IMP Language Deductive Verifier

Naoto Kuwayama, Joel Lucas, Raymond Ly, Andy Wang

Simon Fraser University
CMPT 477: Introduction to Formal Verification
Prof. Yuepeng Wang
Fall 2024

Github: https://github.com/naotokuwa/CMPT477TermProject

Language

1 Grammer of IMP

$$
\begin{array}{rcl}
\text{Expression } E & ::= & Z \mid V \mid E + E \mid E \times E \\
\text{Condition } C & ::= & \text{true} \mid \text{false} \mid E = E \mid E \leq E \\
\text{Statement } S & ::= & V := E \mid S; S \mid \text{if } C \text{ then } S \text{ else } S \\
Z & \in & \text{Integers} \\
V & \in & \text{Variables}
\end{array}
$$

Figure 1: Grammar of IMP

Figure 1 shows the context free grammar of the simple imperative programming language (henceforth IMP) used to construct our target program. We exclude while loops for this project.

2 Grammer of Logic

$$
\begin{array}{rcl}
\text{Expression } E & ::= & Z \mid V \mid E + E \mid E \times E \\
\text{Condition } C & ::= & \text{true} \mid \text{false} \mid E = E \mid E \leq E \mid C \wedge C \mid C \vee C \mid C \rightarrow C \mid \neg C \\
Z & \in & \text{Integers} \\
V & \in & \text{Variables}
\end{array}
$$

Figure 2: Grammar of Logic

Figure 2 shows the context-free grammar of the logic we used to construct pre/post conditions for Hoare triples.

Design

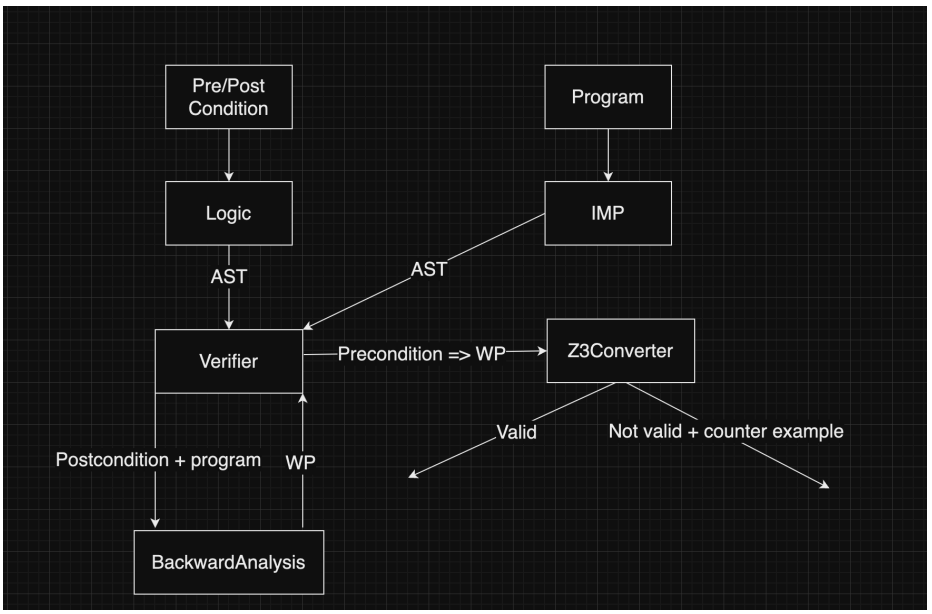

Figure 3: Verifier Workflow

IMP Abstract Syntax Tree - Represents the IMP we are verifying as a structured abstract syntax tree (AST). The IMP language grammar defined in **Figure 1** are represented as nodes in the AST. For instance, the nodes *IntegerExpression* and *VariableExpression* represent integers and variables in a program, respectively. The AST is implemented using the **Visitor Pattern** to allow us to easily add behaviours to AST nodes. All of the

problems we have solved use a post order tree traversal, greatly leveraging the combination of an AST representation and visitor pattern.

