Lab 3: Datalog

Fall Semester 2021
Due: 18 October, 8:00 a.m. Eastern Time
Corresponding Lecture: Lesson 7 (Constraint Based Analysis)

Synopsis

Writing a constraint-based static analysis for C programs with LLVM and Z3. Approaching the same goals of "Lab 2: Dataflow" from a different direction.

Objective

In this lab, you will implement a constraint-based analysis on simple C programs. You will discover relevant facts about LLVM IR Instructions, feed those facts into the Z3 constraint solver, then implement the rules of reaching definitions and liveness analysis in Z3's C++ API.

Resources

- Z3 tutorial and C++ API
    - https://rise4fun.com/z3/tutorial/guide
    - https://z3prover.github.io/api/html/group__cppapi.html
    - https://github.com/Z3Prover/z3/blob/master/examples/c%2B%2B/example.cpp
- Important classes
    - https://z3prover.github.io/api/html/classz3_1_1fixedpoint.html
    - https://z3prover.github.io/api/html/classz3_1_1expr.html

Setup

Download `datalog.zip` from Canvas and unzip in the home directory on the VM. This will create a `datalog` directory.

The following commands set up the lab:

```
$ cd ~/datalog
$ mkdir build && cd build
$ cmake ..
$ make
```

The above command will generate a Makefile using CMake, then generate an executable file `constraint` that calculates either reaching definitions or liveness analysis:

```
$ cd ~/datalog/test
$ clang -emit-llvm -S -c -o Greatest.bc Greatest.c
```

```
$ ../build/constraint -ReachDef Greatest.bc
```

If you've done everything correctly up to this point you should have lots of lines showing empty `In` and `Out` sets for every instruction in `Greatest.bc`.

You can expect to re-run the `make` command often as you work through the lab. You should not need to re-run `cmake`.

Lab Instructions

In this lab, you will design a reaching definition analysis, then a live variables analysis, using Z3. The main tasks are to design the analysis in the form of Datalog rules through the Z3 C++ API, and implement a function that extracts logical constraints in the form of Datalog facts for each LLVM instruction.

We will then feed these constraints, along with your datalog rules, into the Z3 solver. The `main` function of `src/Constraint.cpp` ties this logic together, and provides comments to explain how the main components work together.

In short, the lab consists of the following tasks:
1. Write Datalog rules in the `initialize` function in `Extractor.cpp` to define the reaching definition analysis and live variable analysis.
2. Write the `extractContraints` function in `Extractor.cpp` that extracts Datalog facts from LLVM IR `Instruction`. You will likely need to extract different facts for reaching definitions analysis and liveness analysis.

Relations for Datalog Analysis. The skeleton code provides the definitions of necessary Datalog relations over LLVM IR in `Extractor.h`. In the following subsection, we will show how to represent a LLVM IR program using these relations.

The relations for the reaching definition analysis are as follows:
- `Kill(X,Y)`: Definition Y is killed by instruction X
- `Gen(X,Y)` : Definition Y is generated by instruction X
- `Next(X,Y)` : Instruction Y is an immediate successor of instruction X
- `In(X,Y)` : Definition Y may reach the program point immediately before instruction X
- `Out(X,Y)` : Definition Y may reach the program point immediately after instruction X

You will use these relations to build rules for both analyses in `Extractor.cpp`.

Defining Datalog Rules from C++ API. You will write your Datalog rules in the function `initialize` using the relations above. Consider an example Datalog rule:

A(X, Y) :- B(X, Z), C(Z, Y).

