Part 2

Due Date Jan 31 11:59:59 PM

Getting Started

If you are using docker,

> There is folder under "Passes" called "DFA" which contains `231DFA.h`. This is the file you will have to modify and use in this part of the project. You can also implement your pass in the same folder for convenience. As usual, after you launch Docker, the DFA folder will be found at `/LLVM_ROOT/llvm/lib/Transforms/CSE231_Project/DFA/`.

Dataflow Analysis Framework

You need to implement a generic intra-procedural dataflow analysis framework. We provide an incomplete base template class `class DataFlowAnalysis` in `231DFA.h`. To create a dataflow analysis based on the base `class DataFlowAnalysis`, you need to provide:

1. `class Info`: the class that represents the information at each program point;
2. `bool Direction`: the direction of analysis. If it is true, then the analysis is a forward analysis; otherwise it is a backward analysis.
3. `Info InitialState`: the initial state of the analysis.
4. `Info Bottom`: the bottom of the lattice.

The first two parameters are the parameters of the template class `DataFlowAnalysis`. For example, to create subclass of `DataFlowAnalysis` for a forward analysis in which information is represented by class `MyInfo`, we use

```
class MyForwardAnalysis : public DataFlowAnalysis<MyInfo, true> { ... };
```

The last two parameters are the parameters to the constructor of `DataFlowAnalysis` and its subclasses. For example, in `class MyForwardAnalysis`, we need to implement a constructor declared as `MyForwardAnalysis(MyInfo & InitialStates, MyInfo & Bottom);`.

Directions

You need to implement `class DataFlowAnalysis` in `231DFA.h` so that it performs a forward analysis. We provide most of the code for forward analysis in `class DataFlowAnalysis`. You only need to implement the worklist algorithm in function `runWorklistAlgorithm`.

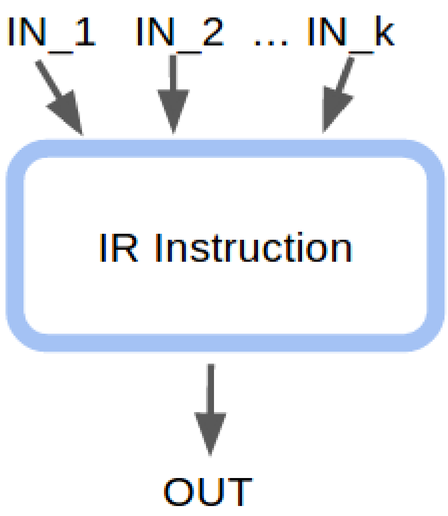
Reaching Definition Analysis

In part 2, you will also need to implement a **reaching definition analysis** based on the framework you implemented.

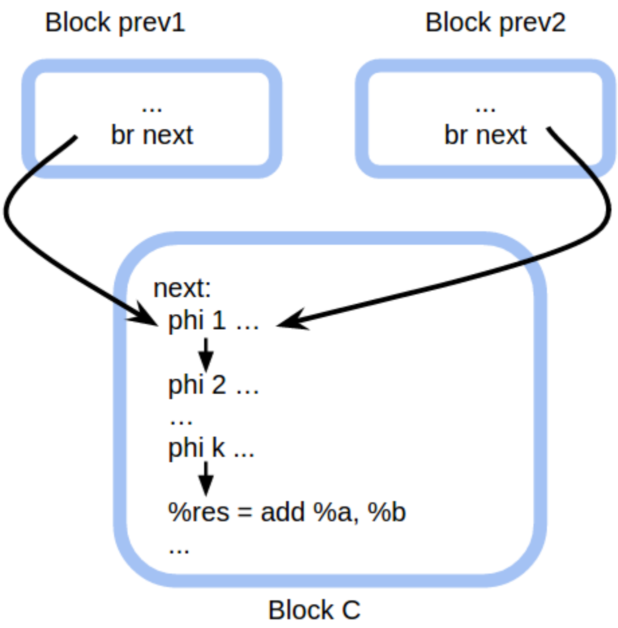
Control Flow Graph (CFG)

LLVM builds a CFG for a given function, and this CFG is available when your function pass is running. Let us call it **LLVM CFG**. The function `void initializeForwardMap(Function * func)` in `231DFA.h` also builds a CFG of the given function `func` for **forward analyses.** Let us call it **DFA CFG**. Analyses based on `231DFA.h` should work on **DFA CFGs.** DFA CFGs are slightly different from what we have seen during lectures.

