Machine Project I

CS 526 — Advanced Compiler Construction
Spring Semester 2021

Due: 11:59pm (Urbana, IL Time Zone) Thursday 2/18/2021

Goal

For this project, you will implement a *Scalar Replacement of Aggregates* pass in LLVM, which operates on a single function at a time. *The goal* of this pass is to replace small, fixed-size aggregate objects (e.g., structures or small constant-size arrays) with separate variables corresponding to the fields of the original object. *The primary benefit* of this pass is that it allows global dataflow optimizations to be applied to fields of aggregate objects.

This transformation is often done only as a global (not interprocedural) pass, i.e., for aggregate objects that are provably created and used only within a single function. That is your goal for this project as well.

What You Need To Do

For this project, you can focus on individual structure objects and ignore arrays. You can also ignore the number of fields, i.e., transform all structures that are otherwise legal. You can focus on objects on the stack, i.e., those allocated using the `alloca` instruction, because transforming globals is an interprocedural transformation and transforming heap-allocated objects can be tricky. *You should recursively transform structures containing structures until no more correct transformations are possible.*

Your goal, therefore, is to eliminate an `alloca` of a structure object and replace it with allocations of individual objects for the fields of the structure. Each new object should be allocated on the stack using an `alloca` instruction (as if it were a local variable in `C`). A high-level algorithm for you to follow is given later in this handout.

Here are some suggestions for passes you should run, before and/or after your pass. Experiment with these and think about which will help increase the effectiveness of your pass.

- `-inline -globaldce`: Inlines small functions, and then eliminates unused ones.
- `-instcombine`: Eliminates unneeded instructions such as redundant casts.
- `-mem2reg`: Eliminates `alloca` instructions for non-aliased scalar variables and puts such variables in virtual registers. *Your pass should enable* `mem2reg` *to put as many scalar fields of structures in virtual registers as possible.*
- `-argpromotion`: Promotes "by reference" arguments to by-val arguments when legal.
- `-sccp`: Sparse conditional constant propagation. This simultaneously propagates constant values and uses them to resolve branches.
- `-dce -simplifycfg -globaldce`: Dead code elimination and branch folding, followed by cleanup of unused functions and global variables.
- `-verify`: Basic syntactic and semantic checks on LLVM bitcode. Run this pass immediately after your pass to check that you have generated "legal" bitcode. This pass is roughly comparable to type-checking the LLVM code you generate, and is *much weaker* than full correctness checking. In particular, it does not tell you anything about the correctness of your transform.

Getting Started with LLVM

LLVM (originally, Low Level Virtual Machine, but the acronym expansion is no longer used) is an open source, production-quality compiler infrastructure that you will use for the projects this semester. You can find all the necessary documentation for LLVM at

http://llvm.org/docs/

In this first MP, part of your job will be to learn the basics of LLVM. See the instructions in the LLVM *Getting Started Guide* for how to download and build LLVM. Follow those instructions to download the source code for LLVM and compile it for your favorite Linux, MacOS or other Unix machine. It is possible to use Windows, but I am largely unfamiliar with it and won't be able to help.

You can use the source code for version 8.0.1 from the Downloads page. The Downloads page provides several pre-compiled versions of the Clang C/C++ front-end, which should work with either version of source (but potentially not with older LLVM revisions). For extra compiler fun, you're also welcome to download and build Clang yourself.

Regardless of where you get the LLVM source, **DO NOT READ THE FOLLOWING SOURCE FILES**:

1. `${LLVM}/lib/Transforms/Scalar/ScalarReplAggregates.cpp`
2. `${LLVM}/lib/Transforms/Utils/PromoteMemoryToRegister.cpp`

${LLVM} is the root of the LLVM tree. The reason for the first file is obvious. The second file is forbidden because it implements the function, `isAllocaPromotable(const AllocaInst*)`, which you must implement yourself.

Recall, the goal of the project is to get you up to speed with LLVM development. A part of this experience is independently figuring out the details of LLVM implementation/data structures and connecting the pieces together. Note that for the second project (which contributes much more to the grade) you will have no hints, so think about this project as an example of what expects you later on.

