REVERSE INTERPRETATION FOR PROGRAM ANALYSIS

RISHABH ARIJEET
210842

UGP Report
UGP-II CS396A

Indian Institute of Technology Kanpur

Supervisor: Dr. Subhajit Roy
Project Mentor: Pankaj Kalita

Acknowledgement

I would like to express my sincere gratitude to Professor Subhajit Roy for their invaluable guidance, support, and encouragement throughout the course of this project. I am deeply grateful for the time and effort they invested in mentoring me. I would also like to extend my thanks to the Ph.D. scholar, Pankaj Kumar Kalita, whose mentorship, technical expertise, and patient guidance were crucial in overcoming the challenges encountered during this project. His valuable assistance throughout the project helped me in a great way.

Abstract

The traditional approach to abstract interpretation using the worklist algorithm is forward analysis, where the abstract states are propagated forward along the control flow graph (CFG) of the program. It processes the nodes in the CFG in a forward direction using the worklist algorithm, following the control flow of the program. In forward analysis, we move from pre-conditions to post-conditions over-approximating the abstract domains, so as to prove the correctness of the program. We are proposing a new approach for proving the incorrectness of programs using a backward analysis kind of approach where we move from post-conditions to pre-conditions under-approximating the abstract domains. In this reverse interpretation approach, we move in the reverse direction of the CFG and try to prove the incorrectness of a buggy program by trying to find buggy points in the pre-conditions.

Contents

Introduction and Background

1.1 Abstract Interpretation

Abstract interpretation is a framework for performing program analysis using abstract values instead of concrete values. We define abstract semantics over the abstract domain, which specifies how the language constructs (e.g., assignments, control flow statements, function calls) are interpreted over the chosen abstract domain.
There are 2 components of an abstract interpretation:

- **Lattice**: This consists of the set of abstract values, plus the relationships among them indicating which properties are stronger than others.
  A lattice consists of two parts: a domain,i.e., a set of points and an ordering or hierarchy among the elements.

- **Transfer functions**: These tell how to perform computations on abstract values. For instance, in order to perform abstract interpretation on a program that contains the expression x + y, the abstract interpretation must have a definition of the + operator over the abstract values that are estimates for the run-time values held in variables x and y.

Abstract interpretation operates by analyzing the program's control flow graph (CFG), which represents the possible execution paths through the program. The CFG is composed of basic blocks, which are sequences of instructions without any control transfers (branches or jumps) except possibly at the end. To build a CFG we first build basic blocks, and then we add edges that represent control flow between these basic blocks. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

1.2 Worklist Algorithm

It is used in abstract interpretation for performing data flow analysis on a control flow graph (CFG). The algorithm is as follows:

1. The worklist is initialized with the list of all basic blocks in the CFG.
2. while worklist is not empty do:
    (a) A basic block (currBB) is dequeued from the worklist.

    (b) Compute the inlist of currBB by considering the "out" states of its predecessor blocks (predList).

    (c) Meet operation is applied to the inlist to compute the "in" state of currBB (currInVal).

    (d) The transferFunction is called with the "in" state of currBB and the block itself (currBB) to compute the "out" state of currBB.

    (e) If the "out" state of currBB has changed compared to the previous iteration, the successors of currBB are enqueued onto the worklist.

This implementation follows the standard worklist algorithm for data flow analysis, where the "in" and "out" states are iteratively computed and propagated through the CFG until a fixpoint is reached (when the worklist becomes empty).

1.3 Over-Approximation v/s Under-Approximation

Forward interpretation overapproximates the abstract domain by ensuring that the computed abstract states always include all possible concrete states that could arise during the program's execution. This overapproximation guarantees soundness, meaning that if a property holds in the abstract domain, it is guaranteed to hold in the concrete domain as well. Therefore, it can be used to prove the correctness of a program. Now, we propose reverse abstract interpretation by underapproximating the abstract values, meaning that the computed abstract states represent a subset of the possible concrete states that can satisfy the postcondition. It can be used to prove the incorrectness of a program by taking the negation of the desired postcondition and performing a backward analysis. The computed abstract states after taking an intersection with the actual pre-conditions will give the buggy points in the program which will fail to satisfy the post-conditions.

