CSE 231- Project 4

Rajat Grewal

Dataflow Analysis

1. MOD Analysis + MPT

2. ConstantPropAnalysis

Which pass to use?

- We need to implement an inter-procedural analysis and therefore can't use a normal Function pass
- A module pass might work but then it becomes difficult to track the caller-callee relations within the pass
- LLVM Description for ***CallGraphSCCPass*** - `These passes are inherently interprocedural...`
- A good resource to get more info about CallGraphSCCPass - Link
- **CallGraphSCCPass does a bottom-up traversal on the CallGraph, i.e., it starts from the leaf nodes.**
- **The callee info would have been calculated before we go to the caller**

MPT

- MPT stands for May-point-to
- A global set of variables that may be pointed to by some other variable
- Don’t need to worry about the variable doing the actual pointing. This is how this MPT differs from you May-Point-To Analysis in Part 3 of the project
- Don’t discriminate between global or local variables while creating the MPT set
- MPT can be constructed within the doInitialization method of your Pass. All it takes is one pass through the CallGraph

Data Structure

```cpp
std::set<Value*> MPT;
```

Value type can be converted to Instruction, GlobalVariable, Constant, Operator….. (dyn_cast)

Take a look at the class diagram for Value here

When to add elements to MPT set?

1. **global1 = &global2** MPT -> {global2}
   Hint - Get a list of all global variables in the CallGraph. How do you get the value pointed by global1?

```
GlobalVariable* pointed = dyn_cast<GlobalVariable>(glob.getInitializer())
```

2. X = **&Y** MPT ->{Y} //X and Y don't necessarily need to be global
   Hint - Such instruction translate to store in the .ll file.

3. **function(...&operand(s)...) and return &operand** MPT ->{operand(s)}
   Hint - Such instructions translate to Call and Ret respectively in the .ll file.
   Use& type

MOD analysis

- In simpler terms - Store the variables that might be modified in a function
- We just focus on Global variables here as suggested by the Data Structure(unlike MPT)
- MPT was a global set for all functions. This will be on a per functions basis
- We will calculate this in the doInitialization method and the runOnSCC method of the pass
- Ideally, we would have required a worklist algorithm to reach a fixed point but runOnSCC helps us avoid that

Data Structure

```
std::unordered_map<Function*, std::set<GlobalVariable*>> MOD;
```

Note that the Set only takes GlobalVariable*

No need to use the DataFlowAnalysis written in 231DFA.h for MOD

When to populate MOD?

This should happen on a per function basis within doInitialization. Suppose for any given function F, we encounter the following

1. global_var = ___   MOD[&F] ->{global_var}   //This global variable might be modified
   Hint - check if the PointerOperand of a store instruction is global or not?

2. *var = ____   MOD[&F] ->{global_subset(MPT)}   // var can modify anything inside MPT
   Hint - Such instructions translate to a combination of store and load

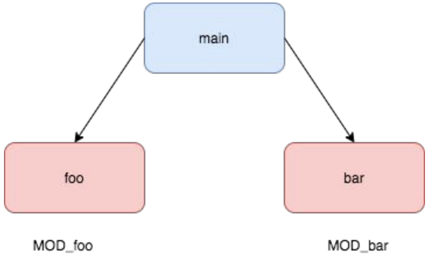
   This is not at all precise but serves our purpose for this assignment.
   PS - Make sure you have the complete MPT set before assigning it to the MOD of a given function

What about modification caused by Callee's?

- This is what is being referred to as CMOD in the write-up.
- runOnSCC(CallGraph &SCC) comes to our rescue here
- Steps -
  - Get CallGraphNode *caller_node from SCC
  - Get Function* caller from the caller_node (using getFunction())
  - Get CallGraphNode *callee_node from the *caller_node
  - Get Function* callee from the callee_node (using getFunction())
  - Union Callee's MOD with the Caller's MOD (set of GlobalVariable*)

$$\text{MOD\_main} = MOD_{foo} \cup MOD_{bar} \cup LMOD_{main} \cup (MPT \,||\, \varphi)$$

What about when we have a loop within the caller and callee?



Done with MOD and MPT

This marks the end of MOD and MPT calculations for our analysis

Why did we calculate them? Because we'll use them in the flow functions for the ConstantPropAnalysis

ConstantPropAnalysis

- This is the place where you would make use of your DataFlowAnalysis written in 231DFA.h
- You'll write a **ConstPropInfo** class that inherits from Info. This implies that you'll implement at least the following functions -
  - print
  - equals
  - Join
  - Any additional getter or setters you need
- You'll implement the ConstPropAnalysis that inherits from DataFlowAnalysis. This implies you'll need to implement your flowfunction method that gets called again and again in your worklist algorithm written in 231DFA.h
- It is a forward analysis

Data Structures

```
enum ConstState { Bottom, Const, Top };
struct Const { ConstState state ; Constant* value; };
typedef std::unordered_map<Value*, struct Const> ConstPropContent;
```

Your ConstPropInfo class will make use of a instance of ConstPropContent to keep track of the information related to your analysis

ConstState is used to replicate the affect of a finite lattice (Top means not constant)

Running the Analysis

- All analyses(PA1-PA3) till now have been started in runOnFunction()
- For ConstantProp, we will run it in the doFinalization method because you have access to all the functions within the CallGraph

```
bool doFinalization(CallGraph &CG) {
    for (Function& F : CG.getModule().functions()) {
        // instantiate a new ConstPropAnalysis
        // with everything initialized to top

        //run the worklist for this function

        //print the result for this function
    }
    return false;
}
```

Flow function example

Binary operator (Unary, CMP, Select are very similar to this)

A = x + y ;

4 possibilities - x -> {constant, not constant}, y->{constant, not constant}

A = 6 + 9;

A = x + 6;

A = 9 + y;

A = x + y;

Demo?

Assume you have the required getter and setters

Another flow function - Call