Use the following LLVM manuals to learn more about the LLVM system:

Downloading and building LLVM:	*http://llvm.org/docs/GettingStarted.html#tutorial*
LLVM command line tools	*http://llvm.org/docs/CommandGuide/*
LLVM IR Reference Manual	*http://llvm.org/docs/LangRef.html*
LLVM Programmer's Manua*l*	*http://llvm.org/docs/ProgrammersManual.html*
Writing an LLVM Pass	*http://llvm.org/docs/WritingAnLLVMPass.html*

Read these manuals very selectively, or you will spend far too much time on them!

High-level Algorithm and Requirements

The top-level function for your pass should iterate the following two steps until no more changes happen:

- Promote some scalar allocas to virtual registers (equivalent to one pass of `mem2reg`).
- Replace some allocas with allocas of the individual fields (i.e., scalar-expand the original allocas).

The mem2reg step promotes some scalar memory locations (e.g., an integer or a pointer) to a register, which can allow more allocas to be scalar expanded. Expanding a structure containing a pointer, followed by promoting the pointer to a virtual register, means that assignments to that pointer now become assignments to SSA virtual registers instead of stores into memory. If the store to this pointer was blocking another structure alloca from being scalar expanded, then these two steps will unblock that expansion. This means that the above two steps must be iterated to find as many expansions as possible.

Important: Construct simple C examples where this behavior happens. We **will** *use such examples in the evaluation. Your examples should include cases that need different numbers of iterations of the above two steps.*

Here are the major requirements for your project, and some tips about useful pieces you can reuse.

- The function `PromoteMemToReg(const std::vector<AllocaInst*> &Allocas, DominatorTree &DT, AliasSetTracker *AST = 0)` invokes the `mem2reg` pass functionality directly. You can call it within your pass to satisfy the first of the two steps, above. See `${LLVM}/include/llvm/Transforms/Utils/PromoteMemToReg.h` for a description of this function.

- One unstated requirement of the `PromoteMemToReg` function is that all the `AllocaInst` instructions in the `Allocas` vector must be promotable, i.e., must satisfy `isAllocaPromotable(const AllocaInst)`. You **MUST NOT** use the existing ("official") version of this function to choose which allocas to promote: **you must implement a sufficient version yourself**, using the following requirements:

  An object allocated using an `alloca` *instruction is promotable to live in a register if the* `alloca` *satisfies all these requirements:*

  (P1) The `alloca` is a "first-class" type, which you can approximate conservatively with
  `isFPOrFPVectorTy() || isIntOrIntVectorTy() || isPtrOrPtrVectorTy`.

  (P2) The `alloca` is only used in a load or store instruction and the instruction satisfies `!isVolatile()`.

  Technically, the use kind (U2) below is also permissible, but the LLVM version does not allow this, so the assertion of `isAllocaPromotable(const AllocaInst)` will fail if you try to permit it.

  Conversely, your code does not have to cover all possible LLVM cases: the above cases capture the vast majority of likely cases and should be sufficient for your testing with real programs (and will be sufficient for mine). If you wish, you may use the official version only in an assertion to verify that your version never labels an alloca as promotable if the official version does not.

  The rest of this section describes the algorithm for the second step above, i.e., the scalar-replacement-of-aggregates step.

- You only need to consider `alloca` instructions that allocate an object of a structure type.

- An `alloca` instruction can be eliminated if the resulting pointer *ptr* is used only in these two ways:

  Q. This list is not exhaustive: can you think of other conditions that are also safe?

  (U1) In a `getelementptr` instruction that satisfies both these conditions:
  - ∗ It is of the form: `getelementptr ptr, 0, constant[, ... constant]`.
  - ∗ The result of the `getelementptr` is only used in instructions of type U1 or U2, or as the pointer argument of a `load` or `store` instruction, i.e., the pointer stored into (not the value being stored).

  (U2) In a `'eq'` or `'ne'` comparison instruction, where the other operand is the NULL pointer value.
  Q. Why is the original pointer not needed in this case?