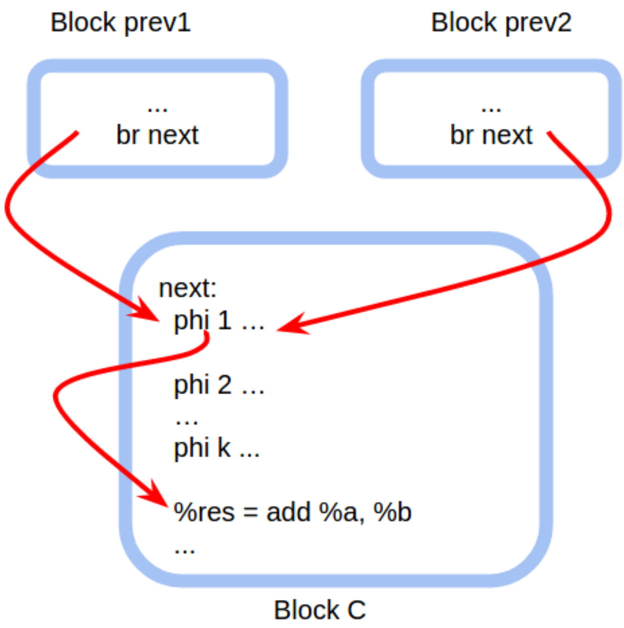
First, any instruction may have more than one incoming data flows, as is shown below



So the first step of any flow function should be **joining** the incoming data flows.
Second, in the lecture PHI nodes are used to synthesize values from different paths in an SSA representation. In LLVM IR, the phi instructions (https://llvm.org/docs/LangRef.html#phi-instruction) serve a similar functionality. However, because the `phi` instruction is an IR instruction and any IR instruction can return at most one value, if multiple values need to be synthesized there will be a number of `phi` instructions at the beginning of a basic block. I.e. in the LLVM CFG, an IR basic block may start with a number of consecutive `phi` instructions. In the DFA CFG, the first `phi` instruction connects to the first non-phi instruction in the same basic block directly, bypassing all the other `phi` instructions. Below is an example to illustrate these two kinds of CFG for the same code:



LLVM CFG



DFA CFG

There are k `phi` instructions at the start of the basic block C. In the LLVM CFG, these `phi` instructions are connected sequentially. In the DFA CFG, `phi 1` has an outgoing edge connecting to the first non-phi instruction `%res = add %a, %b` directly. When encountering a `phi` instruction, the flow function should process the series of `phi` instructions together (effectively a PHI node from the lecture) rather than process each `phi` instruction individually. This means that the flow function needs to look at the LLVM CFG to iterate through all the later `phi` instructions at the beginning of the same basic block until the first non-phi instruction.

Lattice

This analysis operates on LLVM IR, so we only care about the reaching definitions of LLVM IR variables. Note that LLVM IR is in Single Static Assignment (SSA) form, so every LLVM IR variable has exactly one definion. Also, because any LLVM IR instruction that can define a variable (i.e. has a non-void return type), can only define one variable at a time, there is a one-to-one mapping between LLVM IR variables and IR instructions that can define variables. For example, below is an `add` IR instruction:

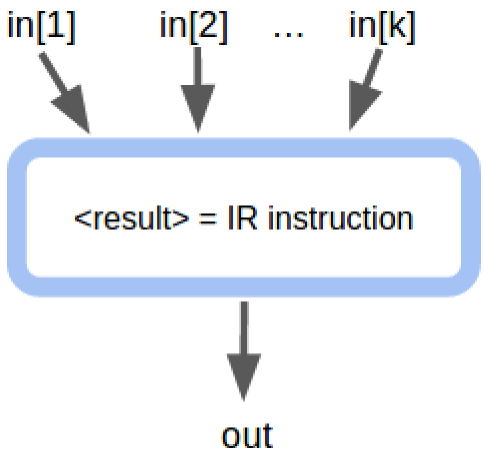
```
%result = add i32 4, %var
```

`%result` is the variable defined by this `add` instruction. None of any other instructions can redefine `%result`. This means that we can use the index of this `add` instruction to represent this definition of `%result`. Since a definition can be represented by the index of the defining instruction, unlike the reaching analysis taught in class, the domain ***H*** for this analysis is ***Ts{ i wi x,W-*** where ***W*** is a set of the indices of all the instructions in the function. The bottom is the empty set. The top is ***W***. ⊑ is ⊆ ("is subset of").

