Programming Assignment 2—Natural Loop Detection

CS502 Compiling and Programming Systems

Purdue University

Background

This programming assignment (PA) requires some understanding of LLVM passes [2] where you will implement your `Natural Loop Detection` as a client analysis pass. In general, LLVM provides 4 kinds of passes working on LLVM IR at four different levels: (1) `ModulePass`, (2) `FunctionPass`, (3) `LoopPass`, and (4) `BasicBlockPass`. In this PA, you will focus on `FunctionPass` that operates at the function level. The class `FunctionPass` has an abstract virtual function, *i.e.,* `runOnFunction`, that demands user-defined `FunctionPass` subclasses to override the virtual function so that they implement it on their own. That is, any LLVM pass operating at the function level—such as `Natural Loop Detection` pass in this PA—should inherit from `FunctionPass` and override the `runOnFunction`. FYI, the return value of `runOnFunction` indicates if it changes the input function being analyzed. Here, the `runOnFunction` of the `Natural Loop Detection` pass should return false due to the lack of code change.

Note that for this PA, you must use the code base provided in Brightspace, *i.e.,* `uscc-PA2.tgz`.

Introduction

With PA1, USC compiler is now able to translate an input source file (`*.usc`) into an LLVM bitcode which is an intermediate representation (IR) used by LLVM compiler [1]; this translation process is so-called syntax-directed translation also known as IR lowering. In this assignment (PA2), students should implement an analysis for natural loop detection on top of the LLVM IR according to the algorithm covered in the class. For each LLVM IR given as an input bitcode file that exists in the `uscc/tests/` directory, students are supposed to detect natural loops therein.

Make sure you read the entire document all the way to the very end thoroughly before starting your implementation.

1 Implementation

All the necessary code for this programming assignment exists in 2 source files: `uscc/opt/Passes.h`, and `uscc/opt/NaturalLoopInfo.cpp`. The first file (`Passes.h`) declares the necessary data structures and functions used for the `Natural Loop Detection` pass, *e.g.,* `struct NaturalLoops`. The second file (`NaturalLoopInfo.cpp`) contains the implementation for all the necessary functions. For the completion of this assignment, students should only modify the `NaturalLoopInfo.cpp`.

The algorithm for natural loop detection is shown below:

1. Find a back edge (Basic Block $A \leftarrow$ Basic Block B; A dominates B)
2. Identify the nodes that are dominated by A
3. Among the nodes dominated by A, find nodes that can reach B without visiting A, *i.e.,* Such nodes + the loop header (A) = natural loop

1.1 Find a Back Edge (Step 1)

Two functions are related to Step 1, *i.e.,* `runOnFunction` and `dfsFindBackEdge`.

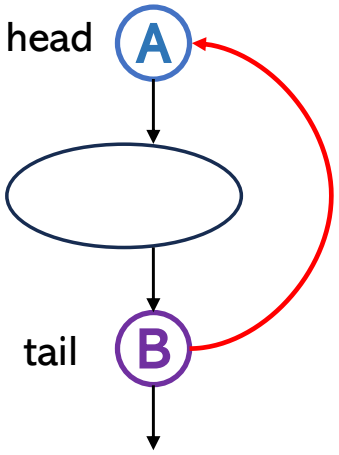

Figure 1: A back edge example.

A back edge is a directed edge (*head* $\leftarrow$ *tail*) in the control flow graph (CFG) where the `head` dominates the `tail`, as shown in Figure 1. The existence of a back edge indicates the presence of a loop, and the `head` is called the *loop header*—since it serves as an entrance to the loop.

1.1.1 runOnFunction (Function &F)

The goal of `runOnFuntion` is to detect all the back edges in the CFG of a given function `F` being analyzed. Every basic block could possibly be a loop header, so we should test all basic blocks of

the CFG in case they are the target of a back edge.

