Dr. Annette Bieniusa
M.Sc. Peter Zeller

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Lab 6: Compiler and Language Processing Tools (WS 2018)

Deadline: 08.02.2018, 14:00

This is an optional exercise sheet. You can get additional points here, but in total you only have to get 50% of the points from sheet 1 to 5. Therefore, if you have more than 32 points in total you are admitted to the final exam.

1 SSA code (10 points)

The optimizations on this exercise sheet work best, when the program has been converted to SSA form before. Therefore in part 1 of this sheet you have to write a transformation pass that converts LLVM code to SSA form. If you want to skip this part, we can send you our code for this transformation.

So far, you translated local variables using the `alloca` instruction to allocate space on the stack. Then you used `store` and `load` instructions to access the variables.

To make optimizations easier, we now want to get rid of these `alloca` instructions, and use temporary variables instead. As each temporary variable can only be defined once, this means that removing `alloca` instructions will transform the code into SSA form.

The LLVM tools already contain a tool to remove the `alloca` instructions, which you can invoke with the following command:

```
opt -mem2reg -S test.ll > test2.ll
```

You can use this tool to see one possible outcome of the transformation and compare the placement of phi-nodes with your own solution.

There are different approaches to convert code into SSA form. We suggest that you use one of the following approaches:

1. The paper "Simple and efficient construction of static single assignment form."[1] describes an algorithm which is based on the idea of *Global Value Numbering*.
2. Many algorithms for converting to SSA form (including the `mem2reg` tool from LLVM) are based on calculating *dominance frontiers*. This is also the approach explained in the lecture on SSA form. You can find a similar description of the algorithm in chapter 19 of the book "Modern Compiler Implementation in Java" by A. Appel.

You only have to look for `alloca` instructions in the entry block of the function. Being in the entry block guarantees that the `alloca` is only executed once, which makes analysis simpler.

[1]Braun, Matthias, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. "Simple and efficient construction of static single assignment form." In Compiler Construction, pp. 102-122. Springer Berlin Heidelberg, 2013. `https://pp.info.uni-karlsruhe.de/uploads/publikationen/braun13cc.pdf`

Furthermore, you only have to consider the `alloca` instructions, where the address is only used in direct loads and stores. If the address of the stack object is passed to a function, or if any pointer arithmetic is involved, the `alloca` should not be considered for the transformation.

2 Dead code elimination (4 points)

Write an optimization, which removes all useless assignments. An assignment is useless, if the assignment has no side-effect and its left hand side (the `TemporaryVar`) is never used afterwards (see: Liveness analysis).

Example:

```
%x = add i32 1, 0
%y = add i32 2, 0
%z = add i32 3, 0
%a = add i32 %x, %z
%b = add i32 %x, %y
call void @print(i32 %a)
ret i32 0
```

$\Rightarrow$

```
%x = add i32 1, 0
%z = add i32 3, 0
%a = add i32 %x, %z
call void @print(i32 %a)
ret i32 0
```

3 Constant propagation and constant folding (6 points)

Write an optimization, which evaluates all `BinaryOperations`, which use constant Integer or Boolean values. When the value of a `TemporaryVar` can be determined by constant propagation and constant folding, all uses of the variable should be replaced with that constant value.

Example:

```
%a = add i32 1, 2
%b = mul i32 %a, 6
%c = sdiv i32 %b, %a
%d = sub i32 %a, %c
%e = add i32 4, %d
call void @print(i32 %e)
ret i32 0
```

$\Rightarrow$

```
%a = add i32 1, 2
%b = mul i32 3, 6
%c = sdiv i32 18, 3
%d = sub i32 3, 6
%e = add i32 4, -3
call void @print(i32 1)
ret i32 0
```

4 Additional Optimizations (X points)

If you want to do this optional exercise to get additional points, please contact us. We will then fix a task and assign it a number of points.

For this exercise you should implement some additional optimizations on our LLVM code, similar to task 1 and 2. An optimization should never change the observable program semantics. The optimization should therefore be well commented and tested (at least 2 tests for each optimizations).

Here is a short list of some optimizations, which could be implemented:

1. Constant conditions (if `Branch` has a constant value as condition, convert to `Jump`)
2. Simplify the CFG
   a) Removes basic blocks with no predecessors.
   b) Merges a basic block into its predecessor if there is only one and the predecessor only has one successor.
   c) Eliminates PHI nodes for basic blocks with a single predecessor.
   d) Eliminates a basic block that only contains an unconditional branch.
3. Function inlining
4. Dead store elimination
5. Redundant load elimination
6. Remove unnecessary bound-checks (e.g. interval analysis)
7. Remove unnecessary null-checks (global analysis)
8. Eliminate virtual method calls (global analysis)

You can also implement other optimizations. Some inspiration can be found in the description of the LLVM optimization passes at `http://llvm.org/docs/Passes.html#transform-passes`.

The list above is roughly sorted by difficulty, note that some of the optimizations can be very difficult to implement correctly. Make sure to comment your code and explain why your transformation is correct.