Assignment 3: Dataflow Analysis

Milan Lagae

Institution/University Name: Vrije Universiteit Brussel
Faculty: Sciences and Bioengineering Sciences
Course: Software Quality Analysis

Contents

1 Intro

This report will discuss an implementation for the assignment “Assignment 3: Dataflow Analysis” for the course: Software Quality Analysis.

2 Discussion Point 1

This section will discuss the implementation of the first discussion point.

2.1 Implementation

This subsection will discuss the implementation for the first discussion point. First, the asserts will be covered present in the IntervalAnalysis.scala file. Continuing, the widen interval & assignments will be covered, present in the ValueAnaylsis.scale file.

2.1.1 Asserts

For both the asserts shown in Listing 1 & Listing 2, retrieve the declaration of the binary operation. Retrieve by using the declaration from s the old interval. Using the widenInterval operation a new interval is created, passing the old interval to it, with the second argument: (i, PInf).

```
// x >= value
case ABinaryOp(GreatThan, id: AIdentifier, ANumber(i, _), _) =>
   val xDecl = id.declaration
  // Get the interval for the declaration
  val old = s(xDecl)
  // Create the new interval by applying (zero is ignored?)
  val newInterval = widenInterval(old, (i, PInf))
  // Update with the new interval
  s.updated(xDecl, newInterval)
```

Listing 1: Assert - Version 1

```
// value >= number
case ABinaryOp(GreatThan, ANumber(i, _), id: AIdentifier, _) =>
  val xDecl = id.declaration
  // Get the interval for the declaration
  val old = s(xDecl)
  // Create the new interval by applying (zero is ignored?)
  val newInterval = widenInterval(old, (i, MInf))
  // Update with the new interval
  s.updated(xDecl, newInterval)
```

Listing 2: Assert - Version 2

2.1.2 Widen Interval

As stated on the slides, the gt operation is the application of the intersect operation on the the list of 4 four values, as shown in Listing 3.

```
case ((l1, h1), (l2, h2)) => {
  IntervalLattice.intersect((l1, h1), (l2, IntervalLattice.PInf))
}
```

Listing 3: widenInterval

2.1.3 Assignment(s)

For the list of assignments, iterate the list of declared ids, and update the state of the declared id with the top value.

```
// var declarations
// ⟨vi⟩= ⟨x=E⟩= JOIN(vi)[x ↦ eval(JOIN(vi), E)]
case varr: AVarStmt => {
  varr.declIds.foldLeft(s) { (state, decl) =>
    state.updated(decl, valuelattice.top)
  }
}
```

Listing 4: Declarations

Create a new interval by applying the eval function on the element. Update the interval by using the id and setting the new interval value.

```
// assignments
// ⟨vi⟩= JOIN(vi)
case AAssignStmt(id: AIdentifier, right, _) => {
  val interval = eval(right, s)
  s.updated(id, interval)
}
```

Listing 5: Declaration

2.2 Results

The results of executing the interval analysis on the loopproject.tip example file with the following command: ./tip -interval wlrw vubexamples/loopproject.tip can be seen in Figure 1.

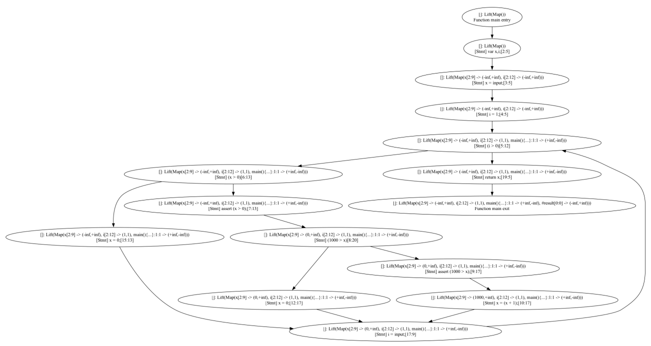

Figure 1: Interval Analysis Result

2.3 Analysis Precision

Question(s): What would be the most precise result? Why does the analysis lose precision on this program?

2.3.1 Sound Result

The most mathematically sound result for x at the return statement is: $x \in [0, 1000]$.

If the input statement at the start of the program declares x to be a negative value, x is immediately set to 0, that being the lower bound.

