Project Overview

Product Name: ONI Code Visualization
Team Name: GraalVM UCSC
Date: April 11 2020

Codebase link: https://github.com/chthota9/GraalVM-UCSC

What is GraalVM?:

- GraalVm is a virtual machine developed by Oracle in order to support various programming languages and execution modes(i.e ahead-of-time compilation) for Java applications in cloud and hybrid environments.
- For the purpose of this project, you will be working with one of GraalVM's components: Native image, a ahead-of-time compilation.
- Links Needed:
    - GraalVm Overview:
        - https://www.graalvm.org/
        - https://www.oracle.com/technetwork/graalvm/overview/index.html
    - GraalVM Native Image: https://www.graalvm.org/docs/reference-manual/native-image/
    - GraalVM Documentation:https://www.graalvm.org/docs/
    - GraalVM Codebase:https://github.com/oracle/graal

Why is GraalVM Native Image useful?

- With GraalVM's ahead-of-time compilation, GraalVM Native Image can compile Java code to a standalone executable called a native image. This native image provides the information we need for our plugin.
- Why GraalVM : https://www.graalvm.org/docs/why-graal/

Goals:

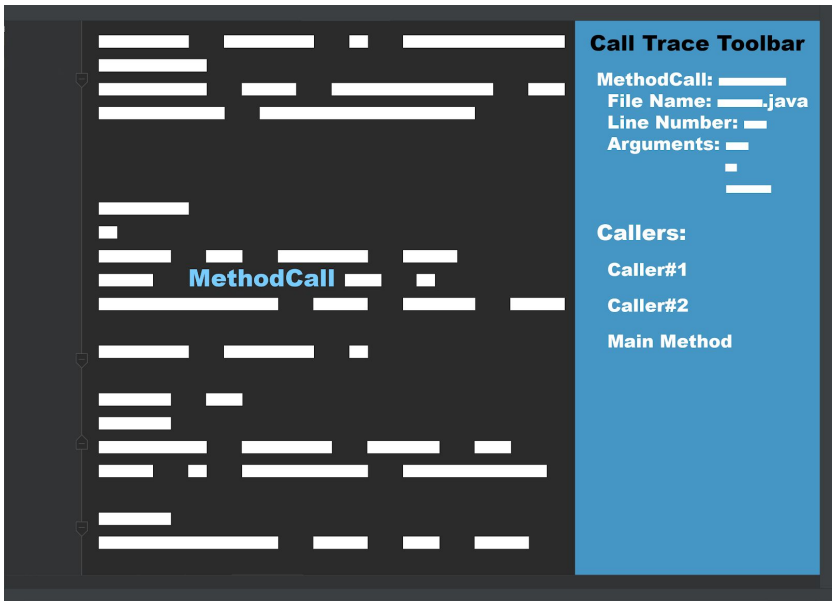
Overall Goal: Create an IDE plugin that visualizes a static analysis report created by Oracle's Native Image of a program and its methods.

Goals Accomplished:

- Manipulating the Static Analysis Report
    - Modify Oracle's CallTreePrinter.java to print out its static analysis report into JSON format (the java package used is linked in the Critical Resource section)
    - Extract only the relevant information(Line Number, File Name, Callees)
        - Functions and Terms to know:

        - bigbang.getUniverse().getMethods() : returns a list of AnalysisMethod objects that consist of methods used in the program
        - AnalysisMethod.java: consists of the necessary functions to return a method's id,name, parameter, return type, etc.
        - StackTraceElement: consists of information of a method's file name and line number location within the file.
        - Bci: bytecode object that helps you identify and trace a method's call path
        - FrameState: beginning of a method's call trace (bci will start at 0), you can use this to retrieve the callers and callees of a method
    - More information on how to set up and generate the report can be found in the User Manual
- Recognize method calls in editor
    - Use PsiElement + PsiTree to recognize where in code a method is called
    - Remove false positives such as variables with same name as methods
- Opening a JSON file via plugin
- Develop a Plugin that mimics the Mockup
    - Relevant data is displayed in a side panel upon selecting "Display Call Trace Visualization".

