Lab 2
Iterative Dataflow Analysis Framework

Part (a): In class, we discussed many interesting data flow analyses such as Reaching Definitions, Liveness analysis, etc. Although these analyses are different in certain ways, for example, they compute different program properties and analyze the program in different directions (forward, backward), they share some common properties such as iterative algorithms, transfer functions, and meet operators. These commonalities make it worthwhile to write a generic framework that can be parameterized appropriately for solving a specific data flow analysis. Further, a well written iterative data flow analysis framework significantly reduces the burden of implementing new data flow passes, the developer only writes pass specific details such as the meet operator, transfer function, analysis direction etc. In particular, the framework should solve any unidirectional data flow analysis as long the analysis supplies the following: (1) Domain (including the Semi-Lattice) (2) Direction (Forwards/Backwards) (3) Transfer Function (4) Meet Operation (5) Boundary Condition (6) Initial Interior Points (Top).
Careful thought should be given to how the analysis parameters are represented. For example, the direction could reasonably be represented as a boolean, while function pointers may seem more appropriate for representing transfer functions.

In this assignment, you will implement such an iterative data flow analysis framework for LLVM IR

Part (b): You will now use your iterative data flow analysis framework to implement a forward data flow analysis (Reaching Definitions) and a backward data flow analysis (Liveness) in LLVM IR. As explained below in more details, each analysis should perform computation at program points.

Liveness: On convergence, your Liveness pass should report all variables that are "live" at each program point.

Reaching Definitions: On convergence, your Reaching Definitions pass should report all the definitions that "reach" each program point.

```
int sum (int a, int b)
{
  int i;
  int res = 1;
  for (i = a; i < b; i++)
  {
   res *= i;
  }
  return res;
}
```

(a)

```
define i32 @sum(i32 %a, i32 %b) {
        entry:
                %0 = icmp slt i32 %a, %b
                br i1 %0,  %bb1, %bb3
        bb1:
                %tmp = sub i32 %b, %a
                br  %bb2
        bb2:
                %1 = phi i32 [ 0, %bb1 ], [ %5, %bb2 ]
                %2 = phi i32 [ 1, %bb1 ], [ %4, %bb2 ]
                %3 = add i32 %1, %a
                %4 = mul nsw i32 %2, %3
                %5 = add i32 %1, 1
                %6 = icmp eq i32 %5, %tmp
                br i1 %6,  %bb3, %bb2
        bb3:
                %res = phi i32 [ 1, %entry ], [ %4, %bb2 ]
                ret i32 %res
}
```

(b)

Figure 1: (a) Simple loop code, and (b) corresponding optimized LLVM bytecode

Implementation Issues

The Single Static Assignment (SSA) form of LLVM intermediate representation presents some unique challenges when performing iterative data flow analysis.:

- Values in LLVM are represented by the Value class. In SSA every value is guaranteed to have only a single definition point, so instead of representing values as some distinct variable or pseudo register class, LLVM represents values defined by instructions by the defining instruction. That is, Instruction is a subclass of Value. There are other subclasses of Value, such as basic blocks, constants, and function arguments. For this assignment, we will only track the liveness of instruction-defined values and function arguments.

- ϕ instructions are pseudo instructions that are used in the SSA representation and need to be handled specially by both Liveness and Reaching Definitions. Since SSA requires that values have a unique definition at any program point (P), it is natural to wonder how a value that is live at P, but has different definitions on the paths leading to it is handled. The SSA solution is to introduce ϕ instructions at the beginning of the basic block containing P, to "combine" all the different definitions, so that all the uses in the block (including at P), see only the definition by the phi instruction. Consider the uses of ϕ (phi) instructions in Figure 1(b) as illustrations. You should carefully consider how your analysis passes are affected by ϕ instructions. The fact that you are working on code in SSA form will have ramifications on how your passes are implemented. For example, the way ϕ instructions are handled will determine the precision of your analysis. Think carefully about what this means to your implementation and briefly explain this in your assignment report.
- Your passes should not output results for the program point preceding a phi instruction since they are pseudo instructions which will not appear in the executable. To guide you in formatting the output of your passes, the expected output of running Liveness analysis on the bytecode from Figure 1(b) is shown in Figure 2 (The live values are shown as underlined at the left-hand side).

```
                    define i32 @sum(i32 %a, i32 %b) {
                      entry:
{%a, %b}
                            %0 = icmp slt i32 %a, %b
{%a, %b, %0}
                            br i1 %0,  %bb1, %bb3
                      bb1:
{%a, %b}
                            %tmp = sub i32 %b, %a
{%a, %temp}
                            br  %bb2
```

```
                bb2:
                        %1 = phi i32 [ 0, %bb1 ], [ %5, %bb2 ]
                        %2 = phi i32 [ 1, %bb1 ], [ %4, %bb2 ]
{%a,%tmp,%1,%2}
                        %3 = add i32 %1, %a
{%a,%tmp,%1,%2,%3}
                        %4 = mul nsw i32 %2, %3
{%a,%tmp,%1,%4}
                        %5 = add i32 %1, 1
{%a,%tmp,%4,%5}
                        %6 = icmp eq i32 %5, %tmp
{%a,%tmp,%4,%5,%6}
                        br i1 %6,  %bb3, %bb2
                bb3:
                        %res = phi i32 [ 1, %entry ], [ %4, %bb2 ]
{%res}
                ret i32 %res
        }
```