Reversing the Program

We can reverse the given sample program(basically we have to reach the pre-condition starting with post-condition) using the following rules:

- We define "assume" as a function which truncates the interval domain as per the condition. For example if a :[0,1000], then after assume($a \leq 5$), a is a:[0,5].

- For simple statements, we have,

$$a \ = a \ op \ b, \ => \ a \ = \ a \ op^{-1} \ c$$

  ,where op^{-1} is the inverse operation of op.

- For IfElse statement, we have

```
                            if (non-det):
if (condition):                 rev(body1)
    (body1)                     assume(condition)
else:               ==>     else:
    (body2)                     rev(body2)
                                assume( ~condition)
```

  ,where rev(body) is the reverse program of a program body.

- Similarly for a While loop, we have

```
                            while (non-det):
while (condition):  ==>         rev(body)
    (body)                      assume(condition)
```

Reverse Interpretation

To get the algorithm for reverse interpretation, we make some changes in the forward implementation like under-approximating the abstract values instead of over-approximating them in order to reach to pre-conditions starting from post-conditions. More precisely, we make changes in the meet operation to now on a fork, take the intersection of the abstract domain values, instead of union(which worked for forward interpretation).

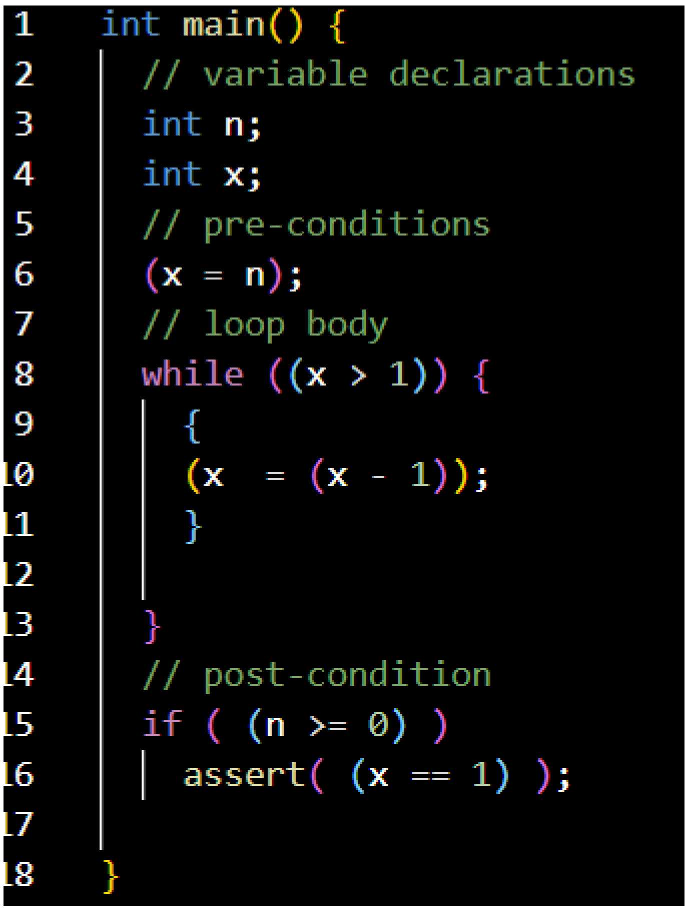
For reverse interpretation, basically what we want to do is the following,

1. **Negate the Postcondition**: Instead of using the given postcondition directly, we take its negation and assume it to be our pre-condition for the reverse program.
2. **Run the Reverse Interpretation Algorithm**: With the negated postcondition as the pre-condition for the reverse program, we apply reverse interpretation on the reverse program to compute the post-conditions.
3. **Analyze the Post-conditions**: We then take an intersection of the original pre-conditions with the new post-conditions that we got from applying the reverse interpretation. These are the set of points which are buggy, i.e. the program behaves incorrectly for these set of points.

