Exception flow analysis for Kotlin

Author: Ing. Filip Dolník

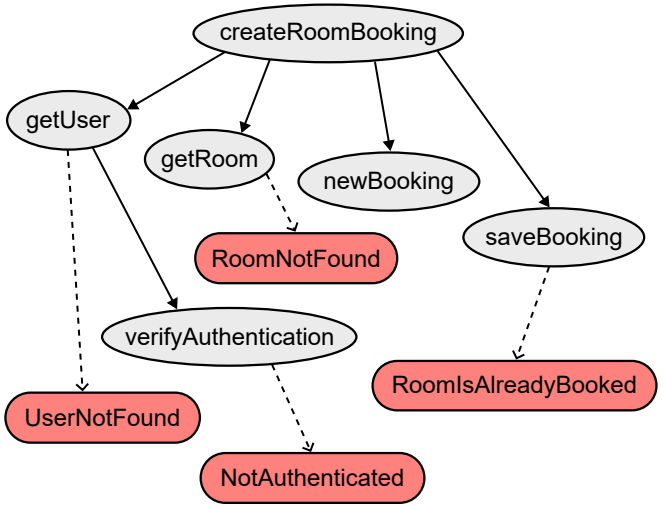
Supervisor: Ing. Jiří Hunka

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Motivation

Exception handling is challenging. Done improperly can cause many problems – from bad UX to security vulnerabilities. It is especially difficult to handle exceptions thrown by APIs of other systems. Exceptions in APIs are usually not correctly documented because maintaining such documentation is time-consuming and error-prone.

Problem demonstration



The graph above shows a sequence of function calls. It demonstrates the complexity of determining what exceptions can be thrown by a function. It is easy to spot exceptions that are thrown directly. However, some exceptions are produced by calling other functions. These functions are spread across the program – making it laborious to find the exceptions.

State of the art

Exception handling is a relatively well-researched problem. Some existing solutions use static analysis of exception flow to generate the required exception documentation.

The problem is that existing research focused on the past generation of programming languages like Java or ML. More modern programming languages, like Kotlin, mix procedural, object-oriented, and functional programming styles. Consequently, modern languages have many complex features. And while the core concepts remain the same, the static analysis must be significantly modified to support those features.

Method

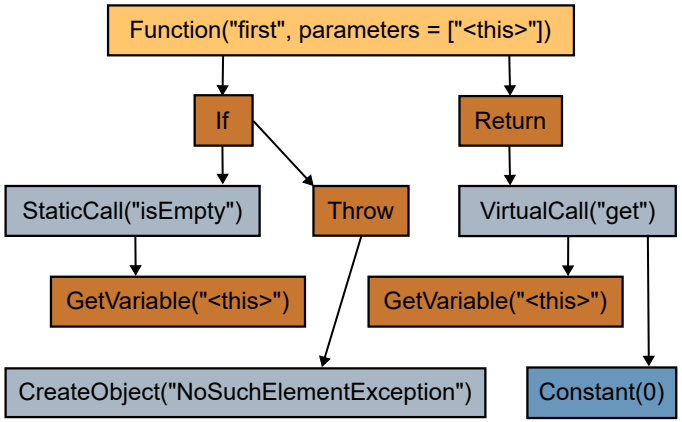
The designed static analysis utilizes an approach called abstract interpretation. Abstract interpretation creates a sound approximation of the analyzed program's semantics. The approximation is obtained by partially executing the program - by omitting unimportant details.

The design is based on a combination of existing algorithms - with some additional changes. The main novelty is in the source code preprocessing phase. During this phase, the complex Kotlin features are lowered to (represented as) simpler features that the abstract interpretation understands.

Source code preprocessing

This phase transforms the source code into an intermediate representation (IR). IR is an in-memory data structure that holds the program's semantics. The Kotlin compiler is used to parse the source code into Kotlin IR. Kotlin IR is then converted into a custom IR designed specifically for this analysis.

```
fun <T> List<T>.first(): T {
    if (isEmpty())
        throw NoSuchElementException("List is empty.")
    return this[0]
}
```



Main contributions

1. Design of static analysis for exception propagation
2. Prototype implementation of the core concepts
3. Acceptance tests covering many Kotlin features
4. Documentation of a significant part of Kotlin IR