Part 3

Due Date Feb 21 11:59:59 PM

In this part you need to reuse and extend the dataflow analysis framework to implement a liveness analysis and a may-point-to analysis.

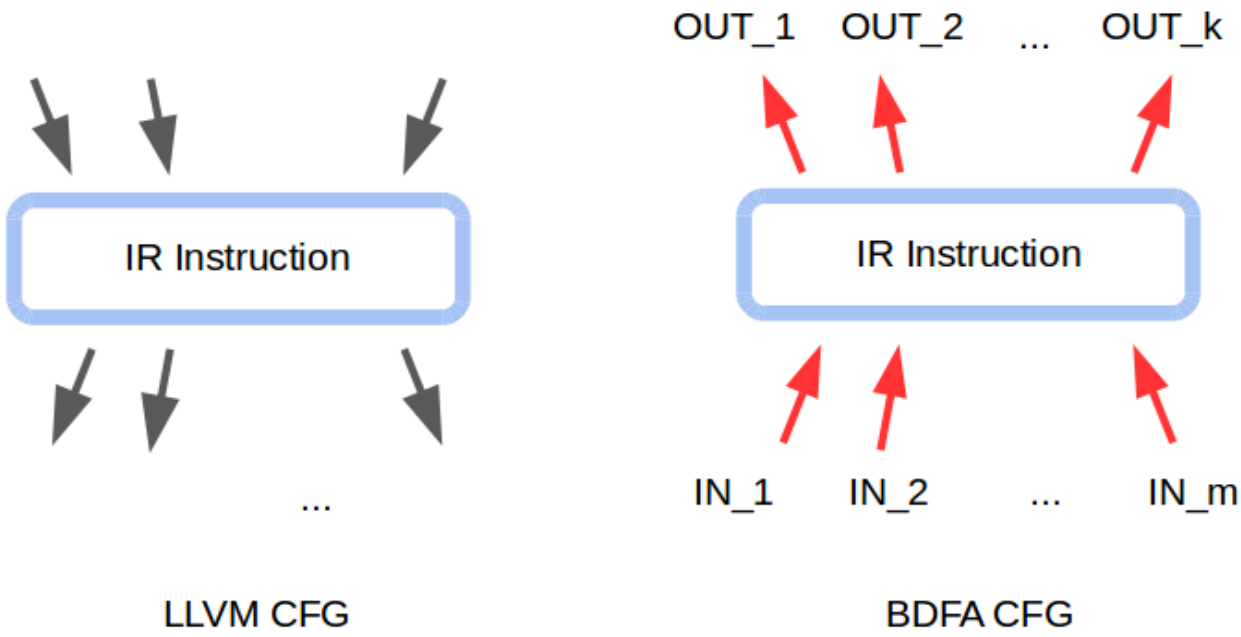
Liveness Analysis

Your first task for this part is to implement a **liveness analysis** based on the framework implemented in the previous assignment. Recall that a variable is ***live*** at a particular point in the program if its value at that point will be used in the future. It is ***dead*** otherwise.

Control Flow Graph (CFG)