Demonstration

Let us assume we have implemented a narrowing kind of approach for abstract values, for example if an interval is decreasing from one side after each iteration, then after a given no of iterations we set it to become empty. Here we assume the threshold for narrowing to be 100. Here are the demonstrations of a few programs on the reverse interpretation algorithm:

4.1 Program1



```
int main() {
  // variable declarations
  int n;
  int x;
  // pre-conditions
  (x = n);
  // loop body
  while ((x > 1)) {
    {
    (x  = (x - 1));
    }

  }
  // post-condition
  if ( (n >= 0) )
    assert( (x == 1) );

}
```

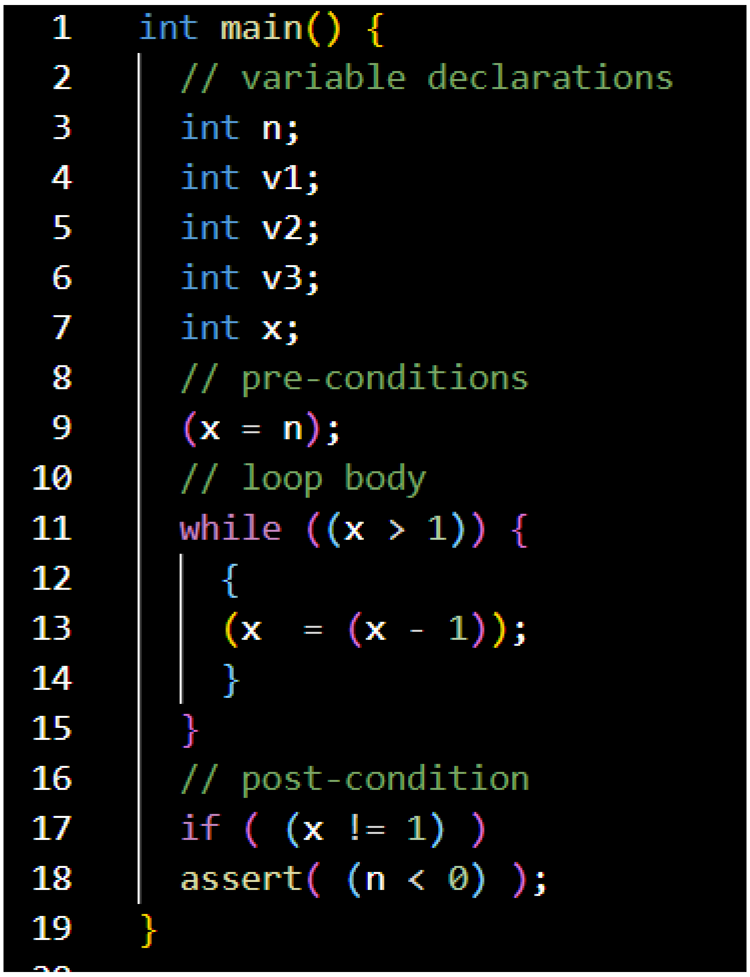
Figure 4.1: Prog1

Now, we have the following results for the program:

- No of iterations = 308
- Pre-condition(reverse program) = x:[[-inf,0],[2,inf]]
- Post-condition(reverse program) = x:[[-inf,0]]
- Bug = x:[0,0]

On taking the intersection of the post-conditions of reverse program with the original pre-condition ,i.e., x≥0, we get a point n=x=0 as the bug. And indeed, for n=x=0, after the loop-body, x is still 0, but according to post-conditions, for n≥0, we must have x=1, which is an error.