To facilitate the implementation, you should be careful with 2 relevant member variables of struct `NaturalLoops`, *i.e.,*`mHeader` and `mDT`—since they are both used for other necessary functions and thus should be set appropriately in `runOnFunction`. The first member variable `BasicBlock* mHeader` should point to a current basic block to be tested, and thus it should be updated whenever a new basic block is visited. On the other hand, the second member variable `DominatorTree* mDT` should be initialized once to the instance of a dominator tree—pre-constructed by LLVM—in the beginning of `runOnFunction`; see Tips below.

Tips: You can use `getAnalysis<DominatorTreeWrapperPass>().getDomTree()` to initialize the dominator tree `mDT`.

Again, for back edge detection, you should iterate all basic blocks in the CFG of a given function `F`, *e.g.,* "`for (auto &bb: F)`". At each iteration, make sure you set `mHeader` to the current basic block `bb` and call `dfsFindBackEdge` with the `bb` passed as an actual argument to see if `mHeader` is the target of a back edge.

1.1.2 dfsFindBackEdge (BasicBlock *current, set <BasicBlock* > &visited)

The goal of this function is to answer the query of whether the basic block `mHeader` is a loop header, in which case `findNaturalLoop` (Section 1.2.1) should be invoked to detect the corresponding natural loop.

Taking the above CFG as an example, suppose `dfsFindBackEdge` takes the basic block *A* of Figure 1 as the first argument `current` with `mHeader` set to the basic block *A*. To determine the basic block *A* is where the head of some back edge reaches, (1) you should visit *A*'s all successors traversing the CFG to check if they have an edge pointing back to *A* as in the case of *B* in Figure 1; such an edge is called *retreating edge*. If that is the case, (2) then make sure that *A* dominates such a basic block *B*. That is because not all retreating edges are a back edge unless the head of their edge dominates the tail!

(1) For the traversal of CFG, you should start from the basic block `current`, *i.e.,* the first argument of `dfsFindBackEdge`, iterating its successors with `llvm::succ_begin` and `llvm::succ_end`. It is important to note that the iterator can only access the immediate (*i.e.,* 1-level) successors of a basic block. Therefore, you need to implement depth-first search (DFS) on your own in `dfsFindBackEdge` to explore all the successors of `mHeader`—the basic block being checked for the loop header query—at each level.

(2) Every time you find a retreating edge to the `mHeader`, you should just check if it is a back edge, *i.e.,* whether the head (`mHeader`) dominates the tail (where the retreating edge starts), by leveraging the following Tips. If it turns out to be a back edge, you should invoke `printBackEdge` whose output is graded. Additionally, this is also the time to call `findNaturalLoop` (Section 1.2.1) to identify the natural loop corresponding to the back edge, *i.e.,* more precisely all the basic blocks belonging to the natural loop.

Tips: You can use `mDT.dominates(`A,B`)` to judge whether a basic block A dominates another basic block B.

1.2 Find a Natural Loop

Three functions are related to this procedure: `findNaturalLoop`, `getDominatedBlocks`, and `dfsReachable`.

1.2.1 findNaturalLoop (BasicBlock *tail)

The role of this function is to find the natural loop resulting from the back edge found in `dfsFindBackEdge`. FYI, the argument `tail` denotes the basic block whose back edge points to `mHeader`. To get the natural loop corresponding to the back edge, you should complete the algorithm's Step 2 and Step 3.

For Step 2 finding all the basic blocks that are dominated by `mHeader`, you should call the function `getDominatedBlocks` (Section 1.2.2). Then, for Step 3, you can call the function `dfsReachable` (Section 1.2.3) to find those nodes dominated by `mHeader` that can reach the `tail` without visiting the `mHeader`.

Once you identify the corresponding natural loop, make sure to print all its basic blocks by calling the function `printNaturalLoop` whose output is graded.

1.2.2 getDominatedBlocks (BasicBlock* block, vector <BasicBlock *> & dominatedBlocks)

This function is to recognize all the blocks dominated by the given block, *i.e.,* the first argument `block`.

You can use `mDT.getNode(block)` to find out the basic blocks that are dominated by the `block`. However, this function can only identify the nodes immediately dominated by `block`. Thus, to discover all its dominated nodes, you need to implement this function in a DFS manner.