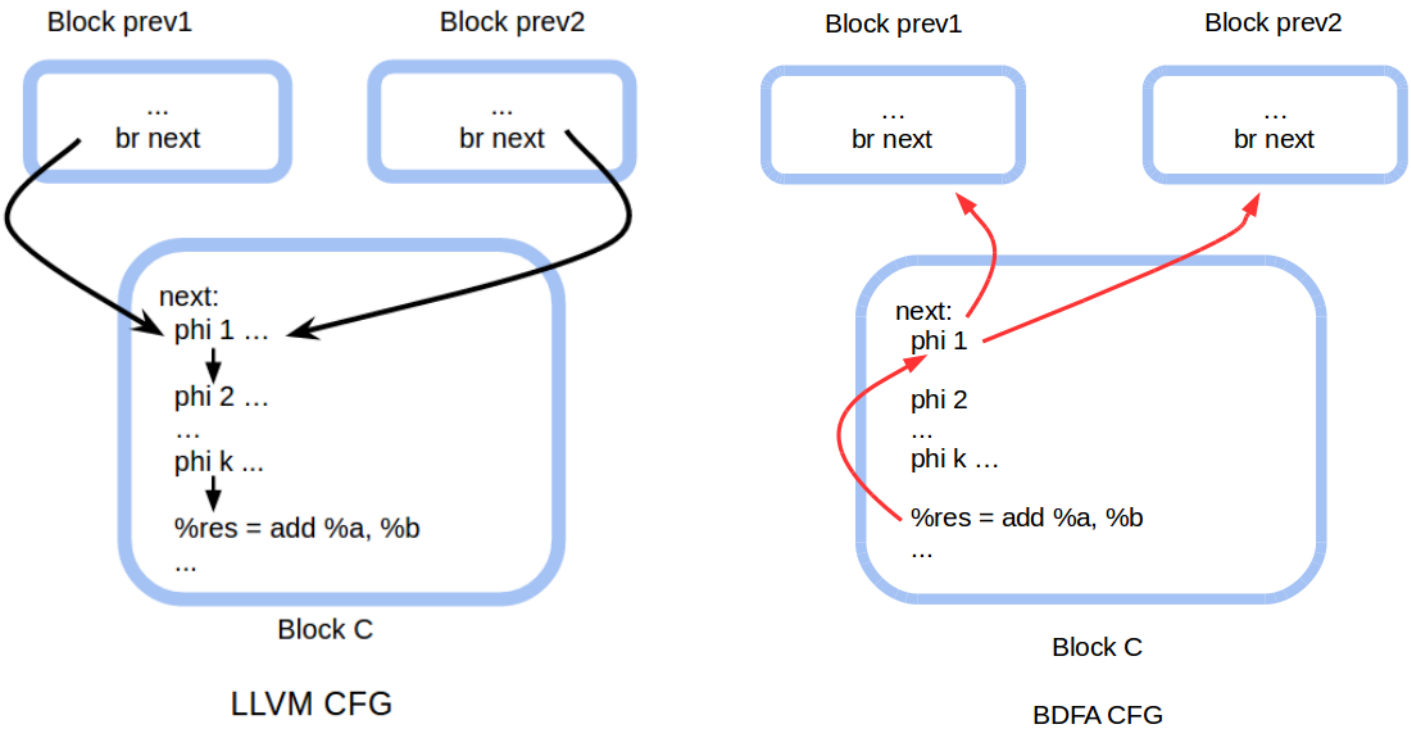
Like in part 2, LLVM builds a CFG for a given function, which we call **LLVM CFG**. Unlike reaching definitions, liveness is a backwards analysis. This means that we need to propagate facts backwards from OUT to IN. The function `void initializeBackwardMap(Function * func)` in `231DFA.h` builds a CFG of the given function `func` for **backwards analyses**. Let us call it **BDFA CFG**. Analyses based on `231DFA.h` should work on **DFA CFGs** or **BDFA CFGs**. The choice between the two is determined by the `Direction` parameter of the `DataFlowAnalysis` class. Below are some of the main points of constructing BDFA CFGs:

First, the edges of a BDFA CFG are reversed with respect to the original LLVM CFG (or the respective DFA CFG for that matter).



Also, instructions may have multiple incoming and outgoing edges, so the first step of any flow function should be **joining** the incoming data flows.

Second, phi instructions (http://releases.llvm.org/9.0.0/docs/LangRef.html#phi-instruction) are handled similarly to part 2: 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 BDFA CFG, the first non-phi instruction connects to the first `phi` in the same basic block directly, bypassing all the other `phi` instructions. Below is an example that illustrates these two kinds of CFG for the same code:



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 BDFA CFG, the first non-phi instruction `%res = add %a, %b` has an outgoing edge connecting to `phi 1` 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

Just like part 2, this analysis operates on the LLVM IR, so we only care about LLVM IR variables that are live at each program point. Note that LLVM IR is in Single Static Assignment (SSA) form, so every LLVM IR variable has exactly one definition. LLVM IR instructions that define variables (i.e. have a non-void return type), can only define one variable at a time. Therefore, 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:

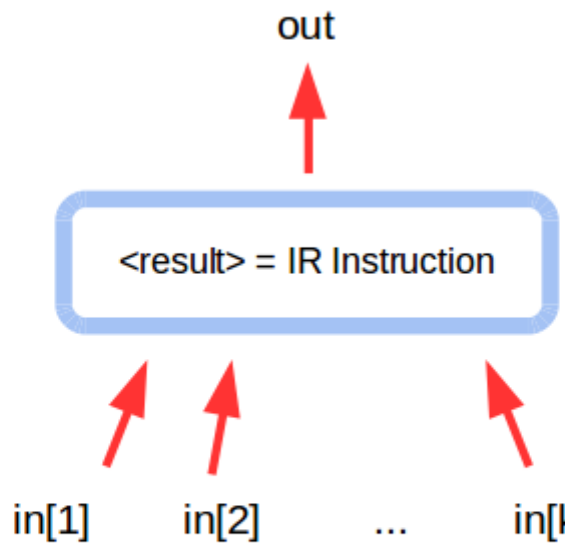
```
%result = add i32 4, %var
```

`%result` is the variable defined by this `add` instruction. No other instructions can redefine `%result`. This means that we can use the index of this `add` instruction to the variable `%result`. At each program point we are interested for the $w_i x$ of live variables. So the domain H for this analysis is $Ts\{ i w_i x, W$ 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. $\sqsubseteq$ is $\subseteq$ ("is subset of").