Flow Function

This analysis works at the LLVM IR level, so operations that the flow functions process are IR instructions. The flow functions are specified below. You need to implement them in `flowfunction` in your subclass of `DataFlowAnalysis`. There are three categories of IR instructions:

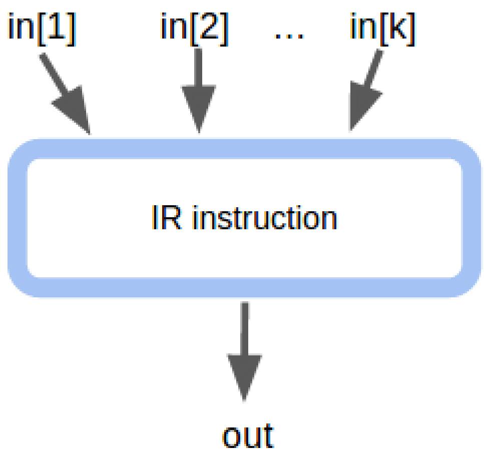
First Category: IR instructions that return a value (defines a variable)



$$out = in[1] \bigcup in[2] \ldots \bigcup in[k] \bigcup \{index\}$$

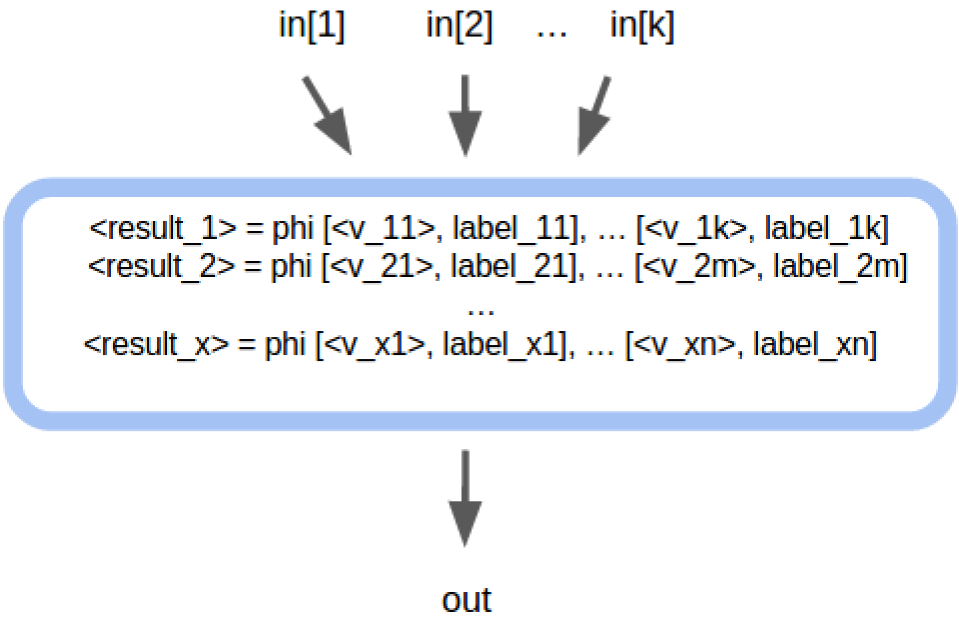
where index is the index of the IR instruction, which corresponds to the variable `<result>` being defined.

Second Category: IR instructions that do not return a value



$$out = in[1] \bigcup in[2] \ldots \bigcup in[k]$$

Third Category: phi instructions



$$out = in[1] \bigcup in[2] \ldots \bigcup in[k] \bigcup indices$$

where ***indices*** is the set of all the indices of the phi instructions.

For the reaching definition analysis, you only need to consider the following IR instructions:

1. All the instructions under binary operations (https://llvm.org/docs/LangRef.html#binary-operations);
2. All the instructions under binary bitwise operations (https://llvm.org/docs/LangRef.html#bitwise-binary-operations);
3. br (https://llvm.org/docs/LangRef.html#br-instruction);
4. switch (https://llvm.org/docs/LangRef.html#switch-instruction);
5. alloca (https://llvm.org/docs/LangRef.html#alloca-instruction);
6. load (https://llvm.org/docs/LangRef.html#load-instruction);
7. store (https://llvm.org/docs/LangRef.html#store-instruction);
8. getelementptr (https://llvm.org/docs/LangRef.html#getelementptr-instruction);
9. icmp (https://llvm.org/docs/LangRef.html#icmp-instruction);
10. fcmp (https://llvm.org/docs/LangRef.html#fcmp-instruction);
11. phi (https://llvm.org/docs/LangRef.html#phi-instruction);
12. select (https://llvm.org/docs/LangRef.html#select-instruction).