1.2.3 dfsReachable (BasicBlock* current, BasicBlock* target, BasicBlock* excludeBlock, set <BasicBlock* > visitedBlocks))

The purpose of this function is to test whether the basic block `current` can reach the basic block `target` without visiting the basic block `excludeBlock`. You need to implement this function in a DFS manner as well.

2 Testing Your Code

Once you complete everything above, you can use the script `testNaturalLoop.py` provides for you to verify if your code works correctly. To perform the testing suite, you can execute the following command from `uscc/tests` directory:

$ python2 testNaturalLoop.py

Note that you must use **python2** to run the script. Using python3 will fail to pass the test cases even though your implementation is correct.

To run a single test case, you can run `../bin/uscc` command with `--natural-loop` flag that enables the `Natural Loop Detection` pass. For example, when `--natural-loop` is specified, USC compiler prints out back edges and their corresponding natural loops to the terminal as shown below:

```
$ ../bin/uscc --natural-loop quick_sort.bc
Back Edge:  while.cond <--- if.end
Natural Loop:  while.cond while.body if.then if.end
```

To check the expected output of a certain test case, you can refer to the `testcase_name.output` from the directory `tests/expected`. For example, the expected output for the test case `bubble_sort.bc` in `tests/expected/bubble_sort.output` like the following:

```
Back Edge:  for.cond <--- for.inc18
Natural Loop:  for.cond for.body for.cond1 for.body5 if.then if.end for.inc
for.end if.end17 for.inc18
Back Edge:  for.cond1 <--- for.inc
Natural Loop:  for.cond1 for.body5 if.then if.end for.inc
Back Edge:  for.cond <--- for.inc
Natural Loop:  for.cond for.body for.inc
```

where the test case `bubble_sort.bc` turns out to have three back edges and three corresponding natural loops.

For your reference, the C source file of the bitcode used in each test is under the directory `tests/`, *e.g.,* the C file of `quick_sort.bc` is in `tests/quick_sort.c`. If you would like to see the CFGs for each test case, *e.g.,* `quick_sort.bc`, you can run the following command:

```
$ ../../bin/opt -dot-cfg quick_sort.bc
```

`opt` is a tool provided in the LLVM compiler that allows you to conduct a variety of specified optimizations or analyses for a given LLVM source file. For example, by taking the option `-dot-cfg`, `opt` can analyze the input file `quick_sort.bc` and outputs its CFGs in the format of `.dot` files. Upon running this command, the CFGs of three functions of `quick_sort.bc` are