Flow Functions

You are asked to implement flow functions that process 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:

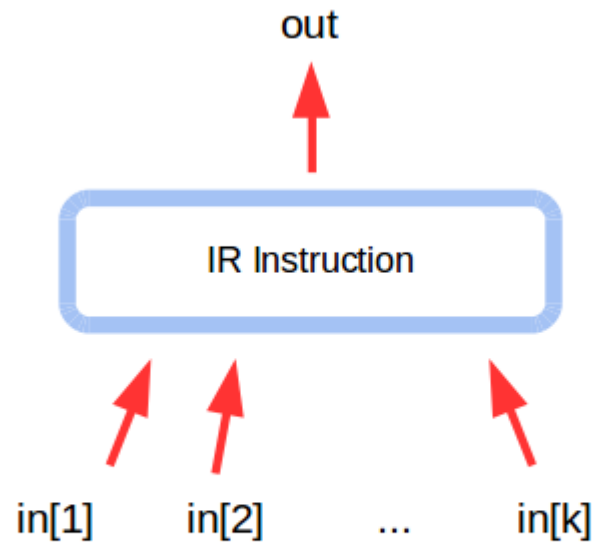
First Category: IR instructions that return a value (define a variable)



$$out = in[1] \cup in[2] \cup \ldots \cup in[k] \cup operands - \{ index \}$$

where $st\,i\,ver\,hw$ is the set of variables used (and therefore needed to be live) in the body of the instruction, and mhi | is the index of the IR instruction, which corresponds to the variable `<result>` being defined. For example, in the instruction `%result = add i32 4, %var` , $st\,i\,ver\,hw$ would be the singleton set { $I_{\%var}$ }, where $I_{\%var}$ is the index of the instruction that defines `%var` .

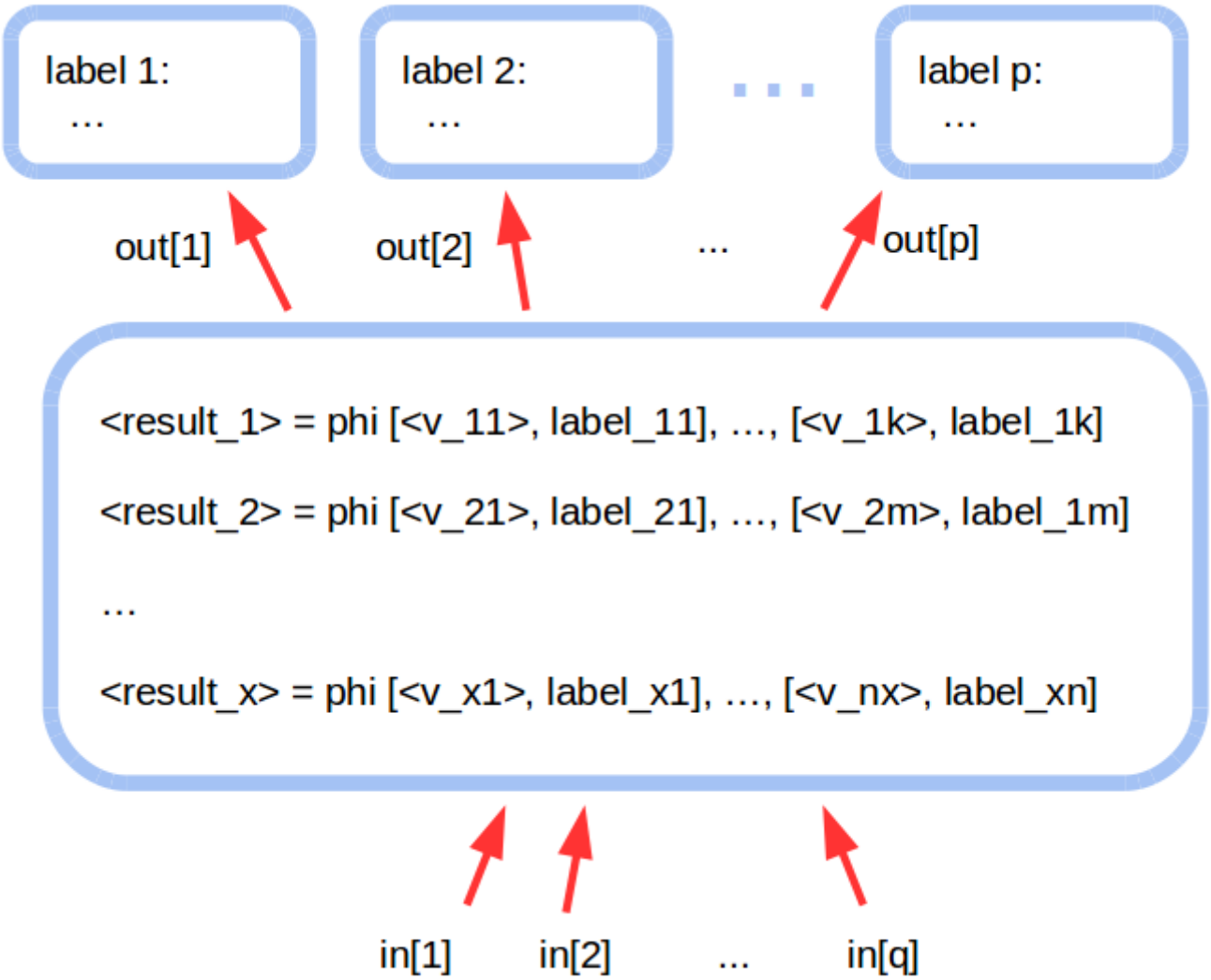
Second Category: IR instructions that do not return a value



$$out = in[1] \cup in[2] \cup \ldots \cup in[k] \cup operands$$

Third Category: phi instructions

We will handle the phi instructions in a slightly specialized way to boost the precision of our analysis.