Every instruction above falls into one of the three categories. If an instruction has multiple outgoing edges, all edges have the same information. Any other IR instructions that are not mentioned above should be treated as IR instructions that do not return a value (the second categories above).

Directions

Info!
For the instructions below, you are allowed to use your own names for source files, since your CMakeLists.txt file (which you will also submit) takes care of the compilation process.

To implement the reaching definition analysis, you need to create

1. `class ReachingInfo` in `ReachingDefinitionAnalysis.cpp` : the class represents the information at each program point for your reaching definition analysis. It should be a subclass of `class Info` in `231DFA.h` ;
2. `class ReachingDefinitionAnalysis` in `ReachingDefinitionAnalysis.cpp` : this class performs reaching definition analysis. It should be a subclass of DataFlowAnalysis. Function `flowfunction` needs to be implemented in this subclass according to the specifications above;
3. A function pass called `ReachingDefinitionAnalysisPass` in `ReachingDefinitionAnalysis.cpp` : This pass should be registered by the name **cse231-reaching**. After processing a function, this pass should print out the reaching definition information at each progrm point to stderr. The output should be in the following form:

   ```
   Edge[space][src]->Edge[space][dst]:[def 1]|[def 2]| ... [def K]|\n
   ```

   where `[src]` is the index of the instruction that is the start of the edge, `[dst]` is the index of the instruction that is the end of the edge, `[def 1]` , `[def 2]` ... `[def K]` are indices of instructions (definitions) that reach at this program point. The order of the indices does not matter;
4. You may assume that only C functions will be used to test your implementation;
5. You may assume that the testing functions do not make any functions calls. However, LLVM does insert calls to intrinsics for analysis and optimization purposes. You should just treat them as **second category** instructions in your flow function, as `call (http://releases.llvm.org/9.0.0/docs/LangRef.html#call-instruction)` is not listed explicitly above;
6. You have to implement the ***exact*** flow functions as specified above. Given these flow functions, you have to get the ***most precise*** result for each edge.
7. Your CMakeLists.txt must create a module named `submission_pt2` to be compatible with our autograder.

Testing

To help you test your code, we provide our solution contained in the docker image. All solutions have been compiled in a module named "231_solution.so". Our reaching definition analysis solution is registed with the same name (cse231-reaching). For example, to run the reaching definition solution, type

```
opt -load 231_solution.so -cse231-reaching < input.ll > /dev/null
```

Turnin Instructions

You will turn in your submission in Gradescope. As soon as you submit, your code will be auto-graded and you should have your grade and some feedback within a few minutes. You are allowed to submit as many times as you want until the deadline.

Grading

Your submission will be graded against 4 benchmarks we developed which satisfy all the requirements (only one function, written in C, no function calls, etc). Unlike part 1, partially correct submissions will receive partial credit (grading strategy presented below). The order of you output does not matter - This includes the line order, as well as the order within a line (the order with which reaching definitions are presented). But make sure you follow the output form to the letter (no extra spaces/tabs, printed in stderr, no extra output). The autograder attempts to clean the extra output (if any), but do not rely on that.

Grading Strategy

1. Same as part 1, your submission is compiled and if anything goes wrong with this step you get a score of 0 and an error message.
2. Each reaching definition printed from our solution module is worth 1 point. You will get 1 point for each match from your submission. Notice that some benchmarks will have more definitions than others, thus their maximum score will be higher. At the end of grading, the autograder will normalize all benchmarks to 100 points.
3. Your submission might print extra edges (edges that are not part of our solution output). Each extra edge will cost you 5 points.
4. Your submission might have missing edges (edges that are part of our solution output but not part of your submission's output). Since missing an edge in a reaching definitions pass can have more serious consequences than an extra edge, we penalize each missing edge with 10 points.

Submission Directions

- Same as with part 1, you have to submit all source files necessary to compile your passes.
- Since you are providing a CMakeLists.txt file, your source code files can have any name you want. But the LLVM module must be named `submission_pt2` and the pass needs to be named `cse231-reaching`

Tutorial

You can find a an overview on the dataflow analysis project here (tutorials/Dataflow-Analysis-Project.pdf).