Project 1

Dilawer Ahmed 20100177
Talha Amir 20100172

Implementation of Framework

Framework is implemented as described in the handout.
It requires the direction (forward,backward) in a Boolean variable with true for forward and false for backward.
It also requires the meet operator which is also a Boolean variable with true for union and false for intersection
It requires a transfer function that returns the bit vector “out”(out will be defined later) and gives the “in” bitvector and domain index which is essentially a mapping to variables. It also receives the block in should run on and all other blocks in and out in case it needs them for handling
There are two phases in framework and setup(runPassSetup) and a run phase (runPassFunction) which work as follows

Setup:
It sets up the boundary and initial conditions and in and outs depending on the direction. If direction is forward the “out” is the out for the basic block and if the direction is backwards the “out” is actually input of the basic block but for ease if implementation it is defined that way.
It also sets up an ordering of the basic blocks that is required by the run phase which is essentially and order in which the block would be visited.

Run:
Run basically goes over the blocks in that calling meetop and transfer function and checking if it converges to stable values. And sets the “in” and “out” of blocks accordingly

Passes

<u>Available Expressions:</u>

For available expressions, the domain of instructions was first constructed, and an array domainIndex was maintained which contained mapping of instruction pointers to their respective index in the bitvector of the inputs reaching the block. Next the set of generated and killed definitions was calculated,the assumption used was that if for a particular instruction its index in the input block was 0, it was being generated, whilst if it was 1 then it was being defined or re-defined if you put it as such. Lastly, the transfer function was applied, this was done using a simple loop over the entire bitvectors and setting the bits accordingly.

Liveness (Live Variable Analysis):

As done previously, a domain was first constructed of the instructions, arguments and phi node instruction and arguments. There were two main portions to the liveness pass, one was handling the values used by the instruction, this was fairly simple and the template code mentioned in the handout proved to be useful. The value was checked to be either an instruction or an argument, then was checked if it was defined in the basic block and if not was added in the use bitvector, as we are considering locally exposed uses for our use bitvectors. The second, and main portion was the handling of phi nodes. For this the documentation and phi node functions in llvm proved useful. The way phi-nodes were handled was that considering the nodes represent alternate points in the in our program paths, if the value we had was a phi node, its incoming values and blocks were extracted using the llvm instructions for phi-nodes. For all the incoming values, the pass would find the incoming blocks to our current phi-node, and these blocks were assumed to be points where the value would exist, and accordingly, the bitvectors containing representations of these particular values would be updated, more specifically, the output bitvectors, since in our backward analysis and the way our framework was used, these are the immediate next point our pass moves on to.
The particular values that are added are then inserted into our use bitvector, since the assumption is that if they exist at one particular program point, the values would be live at the current. Lastly, the transfer function for live variable analysis is then applied to our bitvectors.