Lab 2: Dataflow

Fall Semester 2021

Due: 27 September, 8:00 a.m. Eastern Time

Corresponding Lecture: Lesson 5 (Dataflow Analysis)

Objective

This lab will familiarize you with writing static program analyses using the LLVM compiler infrastructure. LLVM is a collection of compiler and analysis toolchain utilities widely used in the software analysis community. You will use LLVM to implement two intra-procedural dataflow analyses, one forward (reaching definitions analysis) and one backward (liveness analysis). In LLVM, these are referred to as passes over the code.

Note on Past Issues

In past semesters, it has caused a high number of students to be submitted to the Office of Student Integrity for Academic Integrity violations. In particular, it's possible to find solutions to similar analyses on the internet. Looking at these solutions in any form is likely to influence your thinking and cause your solutions to be similar to them, which is an Academic Integrity violation in this class. If you are unclear of our guidelines for what is collaboration and what is cheating, we suggest reviewing that section of the syllabus. If you have any questions about what is allowed and what is not allowed, please reach out to *Instructors* via Ed Discussions for clarification.

Students who submit solutions found to be similar to online resources or other students should expect a 0 grade on the lab, a disciplinary record of an Academic Integrity issue through the Office of Student Integrity, and will not be eligible to receive a final grade of A in the course. Students who have had past Academic Integrity issues may find that OSI assigns them higher penalties.

General Resources

- Getting Started with LLVM:

  http://llvm.org

  http://releases.llvm.org/8.0.0/docs/GettingStarted.html#an-example-using-the-llvm-tool-chain
- LLVM Documentation:

  http://releases.llvm.org/8.0.0/docs/index.html
  http://releases.llvm.org/8.0.0/docs/LangRef.html
  http://llvm.org/doxygen/index.html
- C++ Pointers / References:

  http://www.cplusplus.com/doc/tutorial/pointers/

https://www.geeksforgeeks.org/pointers-vs-references-cpp/

- SSA Intermediate Representation Form (used in LLVM bitcode):
  https://en.wikipedia.org/wiki/Static_single_assignment_form

Programmer's Manual Resources

- Programmer's Manual:
  http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html
- Using `isa<>` to check whether a pointer points to an instance of a particular type:
  http://llvm.org/docs/ProgrammersManual.html#the-isa-cast-and-dyn-cast-templates
- Enumerating basic blocks and instructions in a function:
  http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#basic-inspection-and-traversal-routines
  http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#iterating-over-predecessors-successors-of-blocks
- Important classes:
  http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#the-value-class
  http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#the-user-class
  http://releases.llvm.org/8.0.0/docs/ProgrammersManual.html#the-instruction-class
  http://llvm.org/docs/doxygen/html/classllvm_1_1ValueMap.html
  http://llvm.org/docs/doxygen/html/classllvm_1_1SetVector.html

Lab Setup

1. Download and extract the lab code found on Canvas in the `dataflow.zip` archive.
2. Navigate to `dataflow/build` and run the following commands:

```
cmake ..
make clean
make
```

   You should now see `libDataflowPass.so` under `dataflow/build/dataflow`.

3. Go to the `dataflow/example` directory and generate LLVM bitcode from the programs we will analyze with the following commands:

```
clang -emit-llvm ArrayDemo.c -c -o ArrayDemo.bc
```

```
            clang -emit-llvm Greatest.c -c -o Greatest.bc
```

4. Run a Dataflow pass using the command below to ensure everything works as expected for the test program `ArrayDemo.c`. This pass demonstrates the API by printing the definitions, uses, predecessors, and successors of each instruction.

```
opt -load ../build/dataflow/libDataflowPass.so -Printer < ArrayDemo.bc > /dev/null
```

   In addition to testing your setup, the printer pass (in `Printer.cpp`) is a very useful reference for implementing your own passes.

Lab Instructions

Complete the `doAnalysis` method in the `ReachDefAnalysis.cpp` and `LivenessAnalysis.cpp` (located in `dataflow/dataflow/`) skeleton files to implement the two analyses. Do not write your analysis code outside of these files, as these files are the only ones you will submit. You may use the C++ Standard Template Library (STL) when implementing your analyses and the functionality provided through LLVM, but you may not use other third party libraries.

Your code will need to iterate over the program points in the input program and store the computed dataflow facts in `DataflowAnalysis::inMap` and `DataflowAnalysis::outMap`. Both analyses inherit from the base class `DataflowAnalysis`, which you can find in the header file `DataflowAnalysis.h` located in the directory `dataflow/dataflow/`. Besides including useful classes such as `SetVector` and `ValueMap`, `DataflowAnalysis.h` also defines useful utility functions such as `getPredecessors`, `getSuccessors`, and `isDef`.

LLVM passes are performed on an intermediate representation (LLVM bitcode) generated from the program source code. LLVM bitcode is generated in Static Single Assignment (SSA) form, a common simplification used by compiler infrastructure. In SSA form, each variable is assigned exactly once, and every variable is defined before it is used. Variables are represented directly by the instruction defining it. In fact, the `Instruction` class is a subtype of the `Value` class. You can check whether an instruction defines a variable by checking `getType()->isVoidTy()`. An entry into the `inMap` or `outMap` is the instruction as opposed to a variable. The lectures speak to variables being stored (`x` or `y`) but with the SSA representation being used, the instruction itself is a proxy for the variable. That is why the `inMap` and `outMap` all show the instructions in the printer output.
Since each variable is uniquely defined, variables will never be redefined in SSA form. This will affect the generation of KILL sets in your implementation of reaching definitions analysis, specifically, they will always be empty.

