Lab 4: IFDS / IDE

In this lab we implement an inter-procedural linear constant propagation in the IFDS and IDE frameworks, using the Soot implementation for IFDS/IDE: *Heros*. A linear constant propagation propagates the values of constant variables or variables that depend on constants throughout the program. It is typically used in compiler optimization to reduce the amount of instructions that must be evaluated at runtime. In this lab, you will implement a linear constant propagation in IFDS. Then, you will implement a, IDE version of the linear constant propagation, which is more complex, but yields a more efficient and scalable solution for real-world programs.

Consider the following code:

```
1  class MySimpleClass {
2
3          public int add(int x, int y) {
4                  int z = x + 1;
5                  return z;
6          }
7
8          public static void main(String args[]) {
9                  int a, b, c, d, e;
10                 a = 0;
11                 b = 1;
12                 c = add(a, b);
13                 d = add(c, b);
14                 c = 0;
15                 d = add(c, b);
16                 e = 1;
17         }
18
19 }
```

Figure 1: A simple program using some constant values.

A linear constant propagation propagates constant values that satisfy the linear equation $c = a \cdot x + b$ (with a and b some constant integer literals); it would find that at line 16 the following flow facts $d_i \in D$ hold - with $D = \{\langle a, 0\rangle, \langle b, 1\rangle, \langle c, 0\rangle, \langle d, 1\rangle, \langle e, 1\rangle\}$.

General Instructions

Set up: The code in folder `DECALab4` contains a maven project readily set up. To set up your coding environment:

- Make sure that you have any version of Java running on your machine.
- Import the maven project into your favorite development environment.
- To run the test cases, run the project as a Maven build, with the goals set to "clean test".

- **Important: Make sure that your test cases run via the command line.** To do so, run `mvn clean test` in the `DECALab4` directory.

Project: The project contains:

- `JUnit` tests in the package `test.exercises` of the directory `src/test/java/test/exercises`. Your goal is to make all of those test cases pass.
- target classes in the packages `target.exercise1and2` of the folder `src/test/java`. The test cases are run on those classes.
- two classes containing the flow functions of the two linear constant propagations you will develop in this lab. They are located in the package `analysis` of the folder `src/main/java`. For each exercise, you only need to provide correct implementations for the currently defaulted functions in the classes `IFDSLinearConstantAnalysisProblem.java` and `IDELinearConstant` in the package `analysis`. Do not edit any of the other files. There are `TODO` comments in the code that help you finding the right spot for your implementation.

Submission format: Submit on PANDA a file `DECALab3_user1_user2_user3_user4.zip` (with `user{n}` your PANDA usernames), containing the project with your solution.

Exercise 1

The IFDS analysis is already setup for you. The type of the data-flow facts used is `Pair<Local, Integer>`. Your IFDS linear constant propagation tracks tuples of `Local` $\times$ `Integer` in order to propagate constants throughout the program under analysis.

Observe the code:

```
1 public static void main(String args[]) {
2         int i = 10;
3         int j = 20;
4         i = 30;
5 }
```

Figure 2: Simple main

In this IFDS implementation of linear constant propagation, the lattice has been chosen such that the analysis stabilizes with the following data-flow facts after line 4 in Figure 2: $D = \{\langle i, 10\rangle, \langle j, 20\rangle, \langle i, 30\rangle\}$

Start working on the `getNormalFlowFunction()` factory method. This will allow your implementation to perform intra-procedural analyses. You can leave the other flow function factories for the moment. When having implemented this flow function factory the first test cases should already pass.

Next, extend your linear constant propagation to an inter-procedural analysis by implementing the `getCallFlowFunction()` factory such that the data-flow facts from the caller's context are mapped into data-flow facts in the callee's context.

Then, implement `getReturnFlowFunction()` to map the data-flow facts that hold at the end of a called function back into the caller's context.

At last, implement the `getCallToReturnFlowFunction()`. Restrict your implementation to transfer only the data-flow facts that are not involved in a call.

All test cases for the IFDS linear constant propagation should pass.

