Project 5: generate proofs of homomorphisms

October 13, 2017

1 Project description

Your goal is to program a tool that can automatically generate a proof of functional equivalence for two programs. The input will be a set of three functions:

- the sequential function $f : a- > \sigma$ takes a list as input, and returns a set of variables.
- the join function $join : \sigma \times \sigma \rightarrow \sigma$ takes as input two sets of variables (the output of the sequential function) and returns one output.

The goal is to verify that for every lists a, b, $f(a+b) = join(fa, fb)$ where $+$ is the list concatenation. If this condition is verified, we can then say that f is an homomorphism for the given join. You can have a look at Figure 7 in [1](PDF here) for an example in Dafny, and at the introduction for references on homomorphisms.
You will be provided with a test suite to track your progress and see what specific cases can arise.
We expect at the end of the project to have a tool that can handle functions with at most one loop in their bodies, and all the test cases given during the project. For a more precise description of the programs that the tool should be able to compile into a proof, refer to Section 3. The specificity compared to a normal program is that we have only two functions, and local variables are declared at the function declaration level (no declaration in its body). From an implementation point of view, you can easily parse the program yourself and handle the parsing of the function bodies to another normal C parser (the syntax is a subset of C syntax).
Your program should then be able to read the output of Dafny, and if some parts are not verified, you should be able to output for which part of the join it fails.

Deliverables At the end of the project, you will be asked to provide the **source code** of your tool. It should be **well commented** and easy to understand. We also should be able to **test your tool** on our inputs, so it should be compilable on the Teaching Lab's machines.

Additionally, we would like to have **a small report** (2-3 pages), explaining:

- what are the difficulties you encountered: compiling the inputs in Dafny's language, handling special cases, ...
- how to use your tool.

Notation Your proof generator will be tested on various cases, where the function proposed is provably an homomorphism. You will get a mark proportional to the number of tests passed, and if you pass all tests you will get 60 points (you will be provided with the test suite during the project). Additionally, we will award 20 points if all the proofs can be compiled correctly (but do not verify necessarily). The handling of Dafny's output will be graded on 10 points. A good report will get you 10 points.

2 Language

You can program in any language of your choice, provided you are comfortable with it and you can find sufficient libraries to help you parse the input code. The output should be a program proof for any automatic program prover of your choice, but we strongly recommend using Dafny. The only requirement is that we should be able to run the program prover on the output directly.

3 Program syntax

3.1 Basic types

We only use three basic types in the input programs: `int`, `float` and `bool`.

⟨*type*⟩ ::= 'int' | 'float' | 'bool'

3.2 Expressions

⟨*expression*⟩ ::= ⟨*identifier*⟩
| ⟨*constant*⟩
| '(' ⟨*expression*⟩ ')'
| ⟨*expression*⟩ ⟨*binary-operator*⟩ ⟨*expression*⟩
| ⟨*unary-operator*⟩ ⟨*expression*⟩
| ⟨*identifier*⟩ '(' ⟨*List expression*⟩ ')' // Function call
| ⟨*expression*⟩ '[' ⟨*expression*⟩ ']' // Array access
| ⟨*expression*⟩ ? ⟨*expression*⟩ : ⟨*expression*⟩ // Conditional expression

⟨*binary-operator*⟩ ::= '+' | '-' | 'min' | 'max' | '*' | '/́/ Arithmetic
| '&&' | '||' // Boolean
| '==' '>' | '<' | '>=' | '<=' // Comparisons

⟨*unary-operator*⟩ ::= '-' // Arithmetic
| '!' // Boolean not

3.3 Statements and blocks

⟨*statement*⟩ ::= ⟨*compound-statement*⟩
| ⟨*assignment*⟩
| ⟨*selection-statement*⟩
| ⟨*iteration-statement*⟩

⟨*compound-statement*⟩ ::= '{' ⟨*block-item-list*⟩ '}'

⟨*block-item-list*⟩ ::= ⟨*block-item*⟩
| ⟨*block-item*⟩ ⟨*block-item-list*⟩

⟨*block-item*⟩ ::= ⟨*statement*⟩

⟨*assignment*⟩ ::= ⟨*expression*⟩ '=' ⟨*expression*⟩

⟨*expression-assign*⟩ ⟨*selection-statement*⟩ ::= 'if' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
| 'if' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ 'else' ⟨*statement*⟩

⟨*iteration-statement*⟩ ::= 'for' '(' ⟨*assignment*⟩ ';' ⟨*expression*⟩ ';' ⟨*assignment*⟩ ')' ⟨*statement*⟩

3.4 Input program

In this project, the input program contains only two functions. Note that *join* and *sequential* are keywords in this simple language.
Join declaration:

⟨*join-declaration*⟩ ::= 'join' '(' ⟨*List arg-var*⟩ '|' ⟨*List arg-var*⟩ ')' 'returns' ⟨*List ident*⟩ 'locals' ⟨*List arg-var*⟩ ⟨*compound-statement*⟩

⟨*arg-var*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩

Function declaration:

⟨*function-declaration*⟩ ::= 'sequential' '('⟨*List array-arg*⟩ ',' ⟨*List arg-var*⟩ ')' 'returns' ⟨*List ident*⟩ 'locals' ⟨*List arg-var*⟩ ⟨*compound-statement*⟩

⟨*arg-var*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩ // Scalar arguments

⟨*array-arg*⟩ ::= ⟨*type*⟩ '*' ⟨*ident*⟩ // Array arguments

References

[1] Azadeh Farzan and Victor Nicolet. Synthesis of divide and conquer parallelism for loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 540–555, New York, NY, USA, 2017. ACM.