For the upper bound, there are statement(s) and control flow, constraining the upper bound of the variable x. Once the value of the variable satisfies: $x > 0$, the variable is incremented while the following is satisfied: $(1000 > x)$. Once this predicate is not satisfied anymore, the variable is set to 0. Thus constraining the variable x to the upper bound of 1000.

2.3.2 Precision Loss

As described in [1], the widening operator used during the interval analysis, is an over approximation of the least fixed point. Since the original operator as described in [1], did not stabilize or necessarily have a least fixed point. For the example above, this is the upper bound of the interval is $+\infty$, instead of the sound one: 1000.

TODO: More here?

3 Discussion Point 2

This section will discuss the implementation of the second discussion point.

3.1 Implementation

This subsection will discuss the implementation for the second discussion point, implementing loop unrolling. The files: ValueAnalysis.scala and CallContext.scala have both been modified.

3.1.1 Context

The loop context is created just as the return context is, append the call string context to the existing context and the the k latest context, and discard the rest.

```scala
// MOD-DP2
def makeLoopContext(c: CallStringContext, n: CfgNode, x: statelattice.Element): CallStringContext = {
  // Add node to call string context, while maintaining limit on context length
  CallStringContext((n :: c.cs).slice(0, maxCallStringLength))
}
```

Listing 6: Loop Context

3.1.2 Unrolling

Detecting loop head & start is done by using the loophead method, the n value is passed to it. If it returns true, retrieve the node for which it matched. Retrieve the loophead by taking the head of the result of the operation the done in the loophead method. Create a new context, by passing the values to the function shown in Listing 6. Use the currentContext, loopStart and s as values.

The newly created context is propagated, by using the propagate method, passing the s as the lattice value, in conjunction with the newContext and AstNode for which the if matched.

```scala
//// Discussion Point 2: COMPLETE HERE
// Thus, to determine the starts and ends of loops you must use the cfg.dominators function.
case m: CfgStmtNode if loophead(n) => {
  val node = m.data
  val loopStart = (m.succ intersect dominators(m)).head
  val newContext = makeLoopContext(currentContext, loopStart, s)
  propagate(s, (newContext, m))

  s
}
```

Listing 7: Loop Unrolling

3.2 Results

The results of executing the interval analysis with loop unrolling on the loopproject.tip example file with the following command: `./tip -interval wlrw vubexamples/loopproject.tip` can be seen in Figure 2.

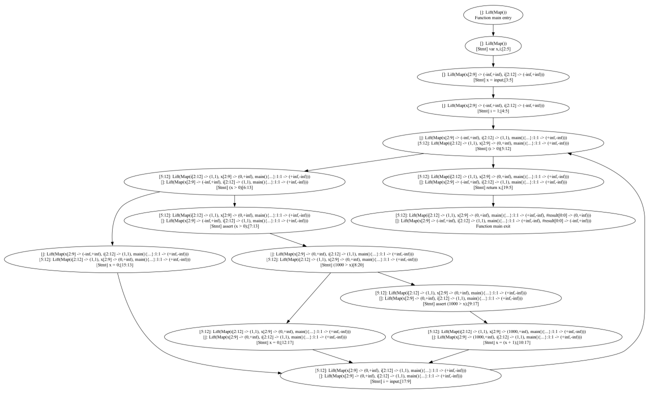
[]: Lift(Map()) Function main entry
[]: Lift(Map()) [Stmt] var x,i;[2:5]
[]: Lift(Map(x[2:9] -> (-inf,+inf), i[2:12] -> (-inf,+inf))) [Stmt] x = input;[3:5]
[]: Lift(Map(x[2:9] -> (-inf,+inf), i[2:12] -> (-inf,+inf))) [Stmt] i = 1;[4:5]
[]: Lift(Map(x[2:9] -> (-inf,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (0,+inf), main(){...}:1:1 -> (+inf,-inf))) [Stmt] (i > 0)[5:12]
[5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (0,+inf), main(){...}:1:1 -> (+inf,-inf))) []: Lift(Map(x[2:9] -> (-inf,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [Stmt] (x > 0)[6:13]
[5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (0,+inf), main(){...}:1:1 -> (+inf,-inf))) []: Lift(Map(x[2:9] -> (-inf,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [Stmt] return x;[19:5]
[5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (0,+inf), main(){...}:1:1 -> (+inf,-inf))) []: Lift(Map(x[2:9] -> (-inf,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [Stmt] assert (x > 0);[7:13]
[5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (0,+inf), main(){...}:1:1 -> (+inf,-inf), #result[0:0] -> (0,+inf))) []: Lift(Map(x[2:9] -> (-inf,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf), #result[0:0] -> (-inf,+inf))) Function main exit
[]: Lift(Map(x[2:9] -> (-inf,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (0,+inf), main(){...}:1:1 -> (+inf,-inf))) [Stmt] x = 0;[15:13]
[]: Lift(Map(x[2:9] -> (0,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (0,+inf), main(){...}:1:1 -> (+inf,-inf))) [Stmt] (1000 > x)[8:20]
[5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (0,+inf), main(){...}:1:1 -> (+inf,-inf))) []: Lift(Map(x[2:9] -> (0,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [Stmt] assert (1000 > x);[9:17]
[5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (0,+inf), main(){...}:1:1 -> (+inf,-inf))) []: Lift(Map(x[2:9] -> (0,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [Stmt] x = 0;[12:17]
[5:12]: Lift(Map(i[2:12] -> (1,1), x[2:9] -> (1000,+inf), main(){...}:1:1 -> (+inf,-inf))) []: Lift(Map(x[2:9] -> (1000,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [Stmt] x = (x + 1);[10:17]
[5:12]: Lift(Map(x[2:9] -> (0,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) []: Lift(Map(x[2:9] -> (0,+inf), i[2:12] -> (1,1), main(){...}:1:1 -> (+inf,-inf))) [Stmt] i = input;[17:9]

