CS510'15 Project #4 Program Slicing using LLVM
(Due: Apr-30, 2015)

April 6, 2015

1 Goal

In this project, you will write an LLVM pass that returns a statically sliced version of a function.

2 Slices

A slice of variable v at statement S is all the statements involved in computing v's value at S. Also see the slides from the class and the referenced research papers for a more elaborate discussion of slicing.

3 Slices in our project

For the purposes of this assignment we make the following restrictions:

- You will only be asked to compute the slice of the return value of a function.
- The function you're meant to slice in the test case is always named "pie" so you can easily identify it.
- pie will only have a single return location (instruction).
- pie's return type will always be $i32$.

For example, a test case might include:

```
; Function Attrs: nounwind uwtable
define i32 @pie(i32 %x) #0 {
entry:
  %mul = shl nsw i32 %x, 1
  %0 = load i32* @y, align 4
  %cmp = icmp eq i32 %0, 10
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  store i32 -2, i32* @y, align 4
  br label %if.end

if.end:                                           ; preds = %if.
    then, %entry
  ret i32 %mul
}
```

If you replace the body of the function with the minimal slice the result is as follows:

```
; Function Attrs: nounwind uwtable
define i32 @pie(i32 %x) #0 {
entry:
  %mul = shl nsw i32 %x, 1
  ret i32 %mul
}
```

Note that this example was generated by the following commands:

```
clang -S -emit-llvm tc1.c
opt -S -mem2reg -std-compile-opts -disable-inling < tc1.ll >
    tc1_m2r.ll
```

☞ You should use the provided version of LLVM (3.5.1) to generate your test cases, i.e. "clang" in the above example should refer to the clang that is built when you compile your code (under project4/build/bin/).

Clang 3.4, 3.5, and 3.5.1 all produce semantically equivalent IR. If your pass works for one of these versions, but not the others, you've made a mistake.

4 Your code

Just like project2, you will write an LLVM pass. This time the output of your pass is the modified version of the input file. To do this, you don't need

to do anything other than modify the module that is given to your pass by runOnModule. When you run:

```
bin/opt -load lib/LLVMFunctionSlicePass.so -FunctionSlicePass <
    tc1_m2r.ll > tc2_out.ll
```

opt will automatically write the module (that your pass should have modified) to the file tc2_out.ll.

All of your code should be in project4/FunctionSlicePass.cpp. There is a relative symlink to this file in the llvm repository dicussed in the next section.

5 Building LLVM

Just like project2, you will need to build LLVM from source, but there will be one additional step this time. To reduce the size of your repos, the LLVM source will itself will not be part of your repository. This is not a problem because you're not allowed to change any files other than project4/FunctionSlicePass.cpp.

These are the commands you need to run to build LLVM and your pass:

```
  git clone <your repository repo>
  cd <your repository name>
  cd project4
  git clone https://bitbucket.org/scottcarr/llvm.git
  mkdir build
  cd build
  cmake -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++ \
     -DLLVM_INCLUDE_TESTS=OFF -DLLVM_TARGETS_TO_BUILD=X86 \
     ../llvm
  make -j8
```

Listing 1: Build Commands

Note that this requires `clang` and `clang++` to be installed on your system. If you're worried about versions, your installed clang should be 3.4. We also recommend you install ld gold on your machine. If you're on a recent Ubuntu run:

```
  sudo apt-get install binutils-gold binutils-dev
  cd /usr/bin
  sudo rm ld
  sudo ln -s ld.gold ld
```

6 How to submit

Like project2, you will submit your code in BitBucket. You are only allowed to change the file project4/FunctionSlicePass.cpp.

☞ Do not commit any files other than project4/FunctionSlicePass.cpp. Do not add your build directory. Do not add the llvm directory.

7 Grading

Your code needs to compile on Ubuntu 14.04 in order to be graded. This will not be an issue if you followed the instruction to only add code to FunctionSlicePass.cpp.

To build your code, the TA will use the exact same set of commands you used to build your code in Listing 1. It is your responsibility to make sure your code builds using those exact commands.

To run the test cases, the TA will first build your code, then from the build directory run:

```
1 bin/opt -load lib/LLVMFunctionSlicePass.so -FunctionSlicePass <
    testcase.ll > testcase_out.ll
2 bin/clang testcase_out.ll
3 ./a.out
```

testcase.ll will contain other code, including code that will test your modified pie function. You are not allowed to modify anything except the body of the pie function.

For each test case, your output will be graded on two criteria:

7.1 Grading Criteria 1: Correctness

For each test case, the TA will generate many input/output pairs from the original unsliced version of pie. Your sliced version of pie should give the same output for the same input as the original version of pie. Of course the original pie function will be deterministic.

7.2 Grading Criteria 2: Size

Your goal is to find the minimal slice in terms of the fewest number of instructions. The smaller the slice, the more points you will get. You will not get any points for incorrect slices, no matter how small.

Side note: there is a way to combine instructions in LLVM using constant expressions, but this won't count towards reducing the size of the slice. Each constant expression counts as one instruction even if multiple constant expressions or instructions appear on the same line.