Figure 2

Part (c): May-point-to Analysis

In this part, you will need to implement a may-point-to analysis based on the framework you implemented. In other words, it computes sets of variable that each pointer may point to. The assumptions for the analysis are:

- The variables allocated locally by LLVM IR instruction alloca and the global variables can be pointees.
- A pointer can be an IR variable of some pointer type.
- Should be field-insensitive i.e. if a pointer points to any field of an aggregate data structure variable, it is considered to point to the whole variable.

Lattice

- Let Pointers (p) be the set of the pointers in the function and variables(v) be the set of variables(allocated in the function and global symbols) accessed by the function.
- The domain D for this analysis is Powerset(S), where S={p → v | p ∈ Pointers && v ∈ variables}.
- The bottom is the empty set.
- The top is S.
- Direction of analysis is forward.
- ⊑ is ⊆ ("is subset of").
- On convergence, the pass should report the may points to information at every point of the program.

Transfer function

The analysis works at the LLVM IR level, so operations that the transfer functions process are IR instructions. You need to define transfer function specifically for alloca, load, store, getelementpointer, select, store, bitcast instructions.

- *Arguments*
  Pointer arguments can point to any global variable (but not local variables generated by alloca)

- *'alloca' instruction:*
  The '`alloca`' instruction allocates memory on the stack frame of the currently executing function, to be automatically released when this function returns to its caller. The return value is a pointer to the allocated variable. The '`alloca`' instruction is commonly used to represent automatic variables that must have an address available.

```
%ptr = alloca i32                       ; yields i32*:ptr
```

%ptr -> %ptr

- *'bitcast .. to' instruction:*

  The '`bitcast`' instruction takes a value to cast and a type to cast it to. If the source type is a pointer, the destination type must also be a pointer of the same size.

```
%Y = bitcast i32* %x to i64*          ; yields i64*:%x
```

If %x -> setV, then %Y -> setV

- *'getelementptr' instruction*

  The '`getelementptr`' instruction is used to get the address of a subelement of an aggregate data structure. It performs address calculation only and does not access memory. The first argument is always a type used as the basis for the calculations. The second argument is always a pointer and is the base address to start from. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed. The return value is a pointer to an inner field of the aggregate variable.

  ```
  %4 = getelementptr inbounds i32* %arr, i32 %i.0
  ```

  %4 -> %arr

- *'load' instruction*

  The '`load`' instruction is used to read from memory. The argument to the `load` instruction specifies the memory address from which to load. The interesting case is when the the value loaded (result) is a pointer itself.

  ```
  <result> = load [volatile] <ty>, <ty>* <pointer>
  ```

  If (pointer -> X, X-> setV) then (result -> setV)

- *'store' instruction*

  The '`store`' instruction is used to write to memory. There are two arguments to the `store` instruction: a value to store and an address at which to store it. The interesting case is when the the value stored is a pointer itself.

  ```
  store [volatile] <ty> <value>, <ty>* <pointer>
  ```

  If (value -> setV, pointer-> Y) then (Y -> setV)

- *'select' instruction*

  The '`select`' instruction is used to choose one value based on a condition, without IR-level branching. The interesting case is when the the values chosen are pointers.

  ```
  <result> = select selty <cond>, <ty> <val1>, <ty> <val2>
  ```

  If (val1 -> X, val2 -> Y) then (result -> X U Y)

- *'phi' instruction*

  The '`phi`' instruction takes a list of pairs of (values, predecessor basic block) as arguments, with each pair for each predecessor basic block of the current block. It logically takes on the value specified by the pair corresponding to the predecessor basic block that executed just prior to the current block. Similar to select, the interesting case is when the the values chosen are pointers.

  ```
  <result> = phi <ty> [ <val0>, <label0>], …
  ```

  If (val0 -> X, val1 -> Y … .) then (result -> X U Y . . .)

- File reference_code contains an example C code and its corresponding LLVM bit code. The expected output of running may points to analysis on the bytecode is also shown in the file on the left hand side.

Assumptions (for all above parts):

- The implemented analyses should be intra-procedural, thus only analyzing the bodies of functions.
- Your implementation will be tested on simple C functions having pointers.
- 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. Ignore such "call instructions" during analysis

Submission instructions:

You need to submit:

- source code for your framework
- source code for liveness pass
- source code for Reaching definitions pass
- source code for may-points to analysis pass
- the associated Makefiles
- 5 example C programs (having pointers, using global variables, taking arguments) on which you tested your passes (Test cases)
- Readme describing instructions to build and run your code
- Also, remember to do a good job of commenting your submitted code.