CSSE4630 Week 5 Lab: Implementing Live Variables Analysis

Mark Utting

Version 1.0

1 Introduction

In this workshop, we will implement the missing parts of the Live Variables analysis in the Scala implementation of TIP:

Reminder: If you have not set up Scala, SBT and TIP on your computer previously, do this now — see the Week 3 workshop instructions.

To check if you have it correctly installed, you should be able to run the 'tip' script at the top level of the TIP repository. Right click on it in IntelliJ and do `Run tip`. This should show a help message in the console output. We want to run this script with parameters corresponding to the following command line:


```
tip -run examples/constants2.tip
```

Note: You can also do this from within IntelliJ: right-click on the 'tip' script and use the `Edit tip...` menu item to edit the run configuration to add the additional parameters — do not include the initial 'tip' script name.

2 Live Variables 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 -livevars -run examples/liveness.tip
```

You should see an error about unimplemented methods...

Fix this problem by completing the implementation in `LiveVarsAnalysis.scala`.

You 'just' need to implement the missing code (the triple question marks) inside the `transfer` method. There are four cases that you need to complete.

Recall that the Live Variable analysis rules are shown in Fig. 1.

The Scala `transfer` function is just implementing the transfer function for each node, so the $JOIN(v)$ has already been done and the input parameter `s` is the result of that join. So your code must just implement the updates after the join.

Hints:

- You can update a Scala map `m` with a single `key` and `value` entry by writing:

  ```
  m + (key -> value)
  ```

$$[\![X = E]\!] = JOIN(v) \backslash \{X\} \cup vars(E)$$
$$[\![var\ X_1, \ldots, X_n]\!] = JOIN(v) \backslash \{X_1, \ldots, X_n\}$$
$$[\![if\ (E)]\!] = JOIN(v) \cup vars(E)$$
$$[\![while\ (E)]\!] = JOIN(v) \cup vars(E)$$
$$[\![output\ E]\!] = JOIN(v) \cup vars(E)$$
$$[\![exit]\!] = \{\}$$
$$[\![v]\!] = JOIN(v) \text{ for all other kinds of nodes } v.$$

Figure 1: Live Variables analysis rules

- You can update *multiple* entries in the map by using the `++` operator with a list of `(key,value)` pairs, or remove a list of variables by using the `--` operator.
- If you have an `AIdentifier` object called x, you can get to its corresponding `ADeclaration` object via x`.declaration`.
- If you need to get the set of variables in an expression E (i.e. $vars(E)$), then the E`.appearingIds` method is your friend.
- Type Errors? I had some situations where the Scala typechecker reported type errors even though both types appeared to be the same. This can happen if there are two different 'paths' to the same type, but those two paths are different instantiations of the same class. So one way of fixing this is to give that instantiation a name, and consistently refer to it using that name. This means that the class is instantiated once, rather than multiple times. For example, I had trouble with the typechecker not recognising `lattice.sublattice.bottom` as correctly typed. So I gave the instantiated PowersetLattice object a name as follows:

