Implementing the into-SSA Transformation with LLVM Semantics

In LLVM IR, the representation of three-address variables differs from the form commonly introduced in many compiler textbooks. LLVM uses the store, load, and alloca instructions to manage variables. The mem2reg pass is then applied to this IR to remove references to these instructions and convert the code to proper SSA (Static Single Assignment) form. In contrast, many compiler textbooks use a simplified syntax that does not include store, load, and alloca constructs. For example:

$$\begin{aligned} x &= y + z \\ y &= z + 3 \\ x &= y + z \end{aligned} \tag{1}$$

To transform this to SSA form, the variables are given numerical subscripts to denote different versions:

$$\begin{aligned} x_1 &= y_1 + z_1 \\ y_2 &= z_1 + 3 \\ x_2 &= y_2 + z_1 \end{aligned} \tag{2}$$

However, LLVM IR does not use numerical subscripts nor allow variable reassignment. Even before the mem2reg pass, each variable definition must dominate its uses, adhering to LLVM's rules for well-formedness. These differences pose challenges when adapting textbook algorithms to work with LLVM's variable handling. In this article, I discuss my experiences and insights gained while trying to apply traditional algorithms to my own IR using LLVM's approach to variable management.

The introduction of store and load instructions can be compared to definitions and uses in the classic SSA form. That leaves us with the alloca instruction. It is easy to see the alloca instruction declares a variable in the IR. This allows us to determine which variables in the IR can span across basic blocks and which variables are temporaries generated for the purpose of computation. We will see that managing variables with store, load and alloca actually simplifies the bookkeeping required to translate into SSA form.

Recall that in the LLVM IR, alloca instructions appear in the form:

```
%1 = alloca SIZE
```

If we imagine that `alloca` allocates memory for a variable, we can consider `%1` the address returned by `alloca`. For our purposes let us call `%1` the alloca variable or alloca target depending on context. Further recall that `load` and `store` instructions accept an `alloca` variable as an argument along with a temporary whose value we wish to store or define.

```
load %2, %1
store %1, %2
```

The `load` instruction above defines the new variable, `%2`, to contain the value stored in the alloca target, `%1`. The `store` instruction states that the value contained inside the alloca target, `%1`, is now defined to be the value defined by `%2`. SSA requires that we "promote" the `alloca` instruction to a register instead, we can do this by keeping track of the loads and stores for each alloca variable using a hashmap. Our map maps each alloca variable to the current temporary that defines it's value. For every `store` instruction we encounter,

we update the map to point to the new definition. For every `load` instruction that we encounter, we replace all uses of the temporary that load defines with the temporary the alloca target maps to.

Another point of confusion is the placement of ϕ nodes, recall that when determining which blocks to place ϕ nodes one must first determine which blocks create new definitions of variables. After all, the purpose of ϕ nodes is to merge competing definitions into one definition. This happens to be the blocks that contain `store` instructions. But when we insert a ϕ node, it must also be treated as a `store` instruction, it redefines the value stored inside the alloca target. So every time we encounter a ϕ node, we must update our map such that the corresponding alloca variable maps to the ϕ node target variable. This also implies we must know which alloca variable each ϕ node maps to, we can create this map during our ϕ node insertion phase. We can summarize our algorithm as follows:

```
InsertPhiNodes(G):
    let P be a map that maps phi nodes to alloca variables.
    for each basic block B in G
        for each instruction I in B

            if I is a store [alloca A], [source variable V] instruction
                Insert phi node for A for each block in IDF(B)
                P[phi] = A for all phi nodes

Let H be a map that maps each alloca to the variable that holds its current value
Rename(Basic Block B, H):

    for each Phi Instruction P in B:
        Find the alloca instruction, I, that P corresponds to
        Use H to get the current variable, V, for alloca, I.
        Insert V into the operand list of P if V is not already in the operand list
        Update H such that H maps I to target variable of P.

    If basic block B has been visited:
        Return
    Else:
        Mark B as visited

    For each instruction I in B:

        if I is a (store [alloca A], [source variable V]) instruction:
            H[A] = V
        else if I is a (load [target variable V], [alloca A]) instruction:
            Replace all uses of V with H[A]


    For each successor, S, of B:
        Save state of H, H'
        Rename (S, H)
        Restore H back to H'
```

Engineering a Compiler (Cooper & Torczon) suggests to perform a DFS walk of the dominator tree in the Rename phase. However, I found this to be quite inconvenient, DFS of the control flow graph will work just as well provided we allow for revisiting of nodes only for the purposes of updating Phi operands.