Compiler Design - WS 2015/16, TU Berlin - Lecturer: Dr. Biagio Cosenza

Exercise 3 and 4: LLVM Analysis & Transformation

In this assignment, you should implement two LLVM analysis passes that detect, print and correct all variable uses before initialization. The assignment is divided in two parts: first, you should be able to detect and print (on the LLVM error stream named `errs`) all uses before initialization. Successively, you should be able to fix the input IR by adding an initialization instruction for each uninitialized variables.

You must develop these analysis tools as two LLVM compiler passes implemented in a single `*.cpp` file. A pass template file (`p3.cpp`) is given in order to implement your solution.

LLVM and C++ Basics

LLVM is written in C++. Even if you are not familiar with it, to code an LLVM pass only few language features are necessary. Should you have any doubts, browse all the examples included in the `LLVM.tar.gz` archive, check the online documentation http://llvm.org/docs and in particular the LLVM programming guide http://llvm.org/docs/ProgrammersManual.html .

An LLVM pass is a derived class (subclass) of the `Pass` class, which implements the analysis by overriding virtual methods inherited from `Pass`. The archive code has many examples, including BasicBlock, Function, Loop, Module, Region as well as SCC passes. For example, you can implement a Function pass iterating all functions in the input code by implementing a class that derives (extends) the base class `FunctionPass` and override the virtual function `bool runOnFunction(Function &F)`. A pass implementation should be registered to the pass manager with a `RegisterPass<MyPass>` template function (see http://llvm.org/docs/WritingAnLLVMPass.html).

LLVM uses C++ reference (e.g., `Function &F)`, which is used like a normal value but conceptually works as a pointer (https://www.dgp.toronto.edu/~patrick/csc418/wi2004/notes/PointersVsRef.pdf).

C++ provides a Standard Template Library (STL) offering a number of useful containers and algorithms, such as list, vector and map (see https://www.sgi.com/tech/stl/stl_introduction.html). LLVM adds to STL containers a number of *ad hoc* containers such as `SmallSet` and `SetVector`. Maps are particularly useful to store elements associated to a key element (e.g., you can store in a struct/class info referring to a basic block) http://www.cplusplus.com/reference/map/map ; LLVM provides specific implementations for `StringMap`, `IndexedMap`, `DenseMap`, `ValueMap`, and `IntervalMap`.

LLVM instructions all extend the base class `Instruction`. The GEP instruction is somewhat similar to C array indexing and field selection (actually it is more complicated, but you do not need to know much more for the scope of this assignment); see the details in http://llvm.org/docs/GetElementPtr.html. It is useful to check LLVM class hierarchy and use C++'s `dynamic_cast<>` to check whether a pointer/reference is a valid subclass class pointer/reference of another class. For example, dynamic cast can be used to check whether an `Instruction` is an `AllocaInst`.

To print a string you should use the predefined LLVM raw error stream:
`errs() << "This is a message on LLVM error stream" << end;`

Part 1: Analysis Pass [Read only]

You must develop an LLVM pass that detects and prints possible uses before initialization, therefore it should not change the input IR.

Your code must run on the IR generated by Clang and **should not** apply any other LLVM passes. You are not allowed to use the PassManager (expect for the pass registration, see below). As a consequence, the IR you will work on **is not** in SSA form (i.e., no PHI nodes).

Review the lectures on dataflow analysis and IR to understand required analysis concepts, in particular:

- Domain type
- Direction (backward or forward)
- Transfer function
- Boundary
- Meet operator
- Equations
- Initialization

As in the provided template file, you should register your pass with the following code:
`static RegisterPass<DefinitionPass> X("def-pass", "Reaching definitions pass");`

where `DefinitionPass` is the name of the class implementing the analysis pass, and "`def-pass`" is the command line argument we will use later to activate the pass from the optimizer (`opt`).

Part 2: Fixing Pass [Modify IR]

The second pass will fix the code by adding an initialization instruction for each non-initialized variable. The initialization value is

- 10 for integer
- 20.0 for float
- 30.0 for double

to be used whenever a variable is used before initialization. Therefore, the second pass will actually change the IR.

To check the type of a value you can use `Type::isIntegerTy()`, `Type::isFloatTy()` or `Type::isDoubleTy()`. To create a new store instruction in the IR, you should create a `StoreInst` object (e.g., `new StoreInst(…)`); thus, you can use the specific constructor parameter or the method `insertAfter()` to add it in the right place.

Limitations

For this assignment, you can only use the IR produced by Clang, as-is. You cannot use other LLVM passes or invoke the LLVM pass manager to call other external code.

As the required pass is machine independent, it will be invoked by `opt` (as we have seen in the other examples). You are not allowed to call LLVM's internal analysis such as dominator tree and live intervals, but you can implement your own within the submitted code, if needed.

In some cases, a constant propagation pass may further improve the analysis by skipping some paths in the CFG, but this **is not** required in this assignment (and the given expected output does not implement it).

Evaluation

For the first pass, the **error stream** of your LLVM def pass (for each input code) should closely match the one provided by our implementation (`test*.def`), which prints all variables that are not initialized. As the order of the variable may be different, we compare the sorted outputs (e.g., using `sort` and `diff -f`).

For the second pass, instead, the **output stream** of the *transformed* code (for each input code, after applying the fix pass) executed with the LLVM JIT (i.e., the command `lli`) will be compared with our implementation (`test*.out`).

You will have all the expected outputs (`*.out`), but only part (5 of 10) of the expected `*.def`: your program may give the correct output even if your analysis over-approximate the set of not-initialized variables, and you should try to get the minimal working solution (grade will be higher if you get the minimal one). Test cases will be also checked manually.

Getting started

To start, you have a file named `p34.cpp` with an empty implementation of def and fix pass. Remember to rename `p34.cpp` with your file name, and to apply this change also in the given Makefile (line 3).

Example. Compile and run your pass with the IR generated for `test1.c`

```
> make
> clang -c -emit-llvm test1.c -o test1.bc
> opt -load ./p34.so -def-pass test1.bc -o test1_def.bc
```

you can redirect the standard error to a file in this way:

```
> opt -load ./p34.so -def-pass test1.bc -o test1_def.bc 2> test1.def
```

the same for the fix pass

```
> opt -load ./p34.so -fix-pass test1.bc -o test1_fix.bc 2> test1.fix
```

Example of `lli` output

```
> lli test1_fix.bc
```

you can redirect the standard output to a file in this way:

```
> lli test1_fix.bc > test1.out
```

Submission

- Use the ISIS website
- Submit the project as a single file, extending the given sample file
- File name has to be: `lastname1_lastname2.cpp`, for example: `maradona_klinsmann.cpp`
- First line of the `cpp` file should include first name, surname and student id, for each group participant, e.g.
  `/* Diego Maradona 10, Juergen Klinsmann 18 */`
- The two passes must be registered, respectively, as `def-pass` and `fix-pass`, as shown in the input file