But first lets take a look at the problem that we would be facing, if we followed a naive approach. Suppose we compute a single out set, common to all outgoing edges. Following the intuition of the previous cases, this set will contain all values from <v_11> to <v_xn> that correspond to variables. This set is unnecessarily large: it includes variables that might never meet their corresponding definition following the CFG all the way to the beginning of the function. For example, the instruction corresponding to <v_11> will not be found in any basic block other than the one under label_11.

The root of the problem is that the above approach ignores an important piece of information that the phi instruction provides, namely the label of the basic block that each value originates from. The pair [<v_11>, label_11], for example, guarantees that <v_11> comes from the block labeled with label_11. Therefore, it is pointless to propagate the fact for <v_11> to any other basic block, and hence safe to avoid doing so. This intuition leads to the following flow function:

$$\text{out}[k] = (\text{in}[1] \cup \text{in}[2] \cup \ldots \cup - \{ \text{result\_i} \mid i \text{ in } [1..x] \}) \cup \{ \text{ValuetoInstr}(v\_ij) \mid \text{label } k == \text{label\_ij} \}$$

Here we include all incoming facts like before and exclude variables defined at each phi instruction. In addition, each outgoing set out[k] contains those phi variables v_ij that are defined in its matching basic block (labeled with label_k). The function ValueToInstr is merely used to extract the defining instruction from each phi value.

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

1. All the instructions under binary operations (http://releases.llvm.org/9.0.0/docs/LangRef.html#binary-operations);
2. All the instructions under binary bitwise operations (http://releases.llvm.org/9.0.0/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. With the exception of Phi instructions discussed above, 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 category above).

Directions

To implement the liveness analysis, you need to

1. implement the `initializeBackwardMap` method in `231DFA.h` that creates the BDFA CFG;
2. create a `class LivenessInfo` in `LivenessAnalysis.cpp` : the class represents the information at each program point for your liveness analysis. It should be a subclass of `class Info` in `231DFA.h` ;
3. create a `class LivenessAnalysis` in `LivenessAnalysis.cpp` : this class performs the liveness analysis. It should be a subclass of `DataFlowAnalysis` . Function `flowfunction` needs to be implemented in this subclass according to the specifications above;
4. write a function pass called `LivenessAnalysisPass` in `LivenessAnalysis.cpp` : This pass should be registered by the name **cse231-liveness**. After processing a function, this pass should print out the liveness information at each program 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;
5. You may assume that only C functions will be used to test your implementation;
6. 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;
7. 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.

May-point-to Analysis

In part 3, you will also need to implement a **may-point-to analysis** based on the framework you implemented.

Lattice

For this analysis, only memory objects allocated locally by LLVM IR instruction alloca (http://releases.llvm.org/9.0.0/docs/LangRef.html#alloca-instruction) can be pointees. A pointer can be either an LLVM IR variable of some pointer type (IR pointer), or a memory object (memory pointer). The analysis considers a pointer points to a memory object as long as the pointer points to any part of the object. Also, a memory object may contain multiple pointers (For example, a C struct that constains `int * p1` and `char * p2`). The analysis considers such a memory object as one memory pointer and it may point to the union of the the memory objects that the pointers inside of it may point to. **DFA identifiers** are used to reference IR pointers and memory objects: The DFA identifier of a IR pointer is `R`i where i is the index of the defining IR instruction of this IR pointer (IR variable). The DFA identifier of a memory object is `M`i where i is the index of the IR instruction that allocates this memory object. For example, suppose that the following IR instruction's index is 10

