**Task 1: Designing Initialized Variables Analysis**

For this example, the analyser needs to capture at the exit of each block whichever variables have been initialized. In simple terms, it will require to add to the existing list of initialized variables in the program whichever are found in the block. If said variable had been explicitly initialized before, then it should just be ignored.

|  |  |  |
| --- | --- | --- |
| Label |  |  |
| 1 |  | b |
| 2 | b | b, f, |
| 3 | b | b, c |
| 4 | b, c, f | B, c, e, f |

 (1)



(2) (3)

 (4)

As such, we define the Monotone Framework as such:

Since the flow of the statements for this analysis can come either from the if.then as well as from the if.else, both of them are included in the Union of the final statement.

As it stands, the lattice for this analysis would be:

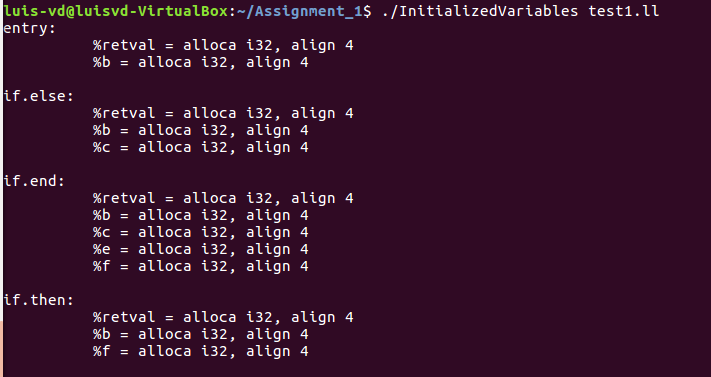
And is partially ordered by subset inclusion

**Task 2: Implementing the Initialized Variables Analysis in LLVM**

For this task, for every .bc or .ll file that is introduced into the LLVM Pass, the code will delve deep into the main part of the program and then analyse each of the Basic Blocks that are presented.

We assume that for this pass we will only be working with single not-nested Boolean cases. As such, we iterate through both cases, True and False and then we continue onto the common end of the program. We make use of multiple sets to artificially keep track of which ones belong to their respective label without including them in others. At the same time we keep a cumulative set which includes the variables that are initialized in either parts of the IF so that they can be all included in the set at the moment of the end.

Output observed for test1.ll:



The algorithm was composed in a way that only the leftmost result would print the pass, resulting in only the most complete labels to be printed in the end.

LLVM and Clang version 3.5 was used for this assignment to be at the same level as the one shown in the documentation from the Demo.

The following commands were run in the following orders to build and run the pass and output of the examples to be tested on the programs:

To build the Pass:

clang++-3.5 -o InitializedVariables InitializedVariables.cpp `llvm-config-3.5 --cxxflags` `llvm-config-3.5 --ldflags` `llvm-config-3.5 --libs` -lpthread -lncurses -ldl

To compile the .ll files:

clang-3.5 -emit-llvm -S -o <filename>.ll <filename>.c

To compile files as .bc instead:

clang-3.5 -emit-llvm -c -o <filename>.bc <filename>.c

To run the pass:

./InitializedVariables <filename>.bc

Unfortunately, it was not possible to have the pass optimized to run with Loops on time.