

# Assignment 3: Dataflow Analysis

Milan Lagae

Institution/University Name:

Vrije Universiteit Brussel

Faculty:

Sciences and Bioengineering Sciences

Course:

Software Quality Analysis

## Contents

1	Intro .....	1
2	Discussion Point 1 .....	2
2.1	Implementation .....	2
2.1.1	Asserts .....	2
2.1.2	Widen Interval .....	2
2.1.3	Assignment(s) .....	2
2.2	Results .....	2
2.3	Analysis Precision .....	2
3	Discussion Point 2 .....	2
3.1	Implementation .....	3
3.1.1	Context .....	3
3.1.2	Unrolling .....	3
3.2	Results .....	3
4	Discussion Point 3 .....	3
4.1	Context .....	3
4.2	Question 2 .....	3
4.2.1	Program .....	4
4.2.2	Difference .....	4
4.3	Finite .....	4
	Bibliography .....	5

## 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 ValueAnalysis.scala 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, PInfin)`.

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

Listing 1: Assert - Version 1

```
1 // value >= number
2 case ABinaryOp(GreatThan, ANumber(i, _), id: AIdentifier, _) =>
3   val xDecl = id.declaration
4   // Get the interval for the declaration
5   val old = s(xDecl)
6   // Create the new interval by applying (zero is ignored?)
7   val newInterval = widenInterval(old, (i, MInf))
8   // Update with the new interval
9   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 list of four values, as shown in Listing 3.

```
1 case ((l1, h1), (l2, h2)) => {
2   IntervalLattice.intersect((l1, h1), (l2, IntervalLattice.PInf))
3 }
```

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.

```
1 // var declarations
2 // <vi>= <x=E>= JOIN(vi)[x ↦ eval(JOIN(vi), E)]
3 case varr: AVarStmt =>
4   varr.declIds.foldLeft(s) { (state, decl) =>
5     state.updated(decl, valuelattice.top)
6   }
7 }
```

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.

```
1 // assignments
2 // <vi>= JOIN(vi)
3 case AAssignStmt(id: ALIdentifier, right, _) => {
4   val interval = eval(right, s)
5   s.updated(id, interval)
6 }
```

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.

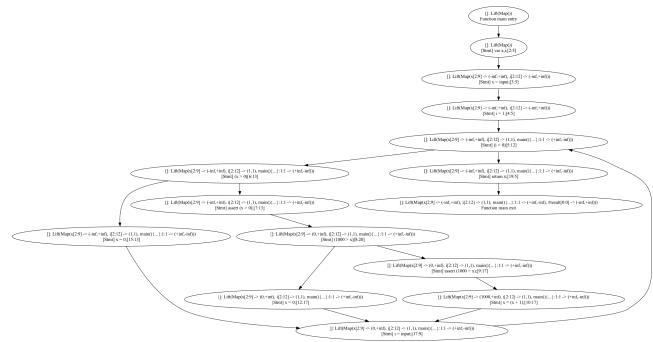


Figure 1: Interval Analysis Result

### 2.3 Analysis Precision

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

## **TODO: Most precise result**

### 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.

```
1 // MOD-DP2
2 def makeLoopContext(c: CallStringContext, n: CfgNode, x: state lattice.Element): CallStringContext = {
3   // Add node to call string context, while maintaining limit on
4   // context length
5   CallStringContext((n :: c.cs).slice(0, maxCallStringLength))
6 }
```

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 current-Context, 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.

```
1 ////////////////////////////////////////////////////////////////////  
2 // Discussion Point 2: COMPLETE HERE  
3 // Thus, to determine the starts and ends of loops you must use the  
4 // cfg.dominators function.  
5  
6 // case m: CfgStmtNode if loophead(n) => {  
7 //   val node = m.data  
8 //   val loopStart = (m.succ intersect dominators(m)).head  
9 //   val newContext = makeLoopContext(currentContext, loopStart,  
10 //     s)  
11 //   propagate(s, (newContext, m))  
12 // }  
13 //  
14 // s
```

Listing 7: Loop Unrolling

### 3.2 Results

The results of executing the interval analysis with loop unrolling on the looppjject.tip example file with the following

command: `./tip -interval wlrw vubexamples/loopproject.tip` can be seen in Figure 2.

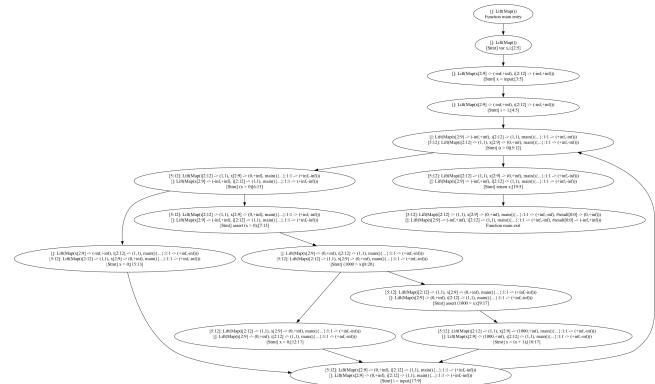


Figure 2: Interval Loop Unrolling Analysis Result

## TODO: Add

## 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 off 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 Question 2

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

TODO: Add

#### 4.2.1 Program

```
1 x = 1;
2 y = input
3
4 while (i > 0) {
5   assert i > 0;
6   if (i % 2 == 0) {
7     x = x + 1;
8   } else {
9     if (x > 0) {
10       assert x > 0;
11       x = x - 1;
12     }
13   }
14 }
15
16 return x;
```

when considering functional sensitivity, the chosen state is to be considered carefully [1].

Listing 8: Example program - functional loop unrolling.

**TODO: Add**

#### 4.2.2 Difference

**TODO: Add**

### 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.

```
1 x = 0;
2
3 // First iteration
4 x = x + 1;
5
6 // While iteration
7 while (true) {
8   x = x + 1;
9 }
```

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 [1], [2]. 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

## Bibliography

- [1] [Online]. Available: <https://cs.au.dk/~amoeller/spa/7-procedural-analysis.pdf>
- [2] [Online]. Available: <https://dl.acm.org/doi/fullHtml/10.1145/3230624>