- In order to eliminate an instruction M, it should be replaced with separate `alloca` instructions, one for each field of the original object. These `alloca` operations should be placed at the entry to the current function.

- Each use of the pointer returned by M must be replaced appropriately. You have to figure out how each of the two kinds of uses listed above should be replaced.

  Note: Remember the principle of Separation of Concerns. *Make the minimal changes needed and let later passes like* `instcombine`, `dce` *and* `deadtypeelim` *do the rest.*

- Because there can be structures nested inside structures, a single scalar-replacement step of your algorithm must iterate until no more structure allocations can be eliminated (in addition to the outer iteration with `mem2reg`).

  Important: For efficiency, don't simply repeat your entire algorithm until nothing changes. Instead, use a worklist containing suitable items and repeat until the worklist is empty.

- It is trivial to create a "correct" transformation by replacing *no* allocations! For this project, your code must count two metrics:

  `NumReplaced` = The number of aggregate allocas broken up.

  `NumPromoted` = The number of scalar allocas promoted to registers.

  You can use the LLVM `STATISTIC` mechanism to compute these metrics.

Implementation Guidelines

Please find a skeleton pass in the file `ScalarReplAggregates-skeleton.cpp` on the course web page. Download it and rename it to `ScalarReplAggregates-$USER.cpp`, where $USER is your campus netid; this name is important because this file is what you must hand in. You will need to submit *only* this file. You will not need external libraries. For helper data structures (vectors, lists, etc.) see the C++ standard library or the specialized versions provided by LLVM.

Follow the instructions in the document, *Writing an LLVM Pass*, to compile, link and run this pass using the `opt` command line tool. Make sure you can see the new pass option when you run 'opt –help'. If you have any difficulties compiling, make sure your LLVM environment is set up as described earlier in this handout.

Think carefully about how your code should be organized. Try to factor out the code into classes and/or functions that capture key functionality. Make the high-level code (the code that drives the overall algorithm) short and easy to understand. Document important parts of the code, including major functions, key assumptions, design choices, etc.

Testing Your Pass

You are responsible for writing test cases to test your pass. In fact, the best way to start this project is to write down a number of test cases that capture as many different promotable structures as possible, both legal and illegal. *The goal is not merely to test your code thoroughly.* Rather, the more different variations you can think of, the better you will understand exactly what your pass must accomplish, and the easier you will find it to organize the cases your pass must handle. This is good, but rarely followed, programming practice, not just for compiler passes, but for any non-trivial programming project.

You should finally test your pass on as many test programs in `llvm/project/llvm-test/` as possible. (See `http://llvm.org/docs/TestingGuide.html` for how to get and use this test suite, which is different from `llvm/test/`).

You can use the Makefile in the test/ directory for your tests. Please be sure to test your code with the sequence of passes given in the Makefile. **For grading we can use any sequence of passes, so please test your code thoroughly.**

Note that `opt` runs the `-verify` pass at the end, so simply running `opt -scalarreplace-netid` (or any sequence of passes with your pass as the last one) will automatically check your output bitcode for consistency.

Bonus Points

Support scalar replacement of small C-style arrays. Consider that the array is small if it has five elements or less. As part of the analysis, make sure that the arrays are local to the function. These tasks will account to the 10% extra points for the grade.

What to turn in

When your code is done, **e-mail me a tar.gz archive named cs526-⟨netid⟩.tar.gz** with the following content:

1. the subdirectory `pass/` should contain only your file `ScalarReplacement-`⟨*netid*⟩`.cpp`.
2. the subdirectory `tests/` should contain your test cases. The tests should be able to run automatically and check that the framework is producing the correct results. As a motivation for how to design the test cases, see the LLVM's capabilities for testing its internal passes in the distribution.

I will run the pass on our battery of hidden tests and will further assess the quality of your work by studying your tests and testing infrastructure. **The overall grade of this homework** will be the combination of testing your pass on our hidden tests and the quality of your testing infrastructure.