4.2 Program2



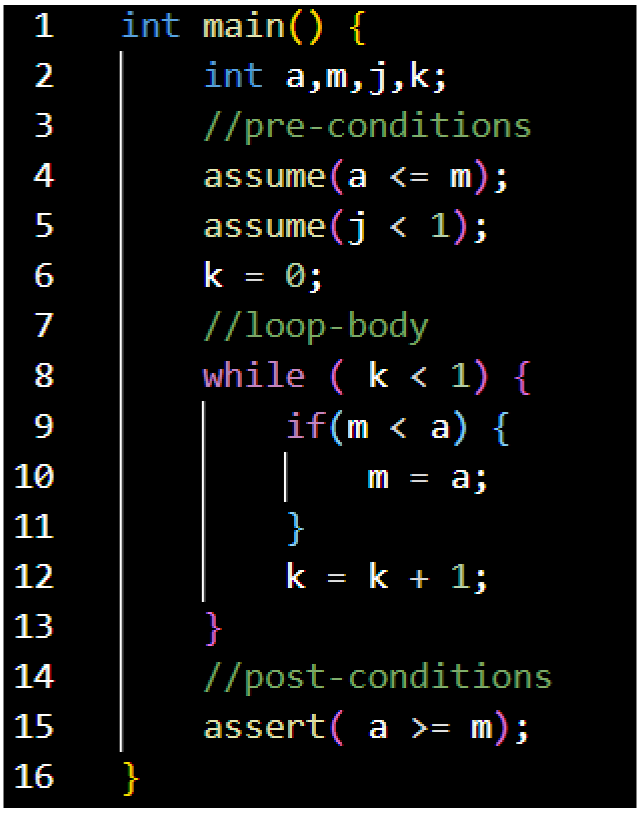
```
int main() {
  // variable declarations
  int n;
  int v1;
  int v2;
  int v3;
  int x;
  // pre-conditions
  (x = n);
  // loop body
  while ((x > 1)) {
    {
    (x  = (x - 1));
    }
  }
  // post-condition
  if ( (x != 1) )
  assert( (n < 0) );
}
```

Figure 4.2: Prog2

- No of iterations = 308
- Pre-condition(reverse program) = x:[[-inf,0],[2,inf]]
- Post-condition(reverse program) = x:[[-inf,0]]
- Bug = x:[0,0]

Here, we assume x≠1 and then run the reverse program. On taking the intersection of the post-conditions of reverse program with the original pre-condition ,i.e., x≠1, we get a point n=x=0 as the bug. And indeed, for n=0 and x=0, after the loop-body, x is still 0, but according to post-conditions, for x≠1, we must have n$lt$0 but we have n=0, which is an error.

4.3 Program3



```c
int main() {
    int a,m,j,k;
    //pre-conditions
    assume(a <= m);
    assume(j < 1);
    k = 0;
    //loop-body
    while ( k < 1) {
        if(m < a) {
            m = a;
        }
        k = k + 1;
    }
    //post-conditions
    assert( a >= m);
}
```

Figure 4.3: Prog3

Now, we have the following results for the program:

- No of iterations = 8
- Pre-condition(reverse program) = a:[[-inf,1]] , m:[[1,inf]]
- Post-condition(reverse program) = a:[[-inf,0]], m:[[1,inf]]
- Bug = a:[[-inf,0]], m:[[1,inf]]

We have taken a≤1 and m≥1 to amount for the assume(a≤m) in the original pre-conditions. On taking the intersection of the post-conditions of reverse program with the original pre-condition ,i.e., a≤m, we get a≤0 and m≥1 as the bug. And indeed, for any a≤0, say a=-5 and any m≥1, say m=9 , after the loop-body, m=9 and a=-5, but it does not satisfy the assert(a≥ m) in the post-conditions, which is an error.

Conclusion

By inverting the traditional forward analysis approach, this method offers a unique perspective on program behavior, enabling the detection of bugs and corner cases that may be overlooked by conventional techniques. Reverse interpretation can be a valuable tool in the program analysis and verification domain, complementing existing techniques and providing a comprehensive approach to ensuring software correctness and reliability.

Bibliography

1. Representation and Analysis of Software
2. Abstract Implementation
3. Simple Abstract Interpreter
4. Program Analysis, Prof Subhajit Roy, ACM Summer School 2019