Program Verification: Project A

Marco Dessalvi(s232442),
Leila Dardouri(s231907),
Jonas Max Lindner(s233333)

October 30, 2023

1 Usage

To use our tool, place yourself in the"../project-a-skeleton/fsharp" directory and:

- To test a specific file run "dotnet run *[filename]*"
- To run the tests written by us run "dotnet run test"

2 Structure

Our project can be divided in the following sections:

1. ASTs: in this folder can be found the ASTs of the various intermediate languages (i.e., Microviper, IVL1, DSA, IVL0 and Predicate Logic), as well as the module containing the type we defined for error handling and report.
2. Translators: in this folder can be found the functions that enables the translation from one verification language to another (i.e., from Microviper to IVL1, from IVL1 to DSA ecc. up until Z3).
3. Utils: in this folder can be found several functions used across the other modules, such as prettyprinting functions, functions for converting AST objects to strings, functions used to handle the variables in the translators etc.
4. Checkers.fs : in this module can be found the checker for a Microviper file. It performs the translation from Microviper up until the set efficient weakest precondition, and gives the obtained formulas to Z3 to check the validity of the program.
5. Program.fs: the entrypoint of the tool.

3 Design decisions

The way we verify the program follows exactly the same steps covered during the course, except for how we resolve the "havoc" command, which is done at the DSA level. That is because variables declarations are moved to the top of the body in DSA, so if we resolve havoc before DSA, we lose the reference to the ”forget”, and the havoc just becomes a new variable declaration on top of the body. Consequently, havoc commands are still present at the IVL1 level. Post-resolution, the IVL0 level contains correctly variables declarations only at the beginning of the body but we keep track of the havoc by updating the variable counter.

We enable total correctness for methods and loops through a“decreases” specification, which differs from Viper’s syntax. To specify a variant, it is possible to add “decreases *[variant]*” right after the method declaration (i.e., before the method’s body), or right after a while statement (i.e., before the loop’s body).

If a method variant is not greater than or equal to 0, we chose to report the line of the variant. If a method variant does not decrease in a method call, we report the line of the method call.

To avoid ambiguity, after the IVL1 encoding we add a suffix to the names of all variables, reflecting the method to which the variables belong. Additionally, we add a prefix “old” to the variables used to store the original values of the parameters of the current method. To implement Extension 6, we add a prefix “orig” to the variables needed to store the original values of the parameters passed to methods calls present within the body. All the other variables have a prefix “my”, added to avoid any name-clashes.

4 Implemented features

In our project, we have implemented the following features:

1. Core 1: partial verification of single methods
2. Core 2: error reporting
3. Core 3: supports for loops
4. Extension 1: supports partial correctness for multiple methods
5. Extension 3: supports efficient weakest precondition
6. Extension 4: supports total correctness for loops
7. Extension 5: supports total correctness for recursive methods
8. Extension 6: lifted the restriction for read-only input parameters

All of them have been tested. The test files are located in the “examples” folder, which is further organized into subfolders, each dedicated to a specific feature. Note that some features are tested “implicitly” (for example, Core 2 is tested across all tests). The expected results for each test can be found in the “test” function, inside the “Checkers.fs” file.

No code belongs or has been shared with other groups.