Compiler Design - WS 2024/25

Exercise 3 and 4: LLVM Analysis & Transformation

In this assignment, you will implement two LLVM analysis passes that detect, print and correct all variable uses before initialization. The assignment is divided in two parts, a code analysis and a code transformation. The analysis should be able to detect and print (on the LLVM error stream named `errs`) all variables that are used before initialization. The transformation 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 (`p34.cpp`) is given in order to implement your solution.

Note, the two assignments can be implemented independently.

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, check the online documentation http://llvm.org/docs and in particular the LLVM programming guide http://llvm.org/docs/ProgrammersManual.html .

An LLVM pass has a `run()` method which takes in a unit of IR (Function, Module, BasicBlock, Loop, Region, SCC) on which it operates, as well as an analysis manager. The `run()` method returns a `PreservedAnalysis`, indicating whether the IR has been changed or preserved. A pass implementation should be registered to the pass manager. For this assignment, we will be using LLVM's *new* pass manager (see https://llvm.org/docs/WritingAnLLVMNewPMPass.html). There are currently two pass managers in LLVM, and you may notice some slight differences in the function signatures for passes, and also in the way passes are registered and invoked.

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" << "\n";`

If you are installing clang and LLVM on your own machine, it is recommended that you use any version that is reasonably up-to-date. You may find it easiest to install LLVM from a package manager, using a recent release build. However, you can also build LLVM from straight from source (see https://llvm.org/docs/GettingStarted.html).

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. An uninitialized use is *possible* if there exists at least one path through the control flow, on which an uninitialized use can occur.

Your code must run on the IR generated by Clang and **should not** apply any other LLVM passes. You should not be using the PassManager expect for the pass registration. As such, you should be working on raw, unoptimized code that has not been modified by other passes.

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

The pass registration is done for you near the end of the template file. You do not need to modify the registration, nor the return value of the pass. With this pass registration, "`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. You could also use an `IRBuilder` to help instantiate the instructions you need.

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-approximates 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 – this is for part 1 only). Testing will be conducted in a fully-automated manner. Your solution program will be tested on a variety of different control flows.

The specific implementation of `test10.c` is a fun (and doable) challenge to support in your analysis pass, and you are invited to have a go at it. However, it opens the door to a world of pointer aliasing, pointer arithmetic (i.e., doing maths with memory addresses), etc, in which it quickly becomes very hard and frequently just impossible for compilers to reason about memory accesses. Therefore, `test10.c` will be considered optional (unmarked) for the first pass (assignment 3), but mandatory (marked) for the second pass (assignment 4). You also do not have to support uninitialized global variables.

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. There are several things to compile:

Building p34.cpp

The cpp file contains the function passes, which you will need to analyse the test programs. It can be build into a `p34.so` file by simply typing:

```
> make
```

Compiling test programs to LLVM IR

To work with the test programs, compile them to LLVM IR by typing:

```
> clang -c -emit-llvm -fno-discard-value-names test1.c
```

This outputs a `test1.bc` file, which contains LLVM bitcode and is not human-readable. You can add `-S` to the above command to get a human-readable `.ll` file, or run `llvm-dis test1.bc` which also produces a `.ll` file. The `-fno-discard-value-names` flag ensures that LLVM retains the names of variables. Try omitting it, and you will see numbers instead of variable names.

Running the def-pass
To run the passes, invoke LLVM's optimiser, passing the compiled pass implementation, the name of the pass you want to run, and an IR file to analyse:

```
> opt -load-pass-plugin ./p34.so --passes=def-pass test1.ll -o test1.bc
```

You may also wish to redirect output, which you can do by appending `2> test1.def` to the above call.

Running the fix-pass
As the fix-pass modifies the IR, we need to save it to a new file. Otherwise, it is invoked in the same way as before:

```
> opt -load-pass-plugin ./p34.so --passes=fix-pass test1.ll -o test1_fix.bc
```

We save the result to a `.bc` rather than a human-readable `.ll` file in order to invoke LLVM's interpreter on this file:

```
> lli test1_fix.bc
```

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

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

Submission

- Before submitting, check that your code compiles and produces the output you expect
- Please rename your `p34.cpp` to `group12.cpp` (substitute your group number) and submit your final file via the ISIS website
- Submit the project as a single file, extending the given sample file
- In the first line of the `cpp` file please qinclude 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