CSSE4630 Week 3 Lab: Dataflow Analyses

Mark Utting

Version 1.0

1 Introduction

Most statically typed programming languages require you to define a type for each variable and parameter when you declare it.

In this lab, we are going to experiment with a static type system that requires no such explicit typing information. Instead, it tries to infer types, and is also *flow-sensitive* in that it can track how types change in different branches of a program.

We will also start writing Scala code, to test if an abstract operator is correctly defined (monotone).

2 (Textbook Ex. 5.35) Design a flow-sensitive type analysis for TIP.

In the simple version of TIP we focus on in this chapter, we only have integer values at runtime, but for the analysis we can treat the results of the comparison operators $>$ and $==$ as a separate type: boolean. The results of the arithmetic operators $+$, $-$, $*$, / can similarly be treated as type integer. As lattice for abstract states we choose:

$$States = Vars \to 2^{\{integer, boolean\}}$$

such that the analysis can keep track of the possible types for every variable.

1. Decide if this analysis will be forwards or backwards? A 'may' analysis or a 'must' analysis?
2. Which of the four elements in your lattice will give a warning when that value is used as a condition (in an 'if' or 'while' statement)?
3. Which of the four elements in your lattice will give a warning when that value is used as an argument to an integer operator like $+$?
4. Draw the lattice, and define what your JOIN operator (least upper bound) must do.
5. Draw up an abstract operator table for each operator (just the $+$, $>$ and $==$ operators will be enough to give you the idea). Identify each cell in the table that should generate a type warning (e.g. colour it red, or append an exclamation mark to the result type in that cell).
6. Specify constraint rules for the analysis (for each type of TIP node).
7. After analyzing a given program, how can we check using the computed abstract states whether the branch conditions in `if` and `while` statements are guaranteed to be booleans? Similarly, how can we check that the arguments to the arithmetic operators $+$, $-$, $*$, / are guaranteed to be integers? As an example, for the following program two warnings should be emitted:

```
main(a,b) {
  var x,y;
  x = a+b;
  if (x) { // warning: using integer as branch condition
    output 17;
  }
  y = a>b;
  return y+3; // warning: using boolean in addition
}
```

Use your static analysis to analyse this program and check that it does indeed generate those two warnings and no other warnings.

8. Now add the line `a = (a==b)` just before the 'if' statement, and analyse the program again. Do you get any changes in the warnings? It is desirable to get an extra warning now, from the `a>b` expression.

9. Move that `a = (a==b)` line inside the 'if' statement, just after the `output 17` and analyse the program again. How many warnings do you get now? Do you think your solution is a good one?

3 Testing that operators are Monotone

Ok, now we get to try our hand at Scala programming!

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`

3.1 Setup

[The following instructions are adapted from the TIP repository on GitHub - see there for extra instructions and alternatives.]

Set up TIP (and Scala) using IntelliJ IDEA:

1. Make sure you have JetBrains IntelliJ installed.
2. Install the Scala Build Tool (SBT) on your computer, from `https://www.scala-sbt.org`
3. In IntelliJ settings (**File / Settings / Plugins**), install the 'Scala' plugin from JetBrains.
4. Use IntelliJ to clone the TIP repository (**File / New / Project from Version Control / Git**). Enter the TIP repository URL `https://github.com/cs-au-dk/TIP`. This will clone the repository onto your computer with the directory name tip. (If you use another directory name, then the next step may fail!) IntelliJ should then detect an SBT project. Click 'Import SBT project' and follow the instructions. (If this is your first Scala project, you will need to setup the Scala SDK.)
5. Right-click on Tip.scala in src/tip, then select "**Run 'Tip'**". To supply arguments, use **Run... / Edit Configurations** in the **Run** menu.

6. Note: If you cloned the repository using git instead of IntelliJ, you will need to import the project from the SBT model (**File / New / Project from Existing Sources**). Since the .idea folder is then regenerated from scratch, in order to keep the inspection profiles you need to checkout the .idea/inspectionProfiles folder and the .idea/codeStyles folder from the original repo, leaving the other generated folders untouched.

3.2 Testing the $+$, $>$ and $==$ operators

Your task is to implement a method in TIP that can check whether an operation as defined in the absPlus, absGt, and absEq tables in SignLattice in **SignLattice.scala** is monotone. (Optional: add a test script to test that all the tables in SignLattice are monotone.)

Recall that a unary function f is monotone iff:

$$\forall a, b \;.\; a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$$

For a binary function, like $a + b$, we must check that it is monotone in its first argument, and then also check that it is monotone in its second argument. So we need both of the following conditions to hold:

$$\forall a, b, c \;.\; a \sqsubseteq b \Rightarrow f(a, c) \sqsubseteq f(b, c)$$

$$\forall a, b, c \;.\; a \sqsubseteq b \Rightarrow f(c, a) \sqsubseteq f(c, b)$$

For simplicity, start by just adding your tests into a 'main' method in **src/tip/lattices/SignLattice.scala**. This will allow you to run your tests directly by clicking on the green run button next to the 'main' method in IntelliJ.

Hint: here is a 'main' method to get you started. Scala looks a bit like a combination of Python and Java, doesn't it!


```
def main(args: Array[String]): Unit = {
  println("Testing plus...")
  for (a <- signValues.keys) {
    val b = Neg
    val out = plus(a, b)
    println(s"$a + $b gives $out")
  }
}
```

When you have written and run your test, you should find that the $+$ operator is monotone. So your test passes! But I never trust a test if it passes the first time it is run...

So, to check that your test correctly detects non-monotone entries, change the last entry on the bottom row of the absPlus table from Top to Pos. Then rerun your test and check that it detects the non-monotone behaviour and gives a helpful error message.

Make sure you test at least three operators: $+$, $>$, and $==$.

Advanced Challenge: to make your tests more concise, can you define a helper method that just takes the operator-to-test as a higher-order function parameter?