**SOLUTION DESCRIPTION**

For my solution, I have a generic template data flow class (DataFlowFrame) with pure virtual functions for the user to define to customize their dataflow analysis. The user can take this and inherit it to their class to simplify the process of creating a data flow analysis pass. LivenessFrame and ReachingDefinitionFrame takes that class and defines the pure virtual functions to create a specific type of data flow analysis object, and live and reach uses those respective objects to run a function pass to output the information.

***Generic Dataflow Framework***

The generic class allows the user to assign the desired Domain, data structure to hold the values, and Point, the scope for in and out sets, as templates to the object. Domain must be a data structure that stores llvm Values for correct print functionality through DataFlowAnnotator class and Point must be either llvm Instruction or BasicBlock as the program points to where the in and out sets will be determined. The object also takes as arguments the llvm Function to analyze, the direction of data flow as a Boolean (true forward, false backwards), and the initialization of in and out sets (true universal set, false empty set). Three functions are predefined with hooks for the user to define to complete the analysis. The createDataFlow function, once the data analysis object is created, can be called to analyze the function and save all the in and out states. Afterwards the getInState and getOutState will return the desired in and out set based on the desired program point (i.e. instruction or basic block pointer), respectively. The remaining functions are pure virtual hooks to define the main genFunction, killFunction, meetFunction, transferFunction and helper functions like domain specific union-ing function (unionSet) and emptySet, boundaryCondition values, and handlePhi function.

The main data flow logic occurs in the createDataFlow function. (The createDataFlow works at the instruction level and stores all domains/sets per instruction since it is the most specific program point necessary to attain the values and allows simple abstraction to locate in and out sets for any program points). This function is separate into two phases—the initialization phase and the in and out set creation phase. The initialization pass iterates over the function to create the gen and kill sets per instruction based on the user’s gen and kill function definition. The universal set is created and initially defines the in and out sets if desired, else the empty set will be used. Next, if forward analysis is desired, the predecessor instruction pointers are saved for each instruction to facilitate the creation process for the meet function, especially blocks with multiple inputs. And if backward analysis is desired, the successor instruction pointers are saved for each instruction for similar reason, especially blocks with multiple outputs. Finally, boundary conditions are set—in set defined by user for first instruction of function for forward analysis and out set defined by user for last instruction of function for backwards analysis.

The second phase iterates through the function, applying the meet function then transfer function for each instruction until the last instruction of the function is completed. The meet function takes the current instruction’s previous set (either all predecessor out sets for forward or all successor in sets for backwards) and the user calculates and defines the in set for forward analysis or out set for backwards analysis. The transfer function takes the gen and kill set and the user calculates and defines the out set for forward analysis or in set for backwards analysis. The phase then checks if either the two functions have changed the in and out sets they were modifying and repeats the process again if a change is found. Furthermore, the forward analysis is straightforward and does not require special handling of multiple predecessor phi functions by the generic class since there can never be multiple phi function predecessors (if instruction has multiple predecessors, then predecessors can only be multiple branch instructions). Therefore, current phi instructions can be handled by the user at the meet function when encountered. However, backwards analysis is trickier since multiple successors can be phi functions and requires further processing. The handlePhi function allows the user to alter the successor’s in sets before passing them to the meet function to be used. Phi functions when encountered as the current instruction to evaluate for in and out sets will never be the last instruction of a basic block and is assured to have only one successor instruction in set to deal with. This handle phi function is necessary to lessen the burden of the user when encountering phi instructions as successors since we force the user to modify the in sets of phi successors before running the meet functions on them.

***Liveness Framework and Live Pass***

The liveness class inherits the generic data flow class and provides SetVector<Value\*> as the domain and BasicBlock\* as program point. It is a backwards flow analysis with initial empty sets.

genFunction: when a Value is used, they are inserted to the gen set for that instruction

killFunction: when a Value is defined, they are inserted to the kill set for that instruction

meetFunction: combines all Values of previous successor in sets and are inserted to the out set

transferFunction: removes the kill Values from out set, and combines them with gen set to create the in set

unionSet: combines two setVectors and returns that set

boundaryCondition: the last instructions out set is the empty set since it will be the return of the function

emptySet: returns the clear set

handlePhi: if the current instruction’s successor is a phi function, only used Values of the phi function that came from the current instruction’s edge will be considered for the out set. Meaning only the correct used Values will propagate up to the correct BasicBlock that has the Value live. (The use value is only live out for the BasicBlock it corresponds to based on the phi node Value and BasicBlock pair).

The live pass creates the liveness frame object. creates the data flow analysis, and outputs the results using the dataflow annotator.

***Reaching Definition Framework and Reach Pass***

The reaching defintion class inherits the generic data flow class and provides SetVector<Value\*> as the domain and BasicBlock\* as program point. It is a forward flow analysis with initial empty sets.

genFunction: when a Value is defined, they are inserted to the gen set for that instruction

killFunction: when a Value is defined, they are inserted to the kill set for that instruction

meetFunction: combines all Values of previous predecessors out sets and are inserted to the in set

transferFunction: removes the kill Values from in set, and combines them with gen set to create the out set

unionSet: combines two setVectors and returns that set

boundaryCondition: the first instructions in set is the arguments of the function

emptySet: returns the clear set

handlePhi: nothing is done for predecessor phi nodes

When Phi nodes are encountered as the current instruction to be analyzed, they act like normal instructions, allowing for their used Values to not alter either the in or out set and killing the definition (which is nothing since this is SSA format)

The reach pass creates the reaching defintion frame object. creates the data flow analysis, and outputs the results using the dataflow annotator.

***Dataflow Annotator***

This is the skeleton dataflow annotator provided but the commented-out code can be used instead to allow instructions to be program point. (simple change to the program point to instruction of the liveness or reaching object and replacing the original with the commented code will print the instruction level in and out sets)

**STRUCTURE OF ARCHIVE**

* *DataFlowFrame*.*h*: defines the generic, template data flow analysis object
* *DataFlowAnnotator.h*: library for the printer functions for DataFlowFrame objects
* *LivenessFrame.cpp/h:* defines the liveness data flow analysis object
* *live.cpp/h:* contains a pass that analyzes the liveness and prints the in and out sets per block
* *ReachingDefinitionFrame.cpp/h:* defines the liveness data flow analysis object
* *reach.cpp/h:* contains a pass that analyzes the reaching definitions and prints the in and out sets per block
* */sample\_input*: folder contains all example input code
* *Makefile:* contains commands to compile code for live.so and reach.so
* *run.sh*: script to make and run liveness and reaching pass on user entered inputs

**SCRIPT CHANGES AND CODE TESTING:**

No changes were made to *run.sh* file

To test the code, simply enter “*./run.sh <sample input file without extension>.”*

Example: *./run.sh small*