Aniketh Aatipamula
EECS 700: Static Analysis

Assignment 1 - Extend Prover

Starting with arrays, this part of the prover was fairly straightforward to implement. I began by parsing out subscripts and returning a select node with the array id and index. I then had to override the Assign visitor to intercept any subscript nodes and return a tastore node with the correct arguments.

For the 'tastore' WP I just implemented the z3 objects for Arrays by evaluating the array id as an Array(Int, Int) type and evaluating the index and expression to store as z3 predicates as well. Since we are storing we can use z3's Store operation. The weakest precondition for array writes, as given in the assignment guidelines, is just the post condition where the array is substituted with the store operation on the array, which concludes the WP for tastore.

The operation for 'select' on the other hand does not modify the program state and therefore doesn't modify the WP. In this case I just modified expr_to_z3 as array accesses are expressions and returned a Select operation on the array and index. All arrays were typed as Int to Int arrays as suggested in the assignment guidelines and therefore it is assumed that all arrays have Int values.

Procedures were fairly more complex to handle as changes to the parser had to be made in order to accommodate for definitions and calls. I started with the Function definition visitor node. I gathered all the params, and looped through the body to find any instances of a requires and and ensures. My code in theory can accept more than one requires/ensures, but only grabs the last requires and ensures for every procedure definition. I also had to grab any instances of assignment in the procedure and add the modified variable to the modifies list for use when proving the WP. This was done for all types of assignment including calls and array stores. All other nodes in the body (including assignments) were just appended to the procedure body. Generic calls were added to the Call node visitor and visitors for function arguments and returns were also fairly straightforward. I did add an assertion that requires and ensures must not be null (None) to make sure that there was at least one of each in the procedure body. A global map of procedures was updated with the parsed definition of the procedure.

Moving on to the prover I had to update the weakest precondition function to accept the mapping of procedures to definitions. A procedure body's WP, as defined in the assignment guidelines, is just: requires implies the WP of the body with the ensures as its postcondition. This was a fairly straightforward implementation, but in order for the requires to be true all old(x) variables in ensures had to be augmented with the condition that they are the same as in the pre-state. This was done with a recursive helper function which scanned ensures for any variables defined as [x]_old and asserted that [x] == [x]_old which was logically 'Anded' to the requires condition. Return was fairly simple and just replaced the node with an assignment to the 'ret' value and kept the same expression.

The WP for call requires that I grab the stored procedure from the global mapping and use its requires and ensures. After converting to z3 predicates, the formals were mapped to the actuals and substituted in both requires and ensures. Additionally the 'ret' variable in the ensures condition was substituted with the lhs of the call operation. The havoc function created a mapping from a variable to an obscured version of that variable. The obscured version was implemented with FreshInt and for each variable in the modified list, the modified variable was substituted with the 'havoced' variable in the post condition which essentially "erased" that variable's value in the post condition. There is no explicit frame condition step after the havoc step as a successful havoc step implies that the frame condition will hold for all non-modified variables. Finally the

WP for the call is essentially: if requires holds, and ensures holds and implies the post condition.

Some other particular design choices I made were having old([arg]) evaluate to just a var [arg]_old within the parser itself. This was mostly to make calculating wp for calls easier. I decided to implement booleans as it was useful for my test cases as well. I also decided that if a return was present it had to contain a value (specified with an assert in the parser). This was mostly because I didn't want to account for any scenarios where there is no return value when there should be. If the procedure doesn't return there should not be a return value that affects the WP. Calls also have an lhs of "_" by default to support function calls without an lhs. This allows for functions to be called (potentially modifying state) without explicitly having to assign to a variable.

NOTE: ChatGPT was used to write some code and clarify points of confusion and or misunderstanding. Specifically, helper functions like find_old_vars were mostly implemented by ChatGPT. Some of the Python AST visitor code was partially generated by ChatGPT as I had it generate an example that didn't do exactly what I want, and modified it by hand to suit my needs. It was also used to generate a few of the test cases.