This rule corresponds to the following formula:
$\forall X, Y, Z.\ B(X, Z)\ \wedge\ C(Z, Y)\ \Rightarrow A(X, Y).$

In Z3, you can specify the formula in the following sequence of APIs in `initialize`. Assume `ctx` and `Solver` are configured as shown in Extractor.cpp. They represent instances of a Z3Context and a *fixedpoint* constraint solver, respectively.

```
/* Declare functions */
z3::func_decl A = ctx.function("A", ctx.bv_sort(32));
z3::func_decl B = ctx.function("B", ctx.bv_sort(32));
z3::func_decl C = ctx.function("C", ctx.bv_sort(32));
/* Declare quantified variables */
z3::expr X = ctx.bv_const("X", 32);   // encode X as a 32-bit bitvector (bv)
z3::expr Y = ctx.bv_const("Y", 32);
z3::expr Z = ctx.bv_const("Z", 32);
/* Define and register rules */
z3::expr R0 = z3::forall(X, Y, Z, z3::implies(B(X,Z) && C(Z, Y), A(X,Y)));
Solver->add_rule(R0, ctx.str_symbol("R0"));
```

Study the above pattern with `forall` and `implies` closely, as you can adapt it to express all the Datalog relations required to complete this lab.

Extracting Datalog Facts. You will need to implement the function `extractConstraints` in `Extractor.cpp` to extract Datalog facts for each LLVM instruction. The skeleton code provides a couple of auxiliary functions in `src/Extract.cpp` and `src/Utils.cpp` help you with this task:
  - `void addX(const InstMapTy &InstMap, ...)`
    - X denotes the name of a relation. These functions add a fact of X to the solver. It takes `InstMap` that encodes each LLVM instruction as an integer. This map is initialized in the `main` function
  - `vector<Instruction*> getPredecessors(Instruction *I)`
    - Returns a set of predecessors of a given LLVM instruction `I`
  - `bool isDef(Instruction *I)`
    - Behaves the same as Lab 2, it returns true iff instruction `I` defines a variable

Miscellaneous.

- For convenience for easy debugging, you can use the `toString(Value *)` function in `Utils.cpp`
- If the `--debug` option is passed through the command line `constraint -ReachDef --debug`), it will print out several relations. You can extend the `print` function in `Extractor.h` for your local development purposes, but you cannot submit `Extractor.h` so use caution if you change it

Input & Expected Output

You will be working with the same sample programs as Lab 2, `ArrayDemo.c` and `Greatest.c`. In Lab 2, you passed bitcode for those programs into an opt pass. In Lab 3, you are passing bitcode for those programs into the `constraint` executable.

A Makefile is provided in the `test` directory to run both sample programs through Reaching Definitions Analysis and Liveness Analysis, redirecting output to files. You can use `make` to test everything, or you can invoke constraint individually like so:

```
$ cd ~/LLVMDatalog/test
$ clang -emit-llvm -c -o Greatest.bc Greatest.c
$ clang -emit-llvm -c -o ArrayDemo.bc ArrayDemo.c
$ ../build/constraint -ReachDef Greatest.bc
$ ../build/constraint -ReachDef ArrayDemo.bc
$ ../build/constraint -Liveness Greatest.bc
$ ../build/constraint -Liveness ArrayDemo.bc
```

`constraint` will produce output formatted just like the opt pass you built in Lab 2. If you build and run the unmodified skeleton, you'll see lots of empty IN and OUT sets per instruction:

```
Instruction:    %1 = alloca i32, align 4
In set:
[]
Out set:
[]
```

The contents of your IN and OUT sets matter, the order does not. Do not worry if the elements of your IN and OUT sets appear in a different order than the provided reference output.

Grading

- The In and Out relations of each test program will be compared against the correct values, with each equally weighted

- For each test program, your score will be $\frac{[\text{\# expected tuples}] - [\text{\# missing tuples}] - [\text{\# extra tuples}]}{[\text{\# expected tuples}]}$ % 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

- We may use additional test programs for grading not supplied in the lab zip. If used, these additional test programs would be of similar size and complexity to the ones you have seen.

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. Submit your single file `Extractor.cpp` via Gradescope.

Do not submit anything else. Make sure all of your code modifications are in the file 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 tests on the grading machine. For Lab 3, Gradescope will provide a score that includes all tests released with the project starter, and additional hidden tests.