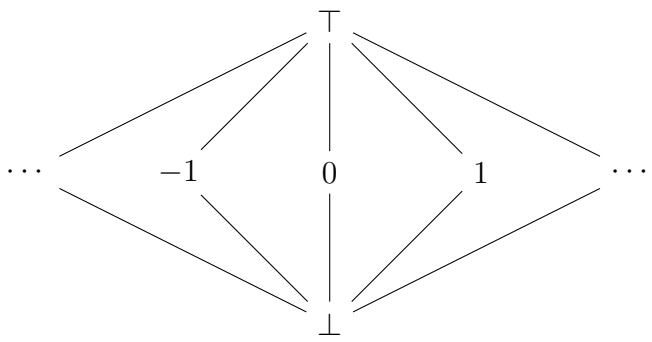
Hints:

- You are free to use the pre-defined flow functions for common tasks defined in the package `heros.flowfunc` like `Gen`, `Kill`, etc..
- You can use the values `-1000` and `1000` as the lower and upper bounds of your analysis: a constant's value that exceeds this interval shall not be tracked. You will notice that the `JUnit` tests involving loops will emit an undesired behavior if you do not incorporate this special treatment.

(5 points)

Exercise 2

In this exercise, you will implement a linear constant propagation using the IDE framework. In this implementation of an IDE-based linear constant propagation, you will use variables as your data-flow facts. The values of the variables are tracked using the edge functions. This is more elegant and more optimized but also more complex to implement. In this version of linear constant propagation, you will use the following lattice. You will need to model it explicitly by overriding the adequate interface methods:



The value domain in this exercise will be $\mathbb{Z}$ (or `Integer` in Java).

In order to encode your lattice, complete the interface factory methods `createJoinLattice` and `createAllTopFunction`. `createJoinLattice` lets you encode the lattice's values. The functions `topElement` and `bottomElement` refer to the top and bottom elements of the lattice. `join` tells the analysis framework how to join two values: how to go up in the lattice.

Then, implement the flow function factories like in the IFDS implementation. Track the flow facts through the program and do not worry about the edge functions yet.

Finally, implement the edge functions that describe the value computation that is performed alongside the edges in the exploded super-graph.

- `EdgeFunction`s are function objects that describe a value computation in IDE alongside edges in the exploded super-graph that can be evaluated (using `computeTarget()`), composed (using `composeWith()`), joined (using `joinWith()`), and compared (using `equalTo()`).
- Like for the flow functions, implement `getNormalEdgeFunction()` in order to obtain an intra-procedural analysis.
- Then, implement the functions `getCallEdgeFunction()`, `getReturnEdgeFunction()` and `getCallToReturnEdgeFunction()` in order to obtain a inter-procedural analysis.
- All these factory functions return an `EdgeFunction` that expresses how a statement affects the value of a data-flow fact.

All test cases should pass.

Hints:

- One way of determining the top and bottom values of the lattice is related to `Integer`'s `MIN_VALUE` and `MAX_VALUE`.
- In the literature, notations for the top and bottom element are not consistent across the world. Sometimes, $\bot$ represents the element at which everything converges (over-approximation reached after going up the lattice) and $\top$ represents the initial element (no information). In this course, and as shown in the lattice diagram in this sheet, we use $\bot$ as the initial element and $\top$ as the over-approximated element. *Heros* does the contrary: it uses $\bot$ as the over-approximated element and $\top$ as the initial element. Thus, the interface method `createAllTopFunction()` must be implemented to return the function representation of the initial element (no information / $\bot$ in the course / $\top$ for *Heros*). Additionally, you must implement a special edge function object for your analysis that represents the over-approximated element ($\top$ in the course / $\bot$ for *Heros*) that denotes the most imprecise element that you can make use of in your implementation of the `EdgeFunctions` interface.
- It is extremely helpful to use a debugger.
- When implementing the IDE analysis draw the exploded super-graph (IFDS/IDE's underlying data structure) for a given target program on a piece of paper before starting to encode flow- and edge-functions in Java.

(5 points)