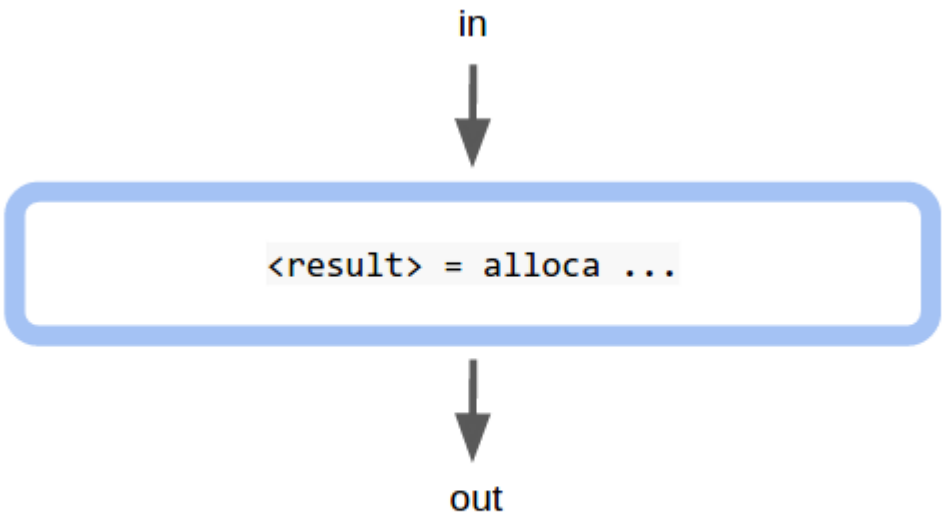
```
%ptr = alloca i32, i32 4
```

then the DFA identifier of the IR pointer, `%ptr`, created by this instruction is **R10**, and the DFA identifier of the memory object allocated by this instruction is **M10**.
Let ***Pointers*** be the set of the DFA identifiers of the pointers in the function (including IR pointers and memory pointers) and ***MemoryObjects*** the set of the DFA identifiers of the memory objects allocated in the function. The domain ***H*** for this analysis is $Ts\{ iwix,W$, where $WAt \rightarrow s\ ¡\ t\ \in Tsmnxiww**s\ \in Qiqsv\}Sfnigxwc$. The bottom is the empty set. The top is W. $\sqsubseteq$ is $\subseteq$ ("is subset of").

Flow Functions

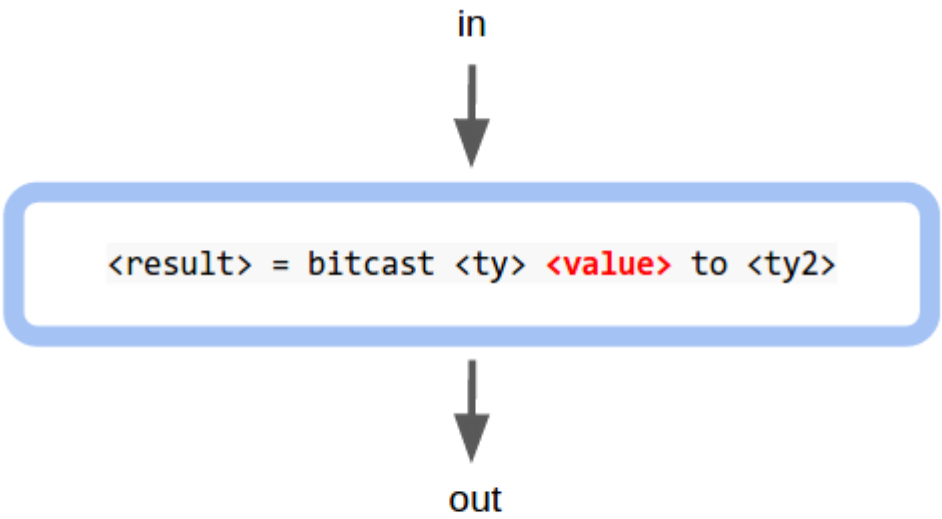
Like the reaching definition analysis from part 2, this analysis works at the LLVM IR level, so operations that the flow functions process are IR instructions. Again, each instruction may have multiple incoming edges, so the first step is to merge all incoming edges with the join operation. For simplicity, the flow functions below assume that the incoming edges have been merged. Also, many LLVM IR instructions have a variation that takes vector type parameter. That variation should be ignored in this analysis. The flow functions are specified below. Assume that the index of the IR instruction in question is ***i***.

1. **alloca (http://releases.llvm.org/9.0.0/docs/LangRef.html#alloca-instruction)**



$$out = in \bigcup \{Ri \rightarrow Mi\}$$

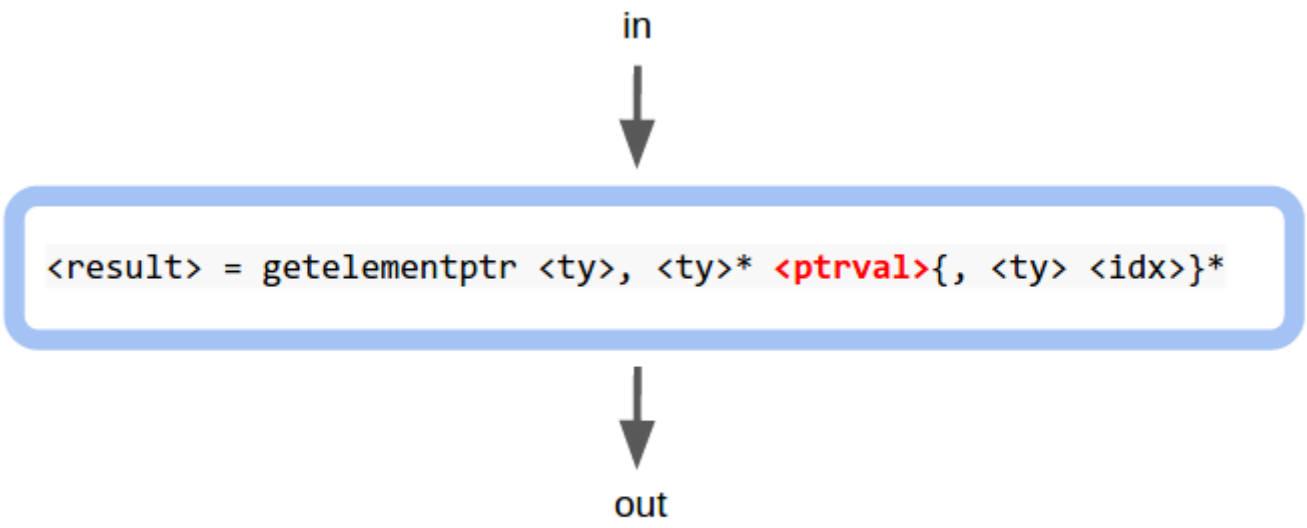
2. **bitcast .. to (http://releases.llvm.org/9.0.0/docs/LangRef.html#bitcast-to-instruction)**