Mockup:

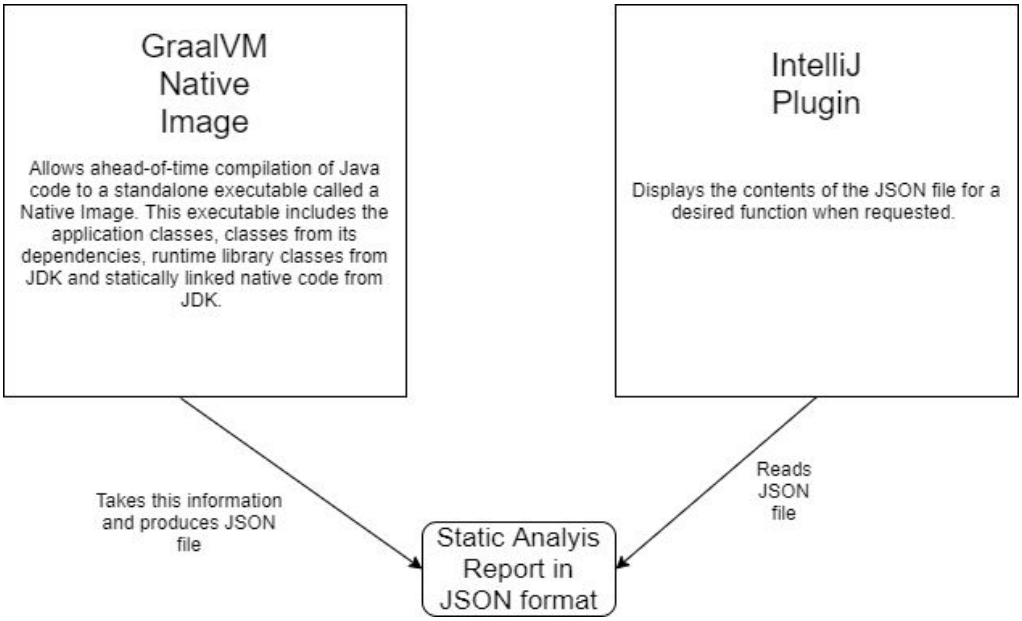


- The static report must be generated outside of the IDE plugin.
    - This was a design decision made by our sponsor.

- The information is displayed in the same window, on a sidebar in the right hand side of the IDE window.
    - We decided to display this in the same window rather than a pop up window as it was easier to view the information and the code side by side.

Current Status:

- Currently, the user must run the static analysis report prior to launching the IDE.
    - CallTreePrinter outputs relevant data to a report JSON file.
        - Data is generated from GraalVM native image
        - Information includes application classes, classes from dependencies etc.
    - The plugin displays the call print trace information from the report JSON file.
        - The IntelliJ plugin loads in the JSON report file from a hard coded file path upon launching Intellij.
        - Within the editor, the user can right click on a function and display relevant information (File name of function definition, line number of function definition, and arguments supplied to the function).
        - Right clicking on a non-function gives the user an error message.



Demo: https://drive.google.com/open?id=1pa3yaWcPOWI2qyB5vilOpAUkoOWGgUSM

Obstacles:

- Oracle's GraalVM software is currently only supported up to Java 1.8, so any newer versions of Java are not compatible.
- Due to the time constraints of the quarter, the plugin is only developed for the IntelliJ IDE.

- There were difficulties with importing libraries to the GraalVM project.
    - Libraries were unrecognized after adding to the build path.
    - Our workaround was to manually copy over the java files and classes into a file so that it could be recognized.
        - We put it into com.oracle.graal.pointsto package that was already being recognized in the path
- Debugging GraalVM's codebase was difficult without a debug tool.
    - To debug, we used breakpoints, print statements, and variable monitoring.
- More working problems can be found in our " Working Prototype and Known Problems Report"

Critical Resources and Technologies:

- https://www.jetbrains.org/intellij/sdk/docs/basics/getting_started.html
    - A general overview of plugin development in Intellij
- https://www.jetbrains.com/help/idea/tool-windows.html
    - A detailed article on working with tool windows, which are essential to plugin development in Intellij
- https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_elements.html
    - Gain an understanding of the PsiElement that allows the recognition of in-editor elements.
    - See also: PsiTree
- https://github.com/stleary/JSON-java
    - JSON- java package used to output the static analysis report into JSON format

End Goal:

- Update the Report JSON in real time within the plugin.
- Add support for other IDEs.
- Create a Github Extension with the same functionality as the IDE plugin.
- Consult the sponsor for more information on what their vision for the final product of the project will be, as they left it open-ended at the end of last quarter.