CSSE4630 Week 6 Lab: Interval Analysis

Mark Utting

Version 1.0

1 Introduction

This workshop is focussed on Interval Analysis, especially understanding and implementing *widening* of intervals.

2 Review: Widening of Intervals

Widening is a general technique that can be applied to many different kinds of analyses and lattices. However, it is particularly useful and important for interval analysis, where we deduce $[min, max]$ bounds for each integer variable (the same idea can be applied to floating-point variables as well, but that's another story). Naive interval analysis can go into an infinite loop when it is analysing loops that contain statements like `i = i+1`, because the lattice of intervals is infinite, so we might keep going up it forever, without reaching a fixpoint! But a widening operator makes larger jumps up the lattice, to ensure that we terminate after a finite number of iterations.

Review pages 80-81 of the textbook, to ensure that you understand the *widening operator* ∇, which takes two intervals for a variable, from iterations i and $i+1$, and decides whether to increase the upper bound or not, and similarly whether to decrease the lower bound or not.

Assuming that the set of integer constants in a program is $B = \{0, 1, 10, 100\}$, evaluate the following widening operator expressions:

1. $[1, 5]\nabla[1, 5]$
2. $[1, 5]\nabla[1, 6]$
3. $[1, 10]\nabla[1, 11]$
4. $[1, 95]\nabla[1, 100]$
5. $[1, 95]\nabla[1, 105]$
6. $[1, 95]\nabla[0, 95]$
7. $[-1, 900]\nabla[-2, 900]$

3 Implementing Interval Analysis

The Live Variables analysis in `src/tip/analysis/LiveVarsAnalysis.scala` is not fully implemented. To see this, run the 'tip' script with the `-livevars` option, on the example program `liveness.tip`.

```
tip -interval wlrw examples/interval3.tip
```

You should see an error about unimplemented methods in `ValueAnalysis.scala`. Fix this problem by completing the implementation of `localTransfer` in `src/tip/analysis/ValueAnalysis.scala`.

You 'just' need to implement the missing code (the triple question marks) inside the `localTransfer` method. This takes the node n as input, plus the state-lattice s, which is the interval analysis state just before this node. There are two cases that you need to complete:

case varr: AVarStmt Here you must extend `s` by adding (using `++`) all of the variables in `varr`. To do this, use a 'list comprehension' (`for (v<-varr.declIds) yield ...`) to iterate through `varr` and map each variable to the bottom of the `valuelattice`.

case AAssignStmt(id: AIdentifier, right, _) Here you need to override just the variable `id` in the map `s` (using `s+(_->_)`). Since `s` maps *declarations* to intervals, make sure you convert `id` to a declaration using `id.declaration`. Map it to the abstract evaluation of the *right* expression in the current state (`s`).

Note: Don't spend too much time implementing these $localTransfer$ cases. If you spend more than 10 minutes on them, there are some sample solutions at the end of this document.

4 Implementing Widening of Intervals

After you have implemented the ValueAnalysis.localTransfer function fully, if you run tip again as above, you should get a new exception showing that an implementation is missing in the `widenInterval` method in `src/tip/analysis/IntervalAnalysis.scala`.


```
[error] scala.NotImplementedError: an implementation is missing
[error]         at scala.Predef...(Predef.scala:288)
[error]         at tip.analysis.IntervalAnalysisWidening.widenInterval(IntervalAnalysis.scala:38)
[error]         at tip.analysis.IntervalAnalysisWidening.widenInterval(IntervalAnalysis.scala:34)
```


You need to implement the main case of $(l1, h1)\nabla(l2, h2)$, which is when both intervals are non-empty. Implement this in the following stages:

1. **None:** Just return the right-hand interval $(l2, h2)$ unchanged. This is effectively the base case of implementing no widening at all! (Actually this is not technically a proper widening implementation, since every widening operator should reach the top of the lattice in a *finite* number of steps). Try analysing the example program again. What happens?

2. **Worst-Case:** Now implement the worst case widening — always widen to the top interval `(MInf, PInf)`. Try analysing the example program again. What happens? Look at the output file `out/interval3.tip_normalized.tip` and see what intervals it has deduced for each node? You should see that, apart from a few integer constant cases like `(0,0)`, almost all the intervals have ended up as `(MInf, PInf)`. This is not a very useful analysis result!

