Richard White

CS 6315 – Automated Verification

Project check-in

Exploration of Static Analysis and Dynamic Symbolic Execution for the Python programming language.

The aim of my project is to explore the existing tools available and propose some base/simplified implementation for dynamic symbolic execution of python functions, for the purpose of automated test case generation. Dynamic Symbolic execution has also proved to be a useful technique in identifying errors/bugs in software.

Challenges:

Python’s ‘duck’ type system, which is strong and dynamic (vs. static) has made these sorts of tools which do exist for languages such as C, C++, Java, C#, trickier to implement in Python.

To help address this concern, I’m focused on a particular sub-set of python syntax using type-inference to assist with static reasoning:

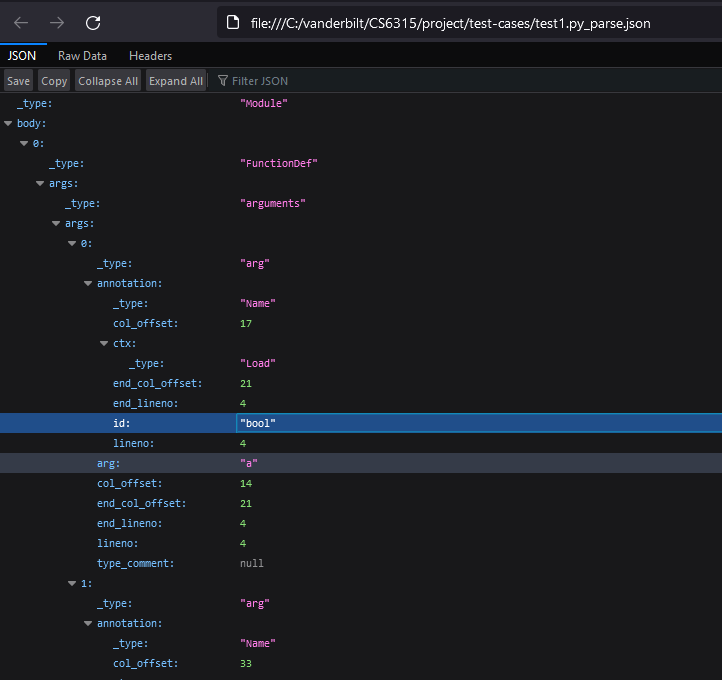
Once I’ve worked out solutions for bool inputs, I want to build out for integers, and then maybe work on floats or more complex structures such as lists/dicts.

A screenshot of a computer

Description automatically generated with medium confidence

Using python’s built-in AST – Abstract Syntax Tree, we can parse this function into a data-structure for interpretation:

From here, we can analyze the AST structure, and parse out the Input variable identifiers and types



These variables won’t get assigned concrete values yet (ignoring default values given): they will remain a ‘set’ of allowable outcomes.

Next, we traverse the body of the function looking for branching logic, such as “If”-like expressions:

A screenshot of a computer

Description automatically generated

From here, we can parse the “test”/condition of the first “IF” statement.  
In this case, its represented as the logical conjunction of A and B.

The next step is to construct concrete values of A and B such that we reach the branch:

In our sample, a = True, b = True.

Once identifying this ‘interesting’ input set that leads us down this branch, begin construction of our Equivalence class of inputs, which will (hopefully) reduce the symbolic input space into a concrete subset.

Once the inputs identified, aim to construct/auto-generate some Unit Tests based on the input.

Similarly, this method can also be used to identifying “dead’ branches that are inaccessible (like some linters do).

Additional Research in this area used to help guide my project:

<https://www.cs.cmu.edu/~emc/papers/Conference%20Papers/Finding%20Errors%20in%20Python%20Programs%20Using%20Dynamic%20Symbolic%20Excecution.pdf>

<https://www.st.cs.uni-saarland.de/edu/automatedtestingverification12/slides/11-DynamicSymbolicExecution.pdf>

This slide below nicely summarizes the general idea of the algorithm I’m working on to process the generated AST to identify unique test cases.

Diagram

Description automatically generated