

Exercise 3: LLVM Analysis & Transformation

In this assignment, you should implement two LLVM 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. Subsequently, you should be able to fix the input IR by adding an initialization instruction for each uninitialized variable.

You must develop these analysis tools as two LLVM compiler passes implemented in a single `*.cpp` file. A pass template file (`pass.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, only few language features are necessary to implement an LLVM pass. Should you have any doubts, browse the examples included in the LLVM examples archive, check the online documentation <http://llvm.org/docs> and in particular the LLVM programming guide <http://llvm.org/docs/ProgrammersManual.html>. The programming guide briefly describes important data-structures and shows how common tasks (like inserting an instruction) can be accomplished.

LLVM supports a number of different pass types. For this assignment we will use a function pass, which will be applied to all functions in the input code. A function pass is implemented by extending the base class `FunctionPass` and overriding the virtual function `bool runOnFunction(Function &F)`. The pass implementation can then be registered in the pass manager using a `RegisterPass<MyPass> global` (see <http://llvm.org/docs/WritingAnLLVMPass.html>).

LLVM uses C++ reference (e.g., `Function &F`), which are used like normal values but conceptually behave 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 additionally provides a number of more specialized containers such as `SmallSet` and `SetVector`. Maps are particularly useful to store elements associated with a key element (e.g., you can associate a struct with 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`, which in turn extends `Value`. For this assignment the `AllocaInst`, `LoadInst` and `StoreInst` instructions are particularly important. The LLVM `dyn_cast<>` or C++'s `dynamic_cast<>` can be used to check whether a pointer/reference is a valid subclass pointer/reference of another class. For example, a 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 of variables before initialization. Your code must run on the IR generated by Clang in debug mode (`-g -O0`) and **should not** apply any other LLVM passes. As a consequence, the IR you will work on **is not** in SSA form (i.e., no PHI nodes).

While there are multiple ways to implement this analysis, we strongly recommend an implementation using iterative data-flow analysis. Some suggestions on how to approach the problem:

1. Solve the problem on paper. Determine what the direction of the analysis is, what the gen and kill sets are, how the data-flow equations for in/out look like, which lattice meet to use (union or intersection) and how the sets need to be initialized. Because the analysis is very simple, you may find that either the gen or the kill set is always empty.
2. It is customary to represent the gen/kill/in/out sets as bitvectors. In LLVM the `BitVector` class can be used for this purpose. In order to use a bitvector, you need to establish a mapping between variables (or LLVM `Value`) and bits. As such, the first step is to create a map (e.g. `ValueMap`) from the relevant values to bit offsets. Similarly, you will also need to create a mapping between `BasicBlocks` and a structure containing the gen/kill/in/out sets for each block.
3. Next, compute the gen and kill sets for each block. Then, implement the equations for in/out. The `BitVector` class provides the necessary set operations (union, intersection, complement). To access the successors or predecessors of a basic block (depending on analysis direction) a `pred_iterator` or `succ_iterator` can be used. You can either continue recomputing all blocks until a fixed point is reached, or use a worklist to only recompute blocks that might have changed.
4. Finally, print all uses of potentially uninitialized variables using `getVarName()` and `getDebugLoc()`.

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

The initialization instruction should be inserted at the start of the function (e.g., directly following the `alloca` instructions), not directly before the use. Inserting the initialization instruction directly before use will result in incorrect results if the variable is only uninitialized on some code-paths.

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(...)`); then, you can use either a 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. Do not use other LLVM passes or analysis routines (such as dominator trees, live interval construction, etc.)

In some cases, additional optimizations such as constant propagation or jump threading could 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).

For this assignment you only need to consider instructions of type `AllocInst`, `LoadInst` and `StoreInst`. It is not necessary to take more complicated situations (such as taking the address of a local variable using `&var` and indirectly initializing it) into account.

We use LLVM 3.5 for evaluation. If you use a different LLVM version for development, make sure your code works with LLVM 3.5 prior to submission.

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 at which points in the program a variable may be used prior to initialization (an uninitialized variable can also be used twice in one line, in which case you should also print two messages).

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*.fix`).

You can use the provided `./test.sh` script to check whether your results are correct.

Your C++ file should also include a brief explanation of the algorithm you used to solve this assignment at the top.

Getting started

To start, you have a file named `pass.cpp` with an empty implementation of the def and fix passes. Remember to rename `pass.cpp` with your file name, and to apply this change also in the given `Makefile`. The `./test.sh` script checks your pass against the expected result. The following files are produced:

- `testN.c`: The tested C file.
- `testN.bc`: LLVM bitcode file as generated by Clang.
- `testN.ll`: Disassembled LLVM IR of the bitcode file.
- `testN.def`: The expected output from the definition pass (potentially uninitialized variables).
- `testN.def.out`: The output your definition pass produced.
- `testN.def.diff`: Difference between your output and the expected output.
- `testN.fix`: The expected output of the fixed code (after running the fixing pass).
- `testN.fix.bc`: The bitcode produced by your fixing pass.
- `testN.fix.out`: The output your fixed code produces.
- `testN.fix.diff`: Difference between your output and the expected output.

You can check the `./test.sh` script to see how the files can be manually generated, and how the necessary utilities (clang, opt, lli, llvm-dis) are invoked.

The `./test.sh` script by default invokes LLVM 3.5 binaries. You can specify a different LLVM version using `LLVM_VERSION=4.0 ./test.sh`, for example. If you are switching between different LLVM versions, make sure to always run `make clean` in between, otherwise your code will likely crash.

Note that we will evaluate your code on LLVM 3.5. If you work with a different LLVM version, please make sure your code also works on LLVM 3.5 when you're done. LLVM 3.5 is available on the Ubuntu 14 lab machines.

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 */`
- A brief explanation of the algorithm you used to solve this assignment should also be included at the top of the file.
- The two passes must be registered, respectively, as `def-pass` and `fix-pass`, as shown in the input file.
- The submission must work on LLVM 3.5.