(100 points) HW: Build a LLVM Branch Coverage Measurement Tool

1. Overview

In this homework, you will build a code analyzer that instruments a target program to record branch coverage in LLVM IR (i.e., clause/condition coverage) achieved in executions. Your task is to implement the code analyzer as an LLVM pass that transforms the target program in IR level. You need to measure the coverage of `grep` to demonstrate that your tool works correctly.

2. LLVM branch coverage

LLVM branch coverage checks if every branch condition (i.e., conditional jump) in a target program was ever evaluated as `true` and/or `false` in test executions. In this homework, we define a branch condition as conditions of `br` instructions and `switch` instructions in LLVM IR (e.g., we do not monitor `select` instructions). Consider that a switch statement in C has one branch condition for each case and `default`. For example, for the switch statement `switch(x){case 0:… case 1:… default: …}`, there are 3 branch conditions `(x==0)`, `(x==1)`, `(!(x==0||x==1))`.

Note. Relation between branching statements in source code and branching instructions in LLVM IR is often **non-trivial** because one condition in source code can be translated into multiple branching instructions with logical short-circuit evaluation (e.g., `c = a && b`).

3. Tool design

In this homework, you need to write an LLVM Pass that inserts probes to a target program, which measure branch coverage by monitoring which condition was evaluated and what was the result in an execution. A coverage tool can achieve this goal by inserting a probe to every branch condition related instruction to extract runtime information (actual values) on the condition evaluation.

You need to write down an LLVM Pass that automatically inserts the probe to proper locations in a target program. An LLVM Pass (generated by an LLVM front-end) traverses and modifies the IR of a target program. Once your LLVM Pass generates a modified IR by inserting the probes, an LLVM back-end generates the binary code whose execution produces the data of the coverage achievement as well.

You should implement your own LLVM Pass based on the given template (see "**LLVM How-to for Homework**"). The source code of a sample LLVM Pass `IntWrite` is also given to you. You can reuse the source code of the LLVM Pass and the runtime module.

4. Tasks

You need to complete the following specific tasks, and **describe the results clearly** in your reports:

1. (15 points) Explain how branching statements and logical operations in C (e.g., `if`, `for`, `switch`, `c?v1:v2`, `c=a&&b`, etc.) are translated into LLVM IR instructions.

2. (5 points) Describe at which location your tool needs to insert a probe and what type of information a probe monitors from a target program

3. (70 points) Write your LLVM Pass and the probe program. You should construct the LLVM Pass based on the given template `CCov.cpp`. You should write the probe function as well. Your tool should meet the following requirements:

A. Your code should have sufficient comments that document your design.

B. Your tool must report how many branch conditions in LLVM-IR exist in the target program.

C. An instrumented target program should produce a file `coverage.dat` that reports how many times each branch is covered in an execution. A branch condition is identified as a pair of line number and an integer index that differentiate the condition from the other conditions of the same line number (e.g., `311.0`, `311.1` and so on).

D. If an instrumented target program is executed multiple times, `coverage.dat` must contain the accumulated result.

E. `Coverage.dat` looks like below:

```
…
311.0 -> 20, 2
311.1 -> 20, 0
311.2 -> 18, 2
…
Total: XXX branches, Covered: YYY branches
```

This is from the result with `grep.c`. (the line 311 in `grep.c` 3 branch conditions). At each line, "`X.Y -> n, m`" means that a branch condition located at line `X` with an index `Y` was evaluated as true for `n` times and evaluated as false for `m` time in the previous test executions so far. The last line reports total number of in a target program, and how many of these were ever covered in executions.

For a `switch` instruction, coverage should be printed for each `case` and (implicit or explicit) `default` as follows:

- `X` in "`X.Y -> n, m`" for each `case` and (implicit or explicit) `default` should be a line # of the `switch` statement.
- `m` in "`X.Y -> n, m`" for each `case` and (implicit or explicit) `default` should be always 0.

For an example of the following C code,

```
10: switch(x){
11:   case 4: …
12:   case 8: …
13:   default: …}
```

suppose that `case 4`, `case 8`, and `default` are executed 3, 4, and 5 times, respectively. Then, the coverage output will be as follows:

```
10.0 -> 3, 0
10.1 -> 4, 0
10.2 -> 5, 0
```

4. (5 points) Apply your tool to `grep.c` (not preprocessed `grep.c`), and submit the output (`coverage.dat`) and report the total branch coverage (i.e., the last line of `coverage.dat`) after executing the instrumented `grep` with the following test cases:

```
./grep -n "if" grep.c
./grep -E "[0-9][0-9]+" grep.c
./grep -E "[[:digit:]][[:alpha:]]" grep.c
```

5. (5 points) Discuss advantages/disadvantages of instrumenting LLVM IR over instrumenting C source code (as you did in your `kcov` homework).

Submission

Your HW report should clearly describe all results of the tasks. Your implementation should include the source code of both your LLVM Pass class and the runtime module.

The homework is evaluated primary based on your HW report which should contain the answer for every question explicitly. If your report and/or softcopy are difficult to understand, you may not get full points. The source code must be able to be compiled with LLVM.