$$out = in \bigcup \{Ri \rightarrow X \mid Rv \rightarrow X \in in\}$$

where ***Rv*** is the DFA identifier of **<value>**.

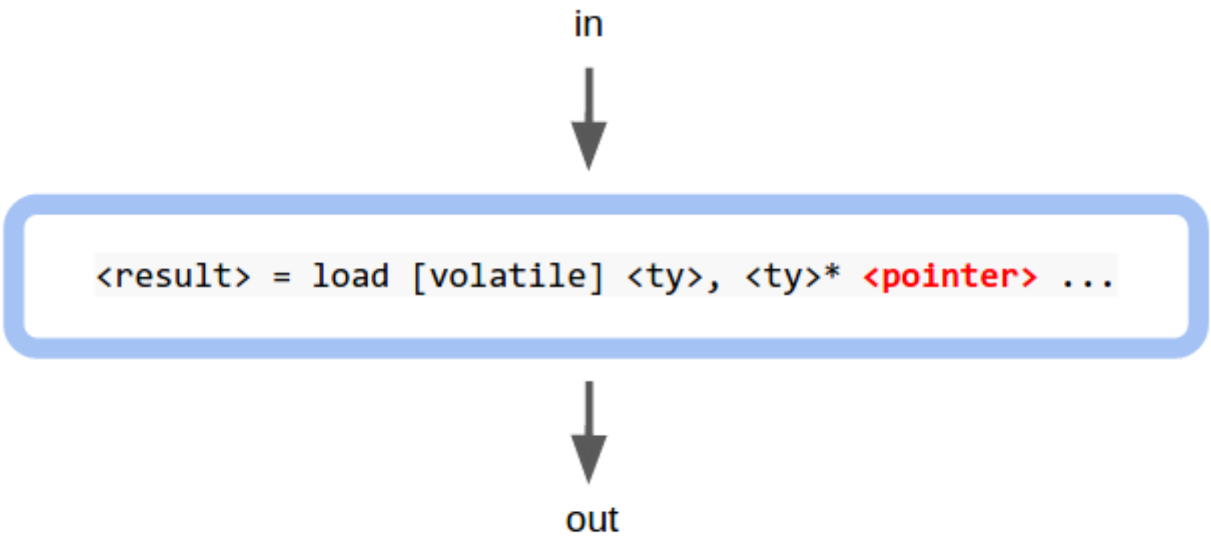
3. **getelementptr (http://releases.llvm.org/9.0.0/docs/LangRef.html#getelementptr-instruction)**



$$out = in \bigcup \{Ri \to X \mid Rv \to X \in in\}$$

where ***Rv*** is the DFA identifier of **<ptrval>**.

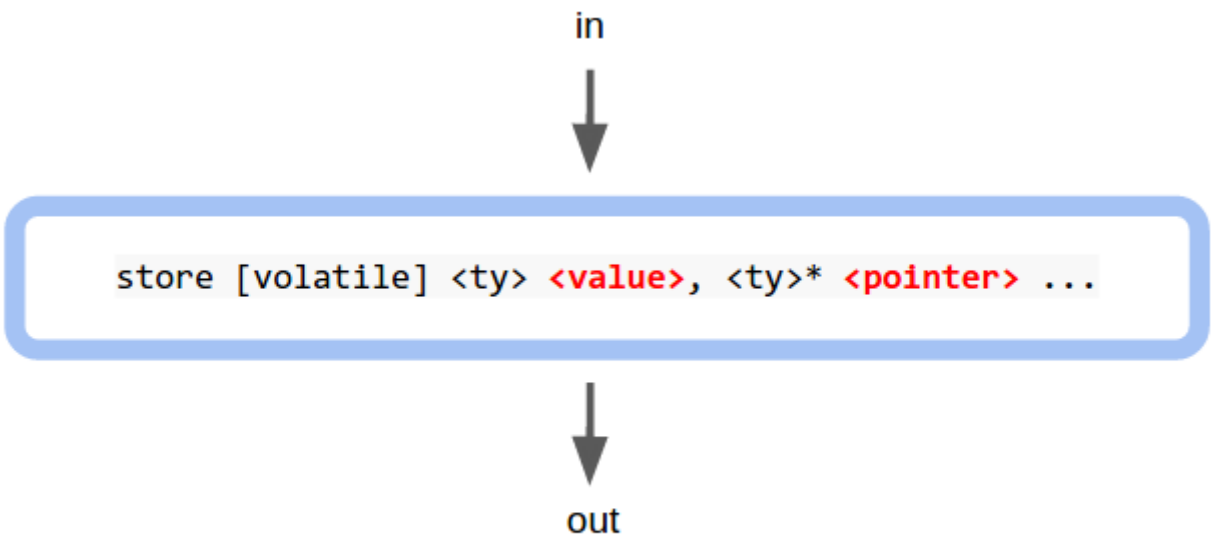
4. **load (http://releases.llvm.org/9.0.0/docs/LangRef.html#load-instruction)**