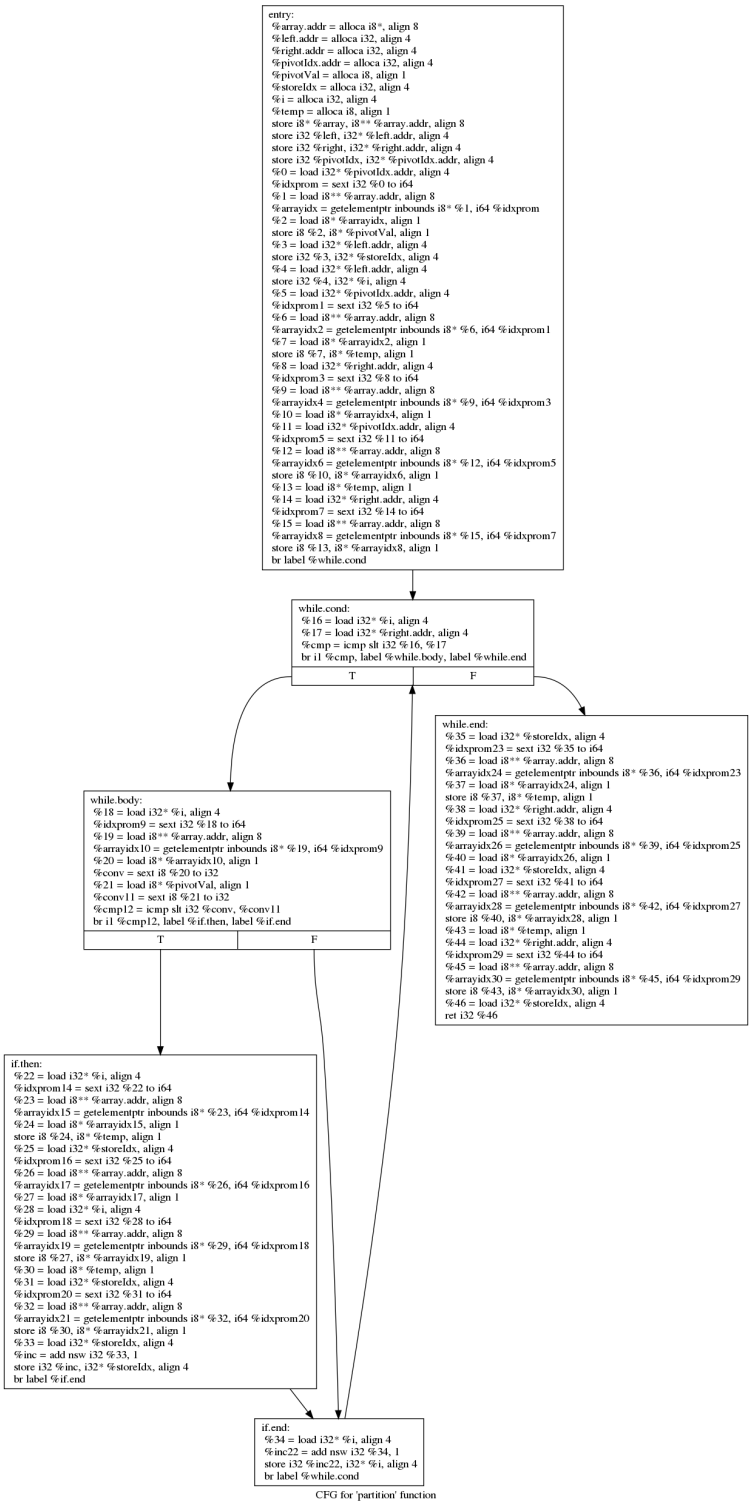

Figure 2: CFG of the partition function.

written to `cfg.partition.dot`, `cfg.quicksort.dot`, and `cfg.main.dot` (text files named according to the three functions).

To visualize the `.dot` files, you can convert them into `.png` files as shown in the following example:

```
$ dot -Tpng your_file.dot -o output.png
```

Figure 2 shows the visualized example CFG of `cfg.partition.dot`.

Conclusion

With this PA, students could understand how to implement LLVM analysis passes. More importantly, students could have a deeper understanding on natural loop detection.

Submission Guideline and Deliverables

(1) A PDF report that shows how you implemented to solve each problem.
(2) A file that compresses the entire USCC package that includes the report
Take the following steps to compress the USCC package and upload the resulting file.

- Make sure you are in the directory where you extracted uscc-PA2.tgz, *i.e.,* the directory has llvm as well as the two symbolic links to bin and lib. Then, enter the uscc directory (*i.e.,* $ cd uscc), open the README.md with your favorite text editor (*e.g.,* vim README.md), and replace the student name and email fields there with your information. Currently, the README.md has the following two lines at the end.

  Student Name(s): John Doe
  Student Email(s): boilermaker@purdue.edu

- Please place your report (it must be PDF file) under the report directory using the following command. Note that the file name must start with 'PA2' followed by your PCA (Purdue Career Account), *e.g.,* boilermaker in the example above.

  $ cp PA2-boilermaker.pdf ./report

- Execute the following three commands to remove object files and compress the entire USCC package. Again you should use the same naming convention.

  $ make clean
  $ cd ..
  $ tar cvfpz PA2-boilermaker.tgz ./uscc

- Upload the compressed file to Brightspace!

References

[1] LLVM Project. Llvm language reference manual. "https://releases.llvm.org/3.5.0/docs/LangRef.html", 2014.

[2] LLVM Project. Writing an llvm pass. "https://releases.llvm.org/3.5.0/docs/WritingAnLLVMPass.html", 2014.