3. **Jump-to-Infinity:** Improve the previous analysis by testing to see if the upper bound is staying the same ($h1 < h2$) or getting worse ($h1 < h2$). If it is staying the same, then return that bound, but if it is getting worse then return `PInf`. Similarly for the lower bounds — if $l2 < l1$ then return `MInf`. Analyse the example program again and look at the output file (preferably using GraphViz). You should see an output graph like the one shown in Fig. 1.

4. **B-Smart:** Finally, let's use the set B of integer constants that appear the program. If you detect an upper bound that is getting worse, then instead of jumping directly to `PInf` just jump to the next larger value in B. Similarly (but going downwards) for lower bounds that are getting worse. Note that the set B (including `MInf` and `PInf`) is already calculated for you in IntervalAnalysis.scala. There is also a helper method `minB(a)` that

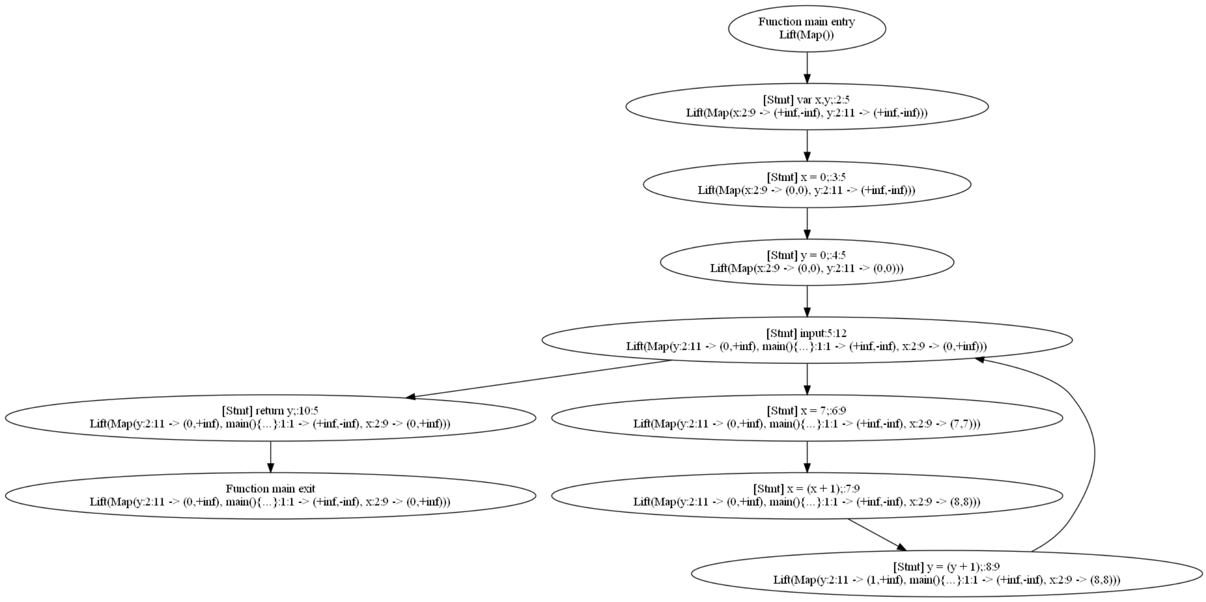

Figure 1: Results of the simple 'Jump-to-Infinity' widening for `interval3.tip`.

returns the minimum value in B that is larger than a, and similarly `maxB(a)` for the opposite direction.

To test your smart implementation, analyse the `interval3.tip` program again and inspect the output. It should be the same as the one shown in Fig. 1, because the largest constant in this program is 7, and x is incremented just past 7, so gets widened to `PInf`! To try a more enlightening example, make a copy of `interval3.tip` called `interval10.tip` and change line 3 from `x=0` to `x=10`, then run your interval analysis on this modified program. You should see an output graph like the one shown in Fig. 2, with the final (exit) interval for x of $(7, 10)$ instead of $(0, PInf)$. Well done!