$$out = in \bigcup \{Ri \to Y \mid Rp \to X \in in \cap X \to Y \in in\}$$

where ***Rp*** is the DFA identifier of **<pointer>**.

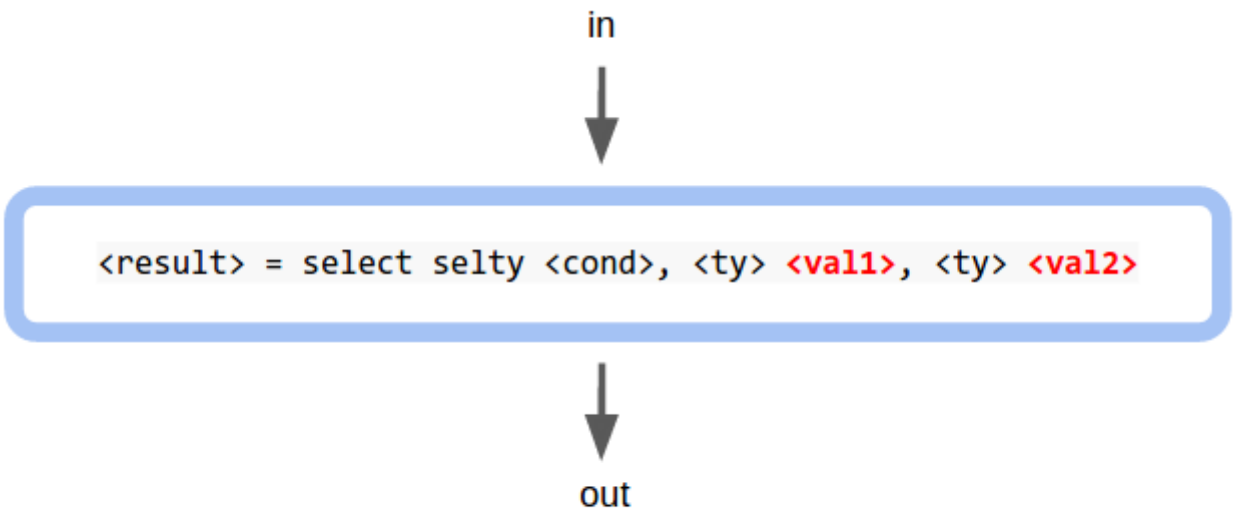
5. **store (http://releases.llvm.org/9.0.0/docs/LangRef.html#store-instruction)**



$$out = in \bigcup \{Y \rightarrow X \mid Rv \rightarrow X \in in \cap Rp \rightarrow Y \in in\}$$

where ***Rv*** and ***Rp*** are the DFA identifiers of **<value>** and **<pointer>**, respectively.

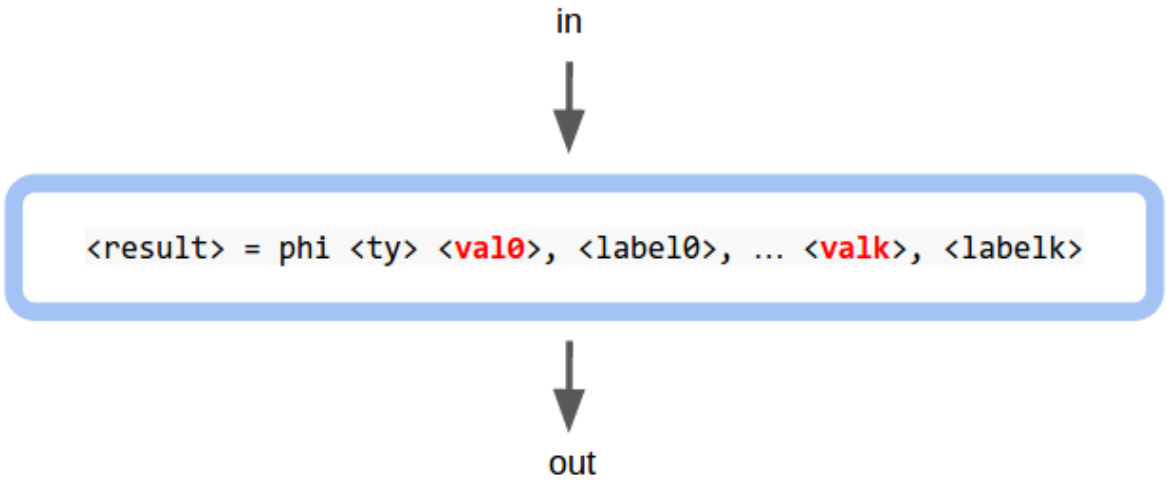
6. **select (http://releases.llvm.org/9.0.0/docs/LangRef.html#select-instruction)**



