();
5
6  //adding the first element of the HashSet hs to the list
7  list.add(hs.iterator().next());
8
9  //printing the only element of the list
10 System.out.print(list.get(0));
```

Listing 2: Nondeterministic traversal-order of one collection leads to nondeterministic content of another collection

Since we do not know which element will be added to the list, due to nondeterministic traversal-order of the `HashSet`, we do not know the content of the list. Thus, at the point where the program creates output, the `LinkedList` collection is nondeterministic in its content, and the output can therefore vary across multiple executions.

Restoring determinism It is also important to mention that determinism could sometimes be restored at some point in the program, even if nondeterministic behaviour was possible earlier:

- Traversal-order determinism could be restored by explicitly converting the collection to a list and sorting it
- Content determinism could be restored by explicitly setting the content of a collection to deterministic values, but this seems unrealistic in practice

3 Core Goals

The main goal of the project is to adapt the abstract interpretation technique in order to develop a tool that analyses Java programs and warns about potentially nondeterministic output.

3.1 Investigation

(i) Investigate the aforementioned, intuitive definitions of traversal-order determinism and content determinism. Investigate the relation between these two types of (non)determinism and formalise the final definitions in a more rigorous way.

(ii) Investigate which collection implementations could lead to nondeterministic output

(iii) Investigate at a finer granularity, which operations on those collections have what effect on determinism

(iv) Investigate which static analysis framework for Java programs to use, e.g. Soot, Wala

(v) Adapt the abstract interpretation technique, i.e. define the abstract domain and the effect of possible statements and expressions on the abstract state of the program both for:

- traversal-order determinism and
- content determinism

3.2 Implementation

(i) Implement a concrete-type analysis. A concrete-type analysis allows us to statically determine the set of possible concrete (i.e. dynamic) types, which a collection object could have during the actual program execution at a particular program point

(ii) Implement a traversal-order determinism analysis using the static analysis framework chosen in 3.1(iii) with abstract domain and abstract transformers defined in 3.1(iv)

(iii) Implement a content determinism analysis using the static analysis framework chosen in 3.1(iii) with abstract domain and abstract transformers defined in 3.1(iv)

(iv) Implement warnings that report potential nondeterministic output to the user

4 Extension Goals

(i) Evaluate expressiveness, precision and usability of the implemented solution by testing the tool on example programs of different complexity

(ii) Extend the tool to allow users to configure which output should be considered by our determinism analysis, e.g. printing something, sending data over a network, storing something in a database etc.

(iii) Extend the tool to support special annotations that 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 it out.

(iv) Extend the tool to propose the users a solution to deal with the possible nondeterministic output, i.e. sort the collection explicitly or not to use some operations on it

(v) Extend the tool to decide, whether a branch condition can yield different results for the same input

References

[1] *Class HashSet*. URL: `https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html`. (accessed: 11.10.2020).

[2] *Class LinkedHashSet*. URL: `https://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashSet.html`. (accessed: 11.10.2020).

[3] *Collections Framework Overview*. URL: `https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html`. (accessed: 07.10.2020).