5 Narrowing?

Finally, after you have implemented the smart widening using B, rerun your analysis of both `interval3.tip` and `interval10.tip` using the widening+narrowing solver (`wlrwn`). That is, use a command line like:

```
tip -interval wlrw examples/interval3.tip
```

What difference do the extra narrowing steps (done after widening) make to these programs?

6 Help!

If you get stuck with Scala, Google it, or use the docs:

- ScalaBook Prelude: `https://docs.scala-lang.org/overviews/scala-book/prelude-taste-of-scala.html`
- CheatSheet: `https://docs.scala-lang.org/cheatsheets/index.html`
- Main Docs page: `https://docs.scala-lang.org`

Note: if you are familiar with Python list comprehensions, the rough equivalent in Scala is to use a 'yield' expression with a 'for' loop, in order to produce a list of values:

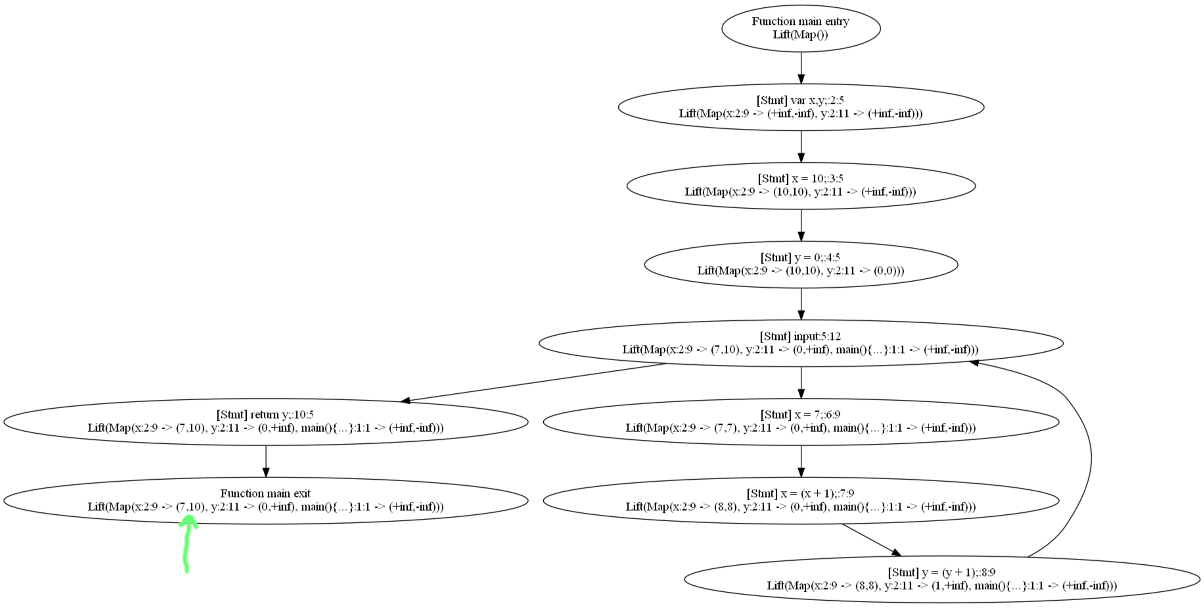

Figure 2: Results of the smart widening for `interval10.tip`.

```
scala> for (i <- 0 until 8) yield i*i
res0: scala.collection.immutable.IndexedSeq[Int] = Vector(0, 1, 4, 9, 16, 25, 36, 49)

scala> for (i <- 0 until 3) yield (i, i*i)
res1: scala.collection.immutable.IndexedSeq[(Int, Int)] = Vector((0,0), (1,1), (2,4))
```


You can experiment with these kinds of Scala expressions using the IntelliJ **Tools / Scala REPL...** menu.

6.1 Sample ValueAnalysis.localTransfer answers


```
// var declarations
case varr: AVarStmt => s ++ (for (v <- varr.declIds) yield (v,valuelattice.bottom))

// assignments
case AAssignStmt(id: AIdentifier, right, _) => s + (id.declaration -> eval(right, s))
```