```
val varsLattice = new PowersetLattice[ADeclaration](allVars)
```

and then used this `varsLattice` name everywhere, replacing all the other instantiations: `new PowersetLattice...` and the typechecker was happy. I think this is related to the path-dependent-types feature of Scala.

2.1 Checking Your Results

To see if you have implemented the analysis correctly:

1. Recompile the LiveVarsAnalysis.scala file. No errors is the first good sign.
2. Run tip as follows: `tip -livevars -run examples/liveness.tip`
3. If that works without errors, it will create an output file `out/liveness__livevars.dot`. You can view this file with a text editor to see the set of live variables at each control-flow node.
4. But even better is to view this output file graphically — *.dot files are graph files that can be viewed either by installing the GraphViz tools from `https://graphviz.org`, or by using a web-based GraphViz viewer such as `http://www.webgraphviz.com`.

You should see an output graph like the one shown in Fig. 2. Note that the two colon-separated numbers after each statement and variable name are the line number and the column number where that statement appeared in the source program, or where the variable was declared.

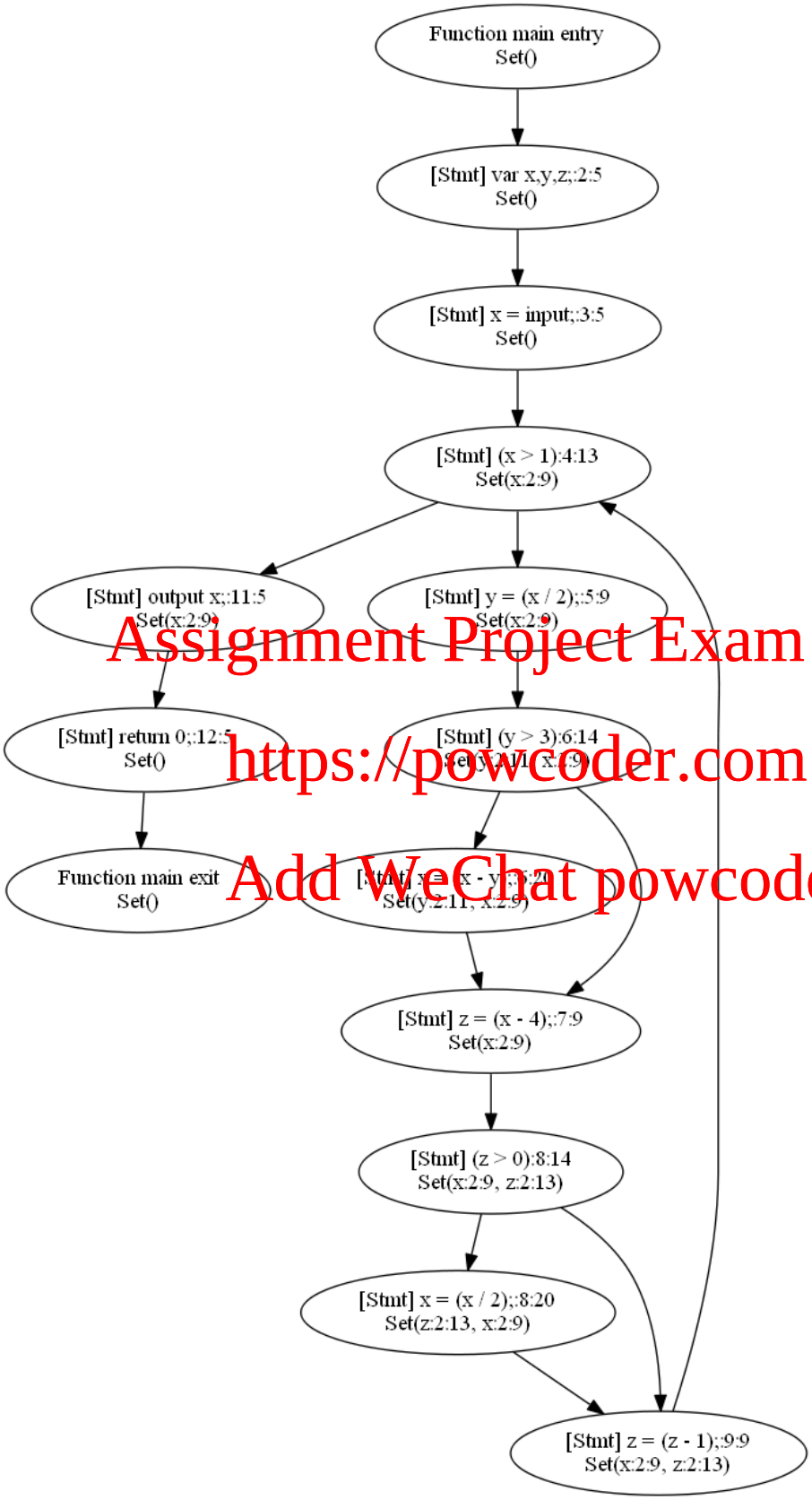

Figure 2: Results of Live Variables Analysis for TIP/examples/liveness.tip

3 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:

```
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.