CS6383 : Assignment #1

README FILE

Raj Ambekar (CS22MTECH12008)
Soumya Banerjee (CS22MTECH12011)

January 29, 2023

Introduction

In LLVM ,optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program. There are 3 categories of passes:

- **Analysis pass:** Analysis passes compute information that other passes can use or for debugging or program visualization purposes.
- **Transformation pass** :Transform passes can use (or invalidate) the analysis passes. Transform passes all mutate the program in some way.
- **Utility pass:** Utility passes provide some utility but don't otherwise fit categorization. For example passes to extract functions to bitcode or write a module to bitcode are neither analysis nor transform passes.

In our code, we have used the ModulePass and the FunctionPass.

The Module pass class : The ModulePass class is the most general of all superclasses that you can use. Deriving from ModulePass indicates that your pass uses the entire program as a unit, referring to function bodies in no predictable order, or adding and removing functions.

The function pass class : In contrast to ModulePass subclasses, FunctionPass subclasses do have a predictable, local behavior that can be expected by the system. All FunctionPass executes on each function in the program independent of all of the other functions in the program. FunctionPasses do not require that they are executed in a particular order, and FunctionPasses do not modify external functions.

Implementation Details

In the ModulePass,we have printed the Clang version, Source repository, and Commit hash. We have used NamedMetaData to retrieve the information in one line,and then we have used substring to separate the print statements in different lines.

We have also printed the target. We have implemented the Module and Function pass in one code, so the ModulePass calls the runOnFunction() so that the other details like footprint, scope etc can be printed.

The var-name is given as input from the CLI, so the FunctionPass calculates the scope, footprint,i.e line numbers where the variable is accessed and the number of reads and writes on the basis of the load and store instructions. We have used a set to store the line numbers and printed it as the footprint, and the scope has been considered as the first point and last point where the variable has been accessed.

Remarks and Status

The starting scope of the variable is the first time when the input variable is initialized with some value and not just declared. The footprint is also calculated keeping in mind the similar concept. For cases not consisting of pointers, our code gives the correct output, however for complex codes consisting of multiple scopes and the same variable name, it may not give the desired result. The number of reads and writes is also calculated on the basis of load and store instructions, so number of reads means number of times load has happened and write means the number of times store instruction has occurred.

The zip file provided maintains the structure as shown in the assignment pdf, so we have provided the Footprint.cpp file, two CmakeLists.txt files maintaining the structure of lib/Transforms and the test folder consists of 5 non-trivial programs. This Readme file is also a part of the zip file provided outside any directory as shown in the assignment pdf.

References

- https://llvm.org/doxygen/classllvm_1_1Module.html
- https://llvm.org/doxygen/classllvm_1_1ConstantInt.html
- https://llvm.org/docs/ProgrammersManual.html
- https://llvm.org/docs/SourceLevelDebugging.html#:~:text=The%20role%20of%20debug%20information,the%20original%20program%20source%20code.
- https://llvm.org/doxygen/InstIterator_8h.html
- https://llvm.org/doxygen/classllvm_1_1Instruction.html