Figure 2: Interval Loop Unrolling Analysis Result

As is shown in the result of the analysis Figure 2, the interval of the result variable: x is still the same as before namely: $[0, +\infty]$. There is now a additional context, for which an interval analysis was performed, but the result of the analysis is the same.

4 Discussion Point 3

This section will discuss the results of the third discussion point.

4.1 Context

Question: Which variables would you include in the context for functional loop unrolling?

Since the bases of functional sensitivity is on the abstract state, it would at least start of with the variable(s) defined in the predicate of the while loop. The more variables added to the context that are defined/used inside of the loop the more precision is gained. Increasing the size of the state to be stored in the context, comes with the drawback that performance might be reduced.

Continuing from the context with at least variable i. The variable x, defined in the loop may also be added.

4.2 Functional vs Callstring

Question: Write a TIP program where functional loop unrolling improves precision compared to callstring loop unrolling, and explain the difference.

4.2.1 Program

```
i = 1;
x = input

while (i > 0) {
  assert i > 0;
  if (i % 2 == 0) {
    x = x + 1;
  } else {
    if (x > 0) {
      assert x > 0;
      x = x - 1;
    }
  }
  i = input;
}

return x;
```

Listing 8: Example program - functional loop unrolling.

4.2.2 Difference

Because of the limitations on (k)-callstring loop unrolling, the analysis will lose the relation between the $i\%2 == 0$ on iterations that are larger than k.

In contrast the functional loop unrolling will be able to 'store' this relation for longer, since for each abstract state a new context is created. Thus maintaining the periodic relation of the $i\%2 == 0$ predicate in memory.

4.3 Finite

Question: Does interval analysis with functional loop unrolling terminate for every program? Explain why or why not (give an example).

Applying the practice of loop unrolling to functional sensitivity does not change the fact that for some given programs the analysis will **not** terminate. An example for such a program can be seen in Listing 9.

```
x = 0;

// First iteration
x = x + 1;

// While iteration
while (true) {
  x = x + 1;
}
```

Listing 9: Example program - functional loop unrolling.

As with functional sensitivity for each abstract state of the program, in this the while loop a new context is generated [2], [3]. Unrolling the first iteration of the loop as displayed in the above program, does not terminate for the given program, since the size of the state (on which functional sensitivity based itself) is not finite in this case. Therefore when considering functional sensitivity, the chosen state is to be considered carefully [2].

Bibliography

[1] C. H. Flemming Nielson Hanne Riis Nielson, *Principles of Program Analysis*. Addison-Wesley, 2015. doi: https://doi.org/10.1007/978-3-662-03811-6.

[2] [Online]. Available: https://cs.au.dk/~amoeller/spa/7-interprocedural-analysis.pdf

[3] [Online]. Available: https://dl.acm.org/doi/fullHtml/10.1145/3230624