Compiler Design - WS 2016/17, TU Berlin - Lecturer: Dr. Biagio Cosenza

Exercise 4: LLVM Constant Propagation

In this assignment you will implement an LLVM pass, which performs constant propagation on SSA form. The pass should be able to evaluate constant expressions and detect variables that have the same constant value on all control-flow paths. For the purposes of this assignment, you will only compute the result of the analysis, but not perform IR modifications based on the result.

Algorithm

The implemented algorithm is known as "Sparse Simple Constant Propagation" (SSCP). It is the basic form of the more powerful and more commonly used "Sparse Conditional Constant Propagation" (SCCP) algorithm. The following is a description of the algorithm, which forgoes the underlying lattice-theoretic foundation.

States: During the execution of this algorithm, each SSA variable is in one of three states:

- Undefined: This is the initial state. We don't yet know whether this variable is constant or not. When working on a variable in this state, we will assume that it is constant with a not yet known value.
- Constant C: The variable has constant value C.
- Overdefined: The variable is not constant.

All variables start in the "undefined" state and are successively lowered to the "constant" and "overdefined" states. Note that a "constant" variable may also become "overdefined". As such, the intermediate state of this algorithm may be *incorrect*, only the final result is correct. Furthermore, note that a variable state may never change in the reverse direction, for example an "overdefined" variable may never become "constant" or "undefined" again. (When implementing, you can add assertions to make sure this never happens.)

Worklist: The algorithm operates on a worklist, which contains instructions that still need to be processed (or processed again).[1] Initially the worklist contains all instructions.

While the worklist is not empty, on each iteration one instruction is removed from the worklist and *visited*. The state of the result of the instruction may change because of this. If this happens, all users of the result are placed into the worklist for reevaluation.

When visiting an instruction, we have to distinguish phi-nodes and other instructions.

Visiting Phi-nodes: In this case, we combine states from multiple control-flow paths. Intuitively, the result of a phi-node is a constant if all the inputs are the *same* constant. More specifically:

1. If *any* Phi operand is "overdefined", the result is also "overdefined".

[1] Of course, it is also possible to keep reevaluating *all* instructions until a fixed-point is reached, similarly to what was done in the previous assignment. Using a worklist is simply more efficient. You can implement this either way.

2. Otherwise, if there are two or more Phi operands that are “constant”, but have different constant values, the result is “overdefined”.
3. Otherwise, if there is at least one “constant” Phi operand and all “constant” Phi operands have the same value C, then the result is “constant” with value C.
4. Otherwise, all Phi operands must be “undefined” and the result is “undefined” as well.

Note that in case 3 it is okay if some of the Phi operands are “undefined”. For example, for a Phi node with operands (undefined, constant C, undefined, constant C) the result is “constant C”. In this case, we’re making an (optimistic) assumption that the “undefined” values also have the constant value C. If this assumption is incorrect, it will be corrected later on.

Visiting other instructions: When visiting other instructions, we will try to evaluate the instruction at compile-time. Of course, this is only possible if we know all operands of the instruction. More specifically:

1. If *any* of the operands is “overdefined”, the result is also “overdefined”.
2. Otherwise, if *any* of the operands is “undefined”, the result is also “undefined”.
3. Otherwise, all operands must be “constant”. In this case we try to evaluate the instruction at compile-time. If this is possible, the result is the resulting “constant”. Otherwise, the result is “overdefined”.

For this assignment constant expression evaluation will only be supported for a small number of instruction types. For instructions which are not specially handled, the result should always be set to “overdefined”.

Implementation Notes

- This time we’re working on SSA form (after the `mem2reg` pass runs).
- The provided pass.cpp file already defines a `State` class, which is used to represent the current state of an LLVM `Value`. Additionally, the code for printing the result of the analysis in the required format is also included. Please do not modify the output format.
- Visiting of instructions is handled using an `InstVisitor`. To visit an instruction `I` call `visit(I)` and the appropriate visitor method (e.g. `visitBinaryOperation`) will be called. The pass.cpp file contains stubs for the visitor methods you need to implement. You only need to handle phi nodes, binary operators, comparisons and casts (as well as a fallback for “all other instructions”).
- LLVMs implementation of SSA form does not make a distinction between an instruction and the result it produces. If the LLVM IR disassembly contains something like `%x = add …`, this is just an artifact of how the IR is printed. There is no explicit `%x` “variable”, it’s just a name given to the `add` instruction result. Any instruction using the result of the `add` will simply have a pointer to the `add` instruction itself as an operand.
- While variables (or rather, `Value`s) are usually initialized to the “undefined” state, function arguments must be initialized to the “overdefined” state, as they are clearly non-constant.
- You only need to support propagation of integers and floating point numbers. In particular, it is not necessary to deal with struct types or pointers.

Output and Evaluation

The output of your pass should be the final result of the analysis, printed to the `errs()` stream. For each instruction, it should be printed in which state the result is, i.e. “undefined, “overdefined” or “constant C”. This should also be printed for instructions which don’t have a meaningful result (such as branches). The pass.cpp already contains code to print the analysis result in this format.

The provided archive contains a number of testN.c files and their expected outputs testN.exp. A Makefile for compiling your pass and a ./test.sh to run your pass and compare its output against the expected outputs is also provided.

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 */`
- Base your implementation on the provided pass.cpp.