After completing the two `doAnalysis` methods, rebuild the analyses using the commands from setup step 3, and then rerun the analyses using following commands to print the results of the analyses on the `ArrayDemo` program:

```
opt -load ../build/Dataflow/libDataflowPass.so -ReachDef < ArrayDemo.bc > /dev/null
opt -load ../build/Dataflow/libDataflowPass.so -Liveness < ArrayDemo.bc > /dev/null
```

If your implementation is correct, your output will match the example output in `ArrayDemo_ReachDef` and `ArrayDemo_Liveness` found in `dataflow/example/`. The order of elements in the IN and OUT sets does not matter, but the number of elements and the values should match exactly. Please note that if your implementation produces extra console output beyond the sets, we will not consider your output as matching.

We have also included another program, `Greatest.c`, and it's expected outputs for testing your implementation. You can use commands similar to those above to analyze this program. We also encourage students to develop their own test cases / corresponding output and share them on Ed Discussions, although we will not be validating them for correctness.

Additionally, we have provided images of the control flow graphs (CFGs) for both programs, named `array_demo_graph.gif` and `greatest_graph.gif`. Each instruction is represented by a node, and edges represent an instruction's successor instruction(s). These graphs are useful for reasoning about the correctness of your `IN` and `OUT` sets for the sample program analyses. We recommend that you take a moment to review the graph and ensure you understand the structure of each program. Pay close attention to the instructions that have multiple successor and/or predecessor instructions.

Helpful LLVM API Information

`ValueMap` has a similar interface to `std::map`. For example, you can access or insert an element using the `[]` operator:

```
ValueMap<Instruction*, int> vm;
Instruction* I = /*...*/;
vm[I] = 5; // inserts <I, 5> to the map
```

LLVM will generate all of the necessary objects for your analyses according to its object model (see `DataflowAnalysis.cpp`). You will not need to create new elements to insert into

`DataflowAnalysis::inMap` or `DataflowAnalysis::outMap`, your code should add the pointer to the object to the `ValueMap`.

`SetVector` has a similar interface to `std::vector`, except that it does not permit inserting duplicate elements. The `==` operator returns `false` for two `SetVector` objects containing the same elements in different orders.

Also, functions in the `<algorithm>` library from the STL work with `SetVector`. For example:

```
    SetVector<int> sv;
    for( int i = 0; i < 5; i++ )
        sv.insert(i);
    sv.insert(0); // has no effect
    if (std::all_of(sv.begin(), sv.end(), isPositive))
        // all_of is from <algorithm>
        printf("All numbers in sv are positive!");
```

assuming `isPositive` is defined elsewhere as:

```
        bool isPositive(int i) {return i > 0;}
```

Items to Submit

For this lab, we will be using Gradescope to submit and grade your assignment. For full credit, the files must be properly named and there must not be any subfolders or additional files or folders. You must include meaningful type annotations, and all public and hidden tests must pass. Submit the following files through Gradescope.

- Submit `LivenessAnalysis.cpp` (50 points)
- Submit `ReachDefAnalysis.cpp` (50 points)

Do not submit anything else. Make sure all of your code modifications are in the files listed above as your other files will not be submitted. In particular, past students have modified header files to import additional headers, leading to a compile error in their submission.

Through Gradescope, you will have immediate feedback from the autograder as to whether the submission was structured correctly as well as verification that your code successfully runs the public tests on the grading machine. The public tests are the same tests provided to you as part of the lab, but these tests won’t (and are not intended to) catch everything. You will get your final score after the assignment due date.

Grading Criteria

Generally, credit for this lab is awarded as follows:

- For each program, both analyses will be equally weighted
- The inset and outset of each analysis will be compared against the correct values, with each equally weighted
- For each set, your score will be $\frac{[\# \mathit{expected\ tuples}]-[\# \mathit{missing\ tuples}]-[\# \mathit{extra\ tuples}]}{[\# \mathit{expected\ tuples}]}\%$ of the total possible points for that set. For example, an output with 10 expected tuples where you found all 10 expected tuples but also two additional tuples would score $\frac{10-0-2}{10}\% = 80\%$of the possible credit

Your dataflow analyses will be graded against multiple programs, including the benchmark programs provided as a part of this lab assignment, which will constitute at least half of your grade. In general, the programs used in grading but not provided as part of the lab are of the same order of complexity as the provided programs. **Please note that your analysis must not produce any output beyond the in and out set values when run.**

While efficiency is important, it is entirely secondary to correctness. You will not gain points for an efficient yet incorrect algorithm. There is a time limit several times longer than the expected execution time for an efficient algorithm in the grader. Any analysis that has not completed at the end of this time limit will not be considered correct as will any analysis that causes a crash.

How Do We Use It?

Dataflow analysis techniques like the ones you implemented in this lab are increasingly used by IDEs to help developers write good code. For example, current versions of Microsoft Visual Studio are able to use a combination of the techniques you implemented to tell you that a variable you've declared in your code is never used - without ever executing your code. Static dataflow analysis techniques such as the two that you implemented in this lab are the foundation of modern code linters.