Programming Assignment 3

Overview

In the class we have learned a few different data-flow analysis like liveness, reaching definition, etc. As a good software engineering practice, it makes sense to implement a generalized iterative data-flow analysis framework and then customize it for different purposes. In the assignment, you are required to design a generalized framework first and then use this framework to write a liveness analysis and reaching definition analysis.

Generalized Data-flow Analysis Framework

You will implement a general parameterized iterative data-flow analysis framework that allows a developer to easily implement any unidirectional data-flow analysis pass by providing the following information:

1. Domain, i.e., the data structure to represent data-flow values, for liveness analysis it is the set of variables or a bit vector of variables
2. Direction (forwards or backwards)
3. Transfer functions
4. Meet operator
5. Initial flow values
6. Boundary condition (flow value for entry node—or exit node for a backwards analysis)

Note: you can use either class inheritance or template class to implement this framework. There are numerous online resource to learn how to write inheritance and template code in C++.

In this assignment, we do not have hard requirement on how the analysis framework should be implemented except that it is abstracted and generic. To give you an example, here is one possible implementation strategy:

1. Write the analysis framework as a template base class from which all the analyses can derive.
2. The operations, such as the meet operator, are pure virtual methods that will be implemented in the subclass.
3. The type used to represent values from the domain is a type parameter for the template.
4. Give a careful thought about how the other analysis parameters are represented. For example, direction could reasonably be represented as a boolean.

Data-flow Analysis

You will now use your iterative data-flow analysis framework to implement Liveness and Reaching Definitions. If you wish to use a bit vector to represent the sets you should look at `llvm::BitVector`, `llvm::SparseBitVector`, or `std::bitset`; however, you can use a different data type if you think it is more appropriate. A utility Annotator pass that helps print out the analysis result will be included in the skeleton code. Use it to dump whatever information your analysis finds to stderr.

Liveness. On convergence, your Liveness pass should report for each program point all variables that are live at that point. You might debug your code by comparing it against the results of LLVM's Liveness pass. Please call this pass "live".

Reaching Definitions. On convergence, your Reaching Definitions pass should report for each program point, all the definition sites that reach that program point. Please call this pass "reach".

Output Format. The annotator pass will call `getInState(BasicBlock *)` and `getOutState(BasicBlock *)` of your liveness and reaching definition analysis and iterate every variable in the set and prints them. So your analysis should implement these two methods and return values correctly. Although in class, we mentioned that the domain for reaching definitions are sets of definitions and the domain for liveness are sets of variables, in LLVM, they are both represented as sets of llvm Values (why?). Note that your generalized framework should support more domains than sets of variables, but your analysis are only providing sets of llvm values to the annotator class. Also, we only test Basic Blocks as a program point but your framework should be able to handle instructions as program point.

Implementation Issues

LLVM's representation of Single Static Assignment (SSA) form presents some unique challenges when performing iterative data-flow analysis.

1. Values in LLVM are represented by the Value class. In SSA form, every value has a single definition, so instead of representing values as some distinct variable or pseudo register class, LLVM represents values by their defining instruction. That is, Instruction is a subclass of Value. There are other subclasses of Value, such as basic blocks, constants, and function arguments. For this assignment, we will only track the liveness of instruction-defined values and function arguments.
2. Phi instructions are pseudo instructions that are used in the SSA representation and need to be handled specially by both Liveness and Reaching Definitions.You should carefully consider how your analysis passes are affected by Phi instructions. The fact that you will be working on code in SSA form means that computed values are never destroyed. Think carefully about the ramifications of this fact on your implementation.

3. To guide you in formatting the output of your passes, we have provided an example output in the skeleton code.
4. Please do not use any LLVM API or code that could trivialize this assignment, like LLVM's internal liveness pass, etc. If you are in doubt as to whether an API is allowed, just ask on Piazza.

Submission

Programming Assignment 2 is due on Thursday, Apr 13th, 11:59pm. You will be submitting all your code, your own test cases (if there is any) and a report describing your solution and the structure of your archive (less than 3 pages). If you made any changes that is different from the provided running script, explain them in the report and how to test your code.

In your report, you MUST include all the design decisions you made, like how you design the framework, what's the parameter, and how you handle Phi instructions, etc. If you are not sure what to include in the report, just ask on Piazza. I will make a post on Piazza to summarize your questions and you should check it regularly.

Acknowledgement

This assignment is adapted from CS 380C, taught by Prof. Calvin Lin and Jia Chen (TA).