

CS 6241 : Homework I

Total points : 100

Due : Beginning of class, Wednesday, January 28th 2009

Important Policies:

1. **Homeworks are non-collaborative, please direct questions regarding clarifications to the TA**
2. **Although some homework problems may have solutions available in books, internet sources etc. you are supposed to solve them on your own without looking up such solutions. This rule will be STRICTLY enforced and any act of such lookup will be considered an act of plagiarism and will result in the strictest penalty as per Georgia Tech honor code.**
3. **When necessary make suitable reasonable assumptions and clearly state them. The solutions should be neatly written/typed using standard pseudo code notations.**

Problem I (30 points) [CFGs from binaries] For optimizations, compilers statically construct CFGs and find loops based on an IR(intermediate Representations). An IR, the outcome of the front-end of a compiler, virtually retains all high-level information of source code, which allows you to build a CFG, loop structures, and any data structures for advanced data flow analysis in a straightforward way.

However, suppose that you need to do this job with a *binary executable*, not an IR. For example, security research and reverse engineering would require this kind of binary-level analysis in the absence of source code. The major differences between an IR and a binary are:

- (1) Loss of high-level source information: debugging information which may be shipped with a binary contains limited information of source code (e.g., symbol table including names of variables and functions), but a binary does not contain any high-level semantic information.
- (2) Register allocation: due to limitation of CPU registers, a code generator in the back-end must perform register allocation, mapping a large number of variables onto limited registers, which could result in generating a lot of register spill code.

One of the challenges for recovering CFG from a binary is *indirect* branches. A typical usage of indirect branch could be found in a *jump table* used by a compiler for optimizing a switch-case statement.

Consider the following a switch-case statement:

```
switch (choice)
{
    case 1: foo1(); break;
    case 2: foo2(); break;
    case 3: foo3(); break;
```

```
case 4: foo4(); break;
case 5: foo5(); break;
case 7: foo7(); break;
default: doDefault(); break;
}
```

In general, this switch-case statement is not directly translated to a bunch of if-else statements. A compiler exploits indirect branches to optimize the switch-case like the following pseudo code:

```
jump_table = {case1, case2, case3, case4, case5, caseDefault, case7};
choice = choice - 1;
if (0 <= choice && choice <= 6)
    goto jump_table[choice]; // indirect jump
else
    goto caseDefault; // direct jump
```

Note that an intermediate language (e.g., LLVM's IL) generally has a *switch* instruction. However, no machine code (e.g., x86) has this kind of high-level instruction. Also, note that we are doing this job in static time. Now you are supposed to recover a CFG from the optimized switch-case statement:

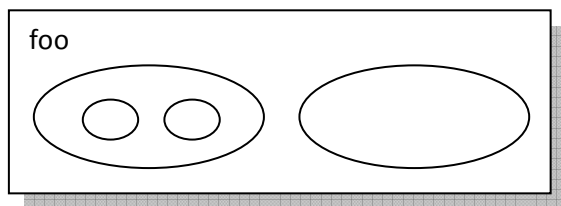
- (1) The simple CFG construction algorithm shown in the lecture note cannot appropriately handle this jump table case. Why?
- (2) Come up with some algorithms and heuristics to handle the jump table case. You should make reasonable assumptions. Hint : You need to worry about finding the target of a branch using indirect jump based on contents of a register.

Problem II (20 points): The loop detection algorithm in the lecture slide does not construct loop hierarchies (loop nests). Consider the following code in a function:

```
void foo()
{
  for (...) {
    for (...) {
      ...
    }
    while (...) {
      ...
    }
  }

  for (...) {
  }
}
```

We need a data structure that can represent loop structures in a given function. For example, the loop structure of the above function could be briefly represented like the following figure (an oval means a loop, and a rectangle represents a function.):



Suppose you are given two C++ classes (class Function and class Loop), and the basic loop discovery algorithm.

Assumptions: This algorithm works at IR level on a function by function basis (This problem is nothing to do with binaries)

- (1) Design *member variables* of these two classes that need to fully represent the relationships (a) between a function and loops within, and (b) among loops.
- (2) Write an algorithm (you may write in a pseudo code) to build the loop hierarchies. You can assume that there is class BasicBlock which represents a basic block in a function.

Problem III (35 points) [**Unreachable code detection/elimination**] Assume that you have a CFG and dominator sets generated, provide following algorithms :

1. A post-dominator(B) of a basic block is a set of nodes that must be executed after the execution of B before the function returns or exits (normally). That is, a node present in the post-dominator set is **guaranteed to execute** after B before the function returns or exits. Compute post-dominator set for all nodes using the CFG.
2. $reachable(B)$ defines a set of nodes reachable from a given basic block B. Using the CFG and post-dominator set information found in (1), write an algorithm to compute $reachable(B)$ for a given basic block B (you must use the post-dominator set information for B, zero credit otherwise).
3. Extend the solution in (2) to systematically compute the $reachable(B)$ set for **ALL** the nodes of CFG. Assume reducible flow graph. This solution should compute $reachable(B)$ in an efficient manner without replicating the work. That is you should use the knowledge of reachable set of a node to compute another node's reachable set.
4. Based on $reachable(root)$, write a simple algorithm that detects and removes unreachable basic blocks from the CFG.

Problem IV(15 points): [**Extended basic block**] An extended basic block is defined as a maximal sub-graph of CFG which does not have a join node in it except as its root node. A join node is formed at a basic block which has two or more incoming control edges. In other words, a join node starts a new extended basic block and in that extended basic block only those nodes are included which do not have a join.

1. Provide an algorithm that traverses the CFG and generates all extended basic blocks in it. In particular, you should generate $EBB(n)$ a set which defines extended basic block rooted at node n, which contains all non-join nodes reachable from n. You should generate $EBB(n)$ for all join nodes of CFG.
2. Consider a node n_1 that belongs to $EBB(n)$. What can you say about the dominator relationship of n_1 and n? Prove this result informally.