Logic - Represents pre/post conditions of Hoare logic. Originally, we planned to represent logic with a different AST, but we ultimately combined IMP and Logic into a single AST. This is because *Expression* and *Condition* in IMP are subsets of *Expression* and *Condition* in Logic according to our context-free grammar defined in ***Figure 1*** and ***2***. Therefore, a single AST represents both IMP and Logic despite their semantic differences.

Backward Analysis - Takes two structures: a program in IMP and a postcondition of a Hoare triple. By following the weakest precondition rules (recursively defined in slide 7 of Lecture 13), this module then computes the weakest precondition of the given program in terms of the given postcondition.

Z3 Converter - Traverses the logic AST in postorder fashion, converting each node to corresponding Z3 objects. After the conversion, it negates the converted formula and checks if it is unsatisfiable, to check validity.

Verifier - Combines all modules (including our AST) into a single application. Given a Hoare triple (statement, pre/post conditions) the Verifier computes the weakest precondition of the statement in terms of the postcondition. Then, it checks the validity of the Hoare triple by checking whether the given precondition implies the computed weakest precondition. If the Hoare triple is invalid, the program provides a counterexample containing variable assignments that do not satisfy the given conditions.

Implementation

IMP/Logic AST Implementation

To represent our IMP and logic grammar as an AST, we used inheritance and three super classes: Condition, Expression and Statement. To avoid undefined behaviour, all objects should only occur once in the tree (i.e. if you have a statement like x + x, you should create two separate x VariableExpression objects). The following table shows which subclasses represent which grammar structure(s).

Condition (C) Subclasses		
BinaryCondition	E = E, E ≤ E	Representation depends on type attribute
BinaryConnective	C ∧ C, C ∨ C, C → C	Representation depends on type attribute. Exclusive to logic grammar
UnaryConnective	¬C	Exclusive to logic grammar
Boolean	true, false	
Expression (E) Subclasses		
BinaryExpression	E x E, E + E	Representation depends on type attribute
IntegerExpression	Z	
VariableExpression	V	
Statement (S) Subclasses		
Assignment	V := E	
Composition	S; S	
If	if C then S else S	

Condition/Expression/StatementSerializeVisitor(s) convert node classes to strings for testing purposes.

Backward Analysis Implementation

To compute the weakest precondition(wp), given a post condition and program (represented as an AST), we implemented: LogicVisitor, Cond/ExprCopyVisitor(s) and Condition/ExpressionReplacementVisitor(s).

LogicVisitor is the start of performing backward analysis and is initialised with a post condition (Q). Given a program, the rules in **Figure 4** are used to recursively visit each node in the AST and compute the wp. As the same visitor visits each node (which may update Q if it is an Assignment), Q is saved/restored before visiting an If statement branch. If no post condition was given, LogicVisitor immediately returns $\{true\}$, as performing backward analysis with the post condition $\{true\}$ will always yield $\{true\}$ as the wp.

$$wp(x := E, Q) = Q[E/x]$$

$$wp(s_1; s_2, Q) = wp(s_1, wp(s_2, Q))$$

$$wp(if\ C\ then\ s_1\ else\ s_2, Q) = C \rightarrow wp(s_1, Q) \wedge \neg C \rightarrow wp(s_2, Q)$$

Figure 4: Backward Analysis Rules

ConditionReplacementVisitor and ExpressionReplacementVisitor perform the Assignment rule. Given a postcondition, both classes traverse the tree and replace variables matching the target symbol with the replacement expression.

CondCopyVisitor and ExprCopyVisitor deep copy the replacement expression, and replace the matching variable with the copy, to avoid wrongfully updating nodes in the post condition after an assignment.

Z3 Implementation

To validate that a given postcondition implies the weakest precondition generated from backward analysis, our program uses Z3. Our program also leverages Z3's ability to create models to generate counterexamples for invalid programs. To do this, we must convert our IMP/Logic business objects to types that Z3 can understand.

