Assignment 2: Program Optimizations with LLVM

EECE 571P — Program Analysis and Optimizations
Winter Term 2015

Due: March 6th, 11:59 PM

Goal

For this project, you will implement a Scalar Replacement of Aggregates pass in LLVM, operating on one function at a time. The normal 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.

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 also ignore structure objects that are allocated on heap via functional calls like `malloc()`. *You should recursively transform structures containing structures until no more correct transformations are possible.*

In LLVM, aggregate objects on stack (other than globals) can only be created via `alloca` instruction. 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 such 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 in the next section.

Here are some suggestions for passes you should run, before or after your pass (the test Makefile provided to you invokes many passes in an appropriate order; see the section on **Testing Your Pass** for more information):

- `-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.*
- `-sccp`: Sparse conditional constant propagation. This simultaneously propagates constant values and uses them to resolve branches.
- `-dce -simplifycfg -deadtypeelim`: Dead code elimination and branch folding, followed by cleanup of unused function declarations and types.

High-level Algorithm

- The top-level function for your pass iterates 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 or mallocs with allocas of the individual fields.

  The mem2reg pass promotes some scalar memory locations (e.g., a pointer) to a register, which can allow more allocas/mallocs to be promoted.

  Suggestion: Construct simple C examples where this can happen.

We have given you both the top-level function to perform the above iteration, and a function that invokes the `mem2reg` pass correctly. You must only implement the second step above, i.e., the scalar-replacement-of-aggregates step. The rest of this section describes the algorithm for this step alone.

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

  Note that you can also consider function calls to the `malloc` *function but the return type will not be a structure type and so you will need to follow uses of the returned pointer value to check if it is a structure type. Doing so is optional but encouraged.*

- A `alloca` instruction can be eliminated if the resulting pointer *ptr* is used *only in the following way*:

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

- 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 three 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, your algorithm must iterate until no more structure allocations can be eliminated. (Note that this iteration is within the scalar-replacement step itself, i.e., the second step of the outer iteration.)

  Important: Don't simply repeat your entire algorithm until nothing changes. Instead, use a worklist containing suitable things 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.

  These are already declared in the skeleton code, using the LLVM `STATISTIC` mechanism. Moreover, the latter (`NumPromoted`) is already computed; use it as a model for the former.

Implementation Guidelines

1. Download the skeleton code on Piazza and extract it somewhere. Copy the folder `ScalarReplAggregates-skeleton` to `$LLVMSRC/lib/Transforms`[1] . Add the folder to CMake list.
2. Add code to the `ScalarReplAggregates-skeleton.cpp` file. This is the only file you will be handing in.
3. Go to `$LLVMDST` and run `make` to create a dynamically loadable shared library in `$LLVMDST/lib/`. On Linux, it will be called libAssign2.so. You can try it out with the following command:

   ```
   opt -load $LLVMDST/lib/libAssign2.so -help
   ```

   You should see your pass (`scalarrepl-assn2`) listed along with other optimization passes.

[1] $LLVMSRC means the root directory of the LLVM sources.

4. 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.

5. Document important parts of the code, including major functions, key assumptions, design choices, etc.

   Important: Write comments as you write code, instead of planning to add them later.

6. Test your pass.

Testing Your Pass

You are responsible for writing test cases to test your pass. You can make changes to the Makefile in the `test/` directory and use it for your tests. The test includes two parts. The first part is that your pass is able to convert certain structure objects to scalars and promote them. The second part is that the executables generated from the unoptimized IR (corresponding to the target `old` in the Makefile) and the optimized IR (corresponding to the target `new` in the Makefile) return the same value. You are encouraged to test the pass on programs that are more complicated than the example given in the `test/` directory.

Note that `opt` runs a verifier pass at the end, so simply running `opt -scalarrepl-assn2` (or any sequence of passes with your pass as the last one) will automatically check your output bytecode for consistency. This is no more than a syntactic and type check; it obviously does not tell you much about the correctness of your transform.

What to turn in

When your code is done, just e-mail a copy of your file `ScalarReplAggregates-skeleton.cpp` as an attachment to the TA with "EECE571 Assignment2" in the subject: gpli@ece.ubc.ca.

Evaluation Criteria

The assignment counts for 15 points. Your coding style counts for 5 points. You will get 5 points out of the left 10 points if your pass works for simple structure objects (only containing built-in-type elements), and you will get all 10 points if your pass also works for nested structure objects.