$$out = in \bigcup \{Ri \rightarrow X \mid R1 \rightarrow X \in in\} \bigcup \{Ri \rightarrow X \mid R2 \rightarrow X \in in\}$$

where ***R1*** and ***R2*** are the DFA identifiers of **<val1>** and **<val2>**, respectively.

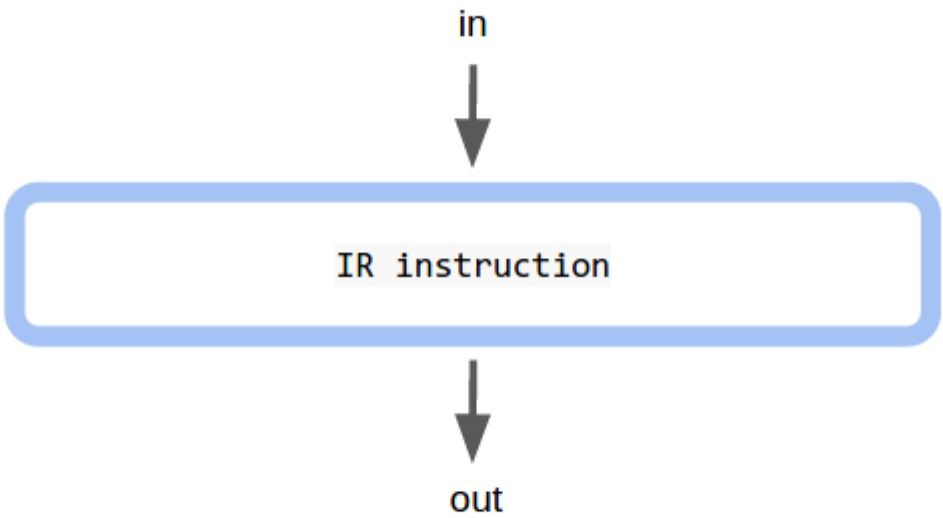
7. **phi (http://releases.llvm.org/9.0.0/docs/LangRef.html#phi-instruction)**



$$out = in \bigcup \{Ri \to X \mid R0 \to X \in in\} \bigcup ... \bigcup \{Ri \to X \mid Rk \to X \in in\}$$

where ***R0*** through ***Rk*** are the DFA identifiers of **<val0>** through **<valk>**, respectively. This is the flow function for one `phi` instruction. You need to make sure that all the `phi` instructions at the beginning of a basic block are processed properly.

8. **Any other instructions**



$$out = in$$

Directions

To implement the may-point-to analysis, you need to create

1. `class MayPointToInfo` in `MayPointToAnalysis.cpp` : the class represents the information at each program point for your may-point-to analysis. It should be a subclass of `class Info` in `231DFA.h` ;
2. `class MayPointToAnalysis` in `MayPointToAnalysis.cpp` : this class performs a points-to 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 `MayPointToAnalysisPass` in `MayPointToAnalysis.cpp` : This pass should be registered by the name **cse231-maypointto**. After processing a function, this pass should print out the points-to information at each program point to `stderr` . The output should be in the following form:

   ```
   Edge[space][src]->Edge[space][dst]:[point-to 1]|[point-to 2]| ... [point-to 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. `[point-to i]` represents what the ***i*th** pointer may point to at this moment, and it should be in the following form:

   ```
   [pointer's DFA ID]->([pointee 1's DFA ID][slash][pointee 2's DFA ID][slash] ... [poi
   ntee m's DFA ID][slash])
   ```

   The orders of the pointers and pointees do not matter;
4. You may assume that only C functions will be used to test your implementation.
5. You may assume that **inttoptr (http://releases.llvm.org/9.0.0/docs/LangRef.html#inttoptr-to-instruction)** never appears in any test cases.
6. You may assume that the testing functions take no parameters and there are no global/static variables. That is, the initial state of the analysis is always bottom;
7. 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.

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-liveness and cse231-maypointto). For example, to run the reaching definition solution, type

```
opt -load 231_solution.so -cse231-liveness < 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.

Submission Directions

- Same as with parts 1 and 2, 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_pt3` and the passes need to be named `cse231-liveness` and `cse231-maypointto`