The ConditionZ3Visitor and ExpressionZ3Visitor classes convert our custom AST objects to Z3 objects. The Visitor Pattern is used to extend the behaviour of nodes in the AST. Given an AST representing any logical formula according to **Figure 2**, the visitors builds an equivalent Z3 formula through a postorder traversal of the tree. While traversing the tree, the visitors check the type of each node and create corresponding Z3 objects by calling Context.mk_(). A benefit of using this approach is our grammar can be easily extended to include any operator that Z3 supports (for example, the >= operator) without needing to change the structure of our AST.

After generation, ConditionZ3Visitor validates Z3 formulas by checking whether the negation of the formula is valid. If the formula is not valid, then a counterexample can be generated through getCounterexample_(). Three different methods are available for getting the counterexample as either a Z3-style string, a simplified string, or a map of variables to values. All three depend on Z3's Solver.getModel() to obtain a model.

Verifier Implementation

The Verifier class is responsible for combining our IMP/Logic AST, Backward Analysis, and Z3 modules into a single interface that can accept and validate an IMP program, an (optional) precondition, and a postcondition. The verify() method uses LogicVisitor to compute the weakest precondition of the program through backward

analysis. ConditionZ3Visitor is used to create a Z3 formula from the given IMP program, and validates it according to the given pre/post conditions. If the program is invalid, a counterexample can be obtained.

Evaluation

Our Verifier is evaluated by 10 Test Programs, designed so that a human can understand and reason about the program, and to test the implemented capabilities of our IMP Language and pre and post conditions. For each program, we created various test cases using valid and invalid pre and post conditions with the program, and the program specifications with an incorrect program as methods in Dafny. We then write the correct and incorrect program in our IMP language, and write the valid and invalid pre and post conditions. Each Dafny method has a corresponding Java Test Method. We then run our verifier on our programs and their specifications, and expect to get the same answer as Dafny. Verifier Tests and Dafny Tests are found in the src/test/java/verifier/ folder.

Program	**Description**	**Run Time**
True/False	Returns the input plus one.	123ms
Trajectory	Returns the trajectory of two inputs a, b, equal to b + (b - a).	93ms
NOR	Returns the nor operation of two inputs p, q, aka returns 1 (true) when neither p nor q is 1 (true).	148ms
Largest of Three	Returns the largest of three inputs.	85ms
Sign of Integer	Returns 1 if the input is greater than 0, returns 0 if the input is 0, returns -1 if the input is less than 0.	91ms
Equality	Returns 1 (true) if the two inputs are equal, else 0.	100ms
Age	Returns the age, given a birth year, with the assumption that the current year is 2024.	153ms
Toggle Calculator	Returns the sum of input parameters a, b if the input parameter toggleAddMul is 1, and returns the product of input parameters a, b if the input parameter toggleAdd is 0.	100ms
Min	Returns the smaller of the two inputs.	95ms
Abs	Returns the absolute value of the input.	155ms

Conclusion & Limitations

We successfully implemented a deductive verifier for a Simplified IMP Language, with a modular design to support extendability. Using the visitor pattern simplified our work, as it provided a structured and efficient approach to handling tree traversal tasks, which were primarily post-order. We also developed granular unit tests for each component, ensuring their correctness and boosting our confidence in the system's overall reliability.

There are two improvements that can be made to our program. The first involves extending support for more complex syntax in the IMP language. Currently, our IMP does not allow constructs like while loops, function calls, or pointers. While adding these features would significantly increase the complexity of the expressible program, the overall design and implementation remain largely unaffected, thanks to the flexibility of the visitor pattern. The second direction is expanding the verifier to target real-world programming languages such as Python or C. Although this would require modifications to the implementation of the Abstract Syntax Tree

(AST) and logic handling, our current design is robust enough to accommodate these changes. By writing new visitors tailored to these languages (or incorporating existing visitors such as LLVM Visitor[1] or Python Visitor[2]), we can ensure a smooth transition to verifying more complex and practical programming scenarios.