How Do We Use It?

While the analyses types are very similar to the last lab so all those applications are still true. Our main purpose in this lab is to show you how declarative and imperative syntax can be used to accomplish the same goal. While we introduced you to a new language that is explicitly declarative, it turns out that many popular languages have adopted some declarative syntax. For example, many languages have a concept of contains today. Contains is declarative programming at its simplest: we are declaring what we want to accomplish without specifying how the language accomplishes the task.

Java Imperative Example

```java
List<String> colors = List.of("red", "green", "blue", "yellow", "orange");
for (String color : colors) {
    if (name.equals("black")) {
        System.out.println("Found");
        break;
    }
}
```
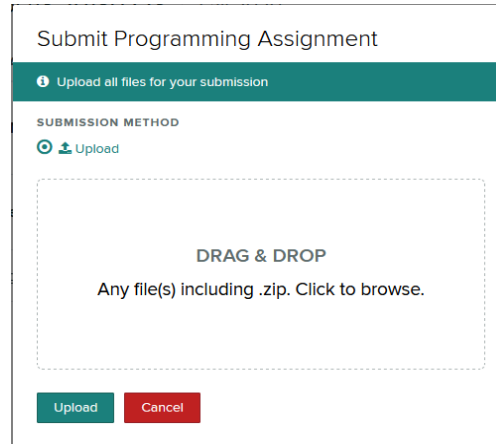
Java Declarative Example

```java
List<String> colors = List.of("red", "green", "blue", "yellow", "orange");
if(colors.contains("black")) {
    System.out.println("Found");
}
```

Notice the declarative example is shorter and clearer about what is being accomplished with no indication of how the compiler will interpret the `contains` function into bytecode.

Many developers have switched to the declarative paradigm without realizing that they are using a programming best practice. One [Java study](#) found that at worst, declarative syntax performed the same as imperative syntax and in many cases, performed better. As discussed in the lesson, a declarative implementation tends to improve as the language changes over time, benefiting from new optimizations, where an imperative implementation generally has constant performance over time. Java 8's declarative implementations perform significantly worse than Java 11's implementations, yet the older code can be run in the newer environment with better performance and both versions perform better than the most common imperative implementations.

Tips for submitting your assignment with Gradescope

- When you go to the Canvas Assignment, you will see a blank Gradescope submission window embedded in the Canvas Page. Follow the on-screen prompt to submit your files.
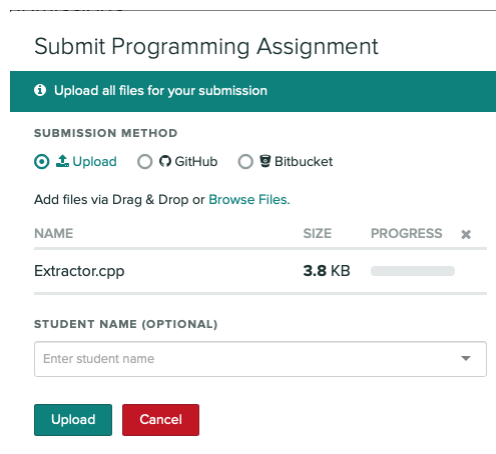
- If there are multiple files in this lab, you can zip the files together and upload the single zip file to the submittal page, or you may select each file individually on the page. If you select the files individually, they must all be uploaded in the same submission.
- A correct submission should look like this image (ignoring size and order):

- You may resubmit your assignment as many times as you want up to the assignment due date. The **Active** submission will be the final grade. By default, every time you upload a new submission, it will become the **Active** submission, but you can change the **Active** submission in your **Submission History** window if needed.
- If you resubmit your assignment, you must include all files as part of the new submissions.
- We provide comments as output of your submission. Use these to make sure your submission is as complete as possible. For example, if you see a compile error, you will not receive any points for that part of the assignment.