Abstract Interpreter Project Report

Parsa VARES
University of Luxembourg
Email: parsa.vares.001@student.uni.lu

January 10, 2025

1 Introduction

This report presents the implementation and explanation of an abstract interpreter for static analysis of C-like programs. The abstract interpreter is built to handle variable assignments, arithmetic operations, comparison operations, and control flow constructs such as `if-else` and loops. The project involves incrementally developing the interpreter to verify assertions and detect general bugs like division-by-zero and integer overflow.

2 Project Goals

The primary objectives of the project are as follows:

1. Verify assertions provided by the user in the form of `assert(condition)`.
2. Detect general bugs such as division-by-zero and integer overflow.
3. Incrementally enhance the interpreter to support constructs required by specific test cases.
4. Implement equational form and abstract fixpoint computation to handle loops and ensure termination.

3 Building and Running the Project

To build and run the abstract interpreter for the provided tests, follow these steps:

3.1 Building the Project

Open a terminal in the root directory of the project and execute the following commands:

```
cmake -S . -B build
cmake --build build
```

3.2 Running the Tests

To run the abstract interpreter on the test cases, use the following command, replacing `tests/easy1.c` with the path to the desired test file:

```
.\build\absint tests/easy1.c
```

For other test files, repeat the command, modifying the test file path as needed

4 Code Structure

The project consists of nine primary source files, detailed below:

4.1 `main.cpp`

This is the entry point of the program. It initializes the parser, constructs the AST, and invokes the abstract interpreter to analyze the program.

```cpp
#include <fstream>
#include <sstream>
#include "parser.hpp"
#include "ast.hpp"
#include "abstract_interpreter.hpp"

int main(int argc, char** argv) {
    if (argc != 2) {
        std::cout << "Usage: " << argv[0] << " tests/00.c" << std::endl
            ;
        return 1;
    }
    ... // File reading and AST construction
    FancyAnalyzer analyzer;
    analyzer.constructSolverSystem(rootAst);
    analyzer.solveSystem();
    analyzer.printSystemState();
    analyzer.printSystemNotes();
    return 0;
}
```

Listing 1: main.cpp

4.2 `abstract_interpreter.cpp`

This file implements the core of the abstract interpreter, including methods for handling AST traversal, fixpoint computation, and bug detection.

```cpp
void FancyAnalyzer::constructSolverSystem(const ASTNode &node) {
    if (node.type == NodeType::IFELSE) {
        ... // Handling if-else construct
    } else if (node.type == NodeType::WHILELOOP) {
        ... // Handling loops with fixpoint computation
    }
    // Recursively process other nodes
    for (const auto &sub : node.children) {
        constructSolverSystem(sub);
    }
}

void FancyAnalyzer::solveSystem() {
    bool isFixedPoint = false;
    do {
        isFixedPoint = solverStep();
        ... // Apply widening if necessary
    } while (!isFixedPoint);
}
```

Listing 2: abstract_interpreter.cpp

4.3 `ActionSemantics.hpp`

Defines semantic actions such as variable declarations, assignments, and merges.

```cpp
namespace semantics {
    class VarAllocator : public StateOperator { ... };
    class SetOperator : public StateOperator { ... };
    class MergeStates : public SolverInstruction { ... };
    class CheckOperator : public StateOperator { ... };
}
```

Listing 3: ActionSemantics.hpp

4.4 `DisjointRangeSet.hpp`

Implements interval arithmetic and the `IntervalsUnion` class for disjoint range representation.

```cpp
class IntervalsUnion {
    void addRange(const Range &rg);
    IntervalsUnion join(const IntervalsUnion &oth) const;
    IntervalsUnion meet(const IntervalsUnion &oth) const;
    ...
};
```

Listing 4: DisjointRangeSet.hpp

4.5 `InvariantStore.hpp`

Manages variable states and supports operations like narrowing, widening, and union.

```cpp
class VariablesState {
    void narrow(const std::string &var, const Range &rng);
    VariablesState unionWith(const VariablesState &oth) const;
    VariablesState widenWith(const VariablesState &oldState) const;
    ...
};
```

Listing 5: InvariantStore.hpp

4.6 `ValueInterval.hpp`

Defines the `Range` class for individual intervals with arithmetic operations.

```
class Range {
    Range unify(const Range &oth) const;
    Range intersect(const Range &oth) const;
    ...
};
```

Listing 6: ValueInterval.hpp

4.7 `parser.hpp`

Parses the input program and constructs the AST.

```
class FancyParser {
    ASTNode parse(const std::string &input);
    ...
};
```

Listing 7: parser.hpp

4.8 `ast.hpp`

This file defines the structure and components of the Abstract Syntax Tree (AST). The AST represents the program's structure, with nodes for different types of operations and constructs such as variables, assignments, arithmetic operations, and control flow.

```

struct ASTNode { size_t id; NodeType type; std::variant<std::string,
    int, BinOp, LogicOp> value; std::vector<ASTNode> children;

arduino
Copy code
void print(int depth = 0) const {
    ... // Recursive printing for debugging
}
};
```

Listing 8: ast.hpp

The AST serves as the backbone of the interpreter, allowing traversal and analysis of program structure during static analysis.

4.9 `abstract_interpreter.hpp`

This file declares the `FancyAnalyzer` class, which is the core component of the abstract interpreter. It provides methods to traverse the AST, construct the solver system, and perform fixpoint computations.

```

csharp
Copy code
bool solverStep();
```

```cpp
void applyWidening(const StatesCollection &oldFrames, StatesCollection
    &newFrames);
...
public: void constructSolverSystem(const ASTNode &node); void
    solveSystem(); void printSystemState() const; void printSystemNotes
    () const; };
```

Listing 9: abstract$_i$*nterpreter.hpp*

This header file connects the implementation in `abstract_interpreter.cpp` to the rest of the project, enabling static analysis of the AST.

5 Handling Control Flow Constructs

5.1 If-Else Handling

The `if-else` construct is evaluated by splitting the program state into branches and computing their intervals independently. After evaluating both branches, the states are joined.

1. Evaluate the condition and split the intervals into `if` and `else` branches.
2. Restrict the variable intervals for each branch.
3. Merge the results of both branches using the join operation.

```cpp
if (node.type == NodeType::IFELSE) {
    auto condition = node.children[0];
    auto ifBody = node.children[1];
    auto elseBody = node.children[2];
    ... // Restrict intervals and compute branches
    solverActions.push_back(std::make_shared<MergeStates>(...));
}
```

Listing 10: If-Else Handling

5.2 Loop Handling

Loops are handled by iteratively evaluating the body until a fixpoint is reached. Widening is applied to ensure termination for unbounded loops.

1. Evaluate the loop condition and restrict intervals.
2. Iteratively execute the loop body and compute state transitions.
3. Apply widening after a predefined number of iterations.

```cpp
do {
    isFixedPoint = solverStep();
    iterationCount++;
    if (iterationCount == WIDEN_THRESHOLD) {
        applyWidening(oldFrames, frames);
    }
} while (!isFixedPoint);
```

Listing 11: Loop Handling

6 Conclusion

This project demonstrates the incremental construction of an abstract interpreter for static analysis. The implementation supports assertions, arithmetic operations, comparison, `if-else` constructs, and loops with fixpoint computation and widening.