Instructions

Basic Example

Given the following program with no precondition and the postcondition 0 <= y:

```
{true}
if x <= 0 y := x * -1 else y := x
{0 <= y}

Verifier = new Verifier();    // Create a Verifier

// Assuming we have the root of an AST representing the program to verify.
Statement program = someManuallyCreatedAST();

// Pre/post conditions must be created manually as well
// Postcondition: 0 <= y
Condition postcondition = new BinaryCondition(ConditionType.LE,
                    new IntegerExpression(0),
                    new VariableExpression("y"));

// Verify the program according to the postcondition
verifier.verify(program, postcondition);    // returns true

// Preconditions are optional
// Precondition: NOT( x == x )
Condition precondition = new UnaryConnective(ConnectiveType.NOT,
   new BinaryCondition(ConditionType.LE, new IntegerExpression(0), new VariableExpression("y")));

verifier.verify(program, precondition, postcondition);    // returns false
// After verifying a program, a counterexample can be generated for invalid programs

String simpleString = verifier.getCounterexampleString() // Get a counterexample as a string
String z3String = verifier.getCounterexampleRaw();  // Get a counterexample as a z3-style string
Map<String, Integer> = verifier.getCounterexampleMap();  // Get a counterexample as a map
```

Creating a Program

The nodes of the AST for the program should follow **Figure 1: Grammar of IMP**. A concrete example can be found in VerifierAbsTest::createProgram().

Expressions:

1. Variables | var x = new IntegerExpression("x")

---

[1] LLVM Project. llvm::InstVisitor Class Template Reference. LLVM, n.d. Web. 27 Nov. 2024. https://llvm.org/doxygen/classllvm_1_1InstVisitor.html.
[2] Python Software Foundation. ast.NodeVisitor — Abstract Syntax Trees. Python 3 Documentation, n.d. Web. 27 Nov. 2024. https://docs.python.org/3/library/ast.html#ast.NodeVisitor.

2. Integers | var zero = new VariableExpression(0)

Conditions:
1. True | var t = new Boolean(true)
2. False | var f = new Boolean(false)
3. Add | var add = new BinaryExpression(ExpressionType.ADD, x, y)
4. Multiply | var mul = new BinaryExpression(ExpressionType.MUL, x, y)

Statements:
1. Assignment | var assign = new Assignment(x, zero)
2. Composition | var composition = new Composition(statement, statement)
3. IfElse | var ifElse = new If(condition, statement, statement)

Creating pre- / post- conditions

To create pre- / post- conditions, create an AST following **Figure 2: Grammar of Logic**. Same nodes as **Creating a Program**, but with additional logical condition operators (only for use in pre- / post-conditions):
1. ... All conditions from IMP +
2. And | var and = new BinaryConnective(ConnectiveType.AND, cond, cond)
3. Or | var or = new BinaryConnective(ConnectiveType.OR, cond, cond)
4. Implies | var implies = new BinaryConnective(ConnectiveType.IMPLIES, cond, cond)
5. Not | var not = new UnaryConnective(ConnectiveType.NOT, cond)

Verifying a Program

verify(program, precondition, postcondition): Returns true if the program with pre- and post- conditions is valid.
verify(program, postcondition): Same as above but with no precondition (precondition = true).

Getting a counterexample

See the end of **Basic Example** for how to obtain counterexamples (inputs which would make the program incorrect according to the given conditions). The most important thing is to call verify() before calling getCounterexample_(), otherwise an exception will be thrown. getCounterexample_() will always return a counterexample for the last verified program. If the program is valid, an empty string or map will be returned.

Division of Work

Naoto Kuwayama: Implemented an AST of IMP/Logic, and the replacement rule. Wrote tests for 2 programs. Wrote the Design, Language and Conclusion/Limitation sections of the report.

Joel Lucas: Implemented the backwards analysis and copy visitors. Wrote tests for 2 programs. Wrote half of the Implementation section of the report.

Raymond Ly: Wrote z3 visitor and verifier modules. Wrote tests for Abs/Min. Wrote half of the Implementation section of the report. Wrote the Code Readme and Instructions.

Andy Wang: Designed IMP programs for testing. Wrote Dafny Methods of test program and pre/post conditions for testing. Developed tests for 4 programs to test our verifier works properly. Wrote the Evaluation section of the report.