

SSABuilder::reset

SSABuilder::reset is responsible for clearing all data associated with the basic blocks in the builder. The member variables mVarDefs, mIncompletePhis, and mSealedBlocks store this information.

We must recognize that these containers do not own the LLVM objects (BasicBlock, Value, PHINode); their lifetimes are managed elsewhere. Therefore, we only need to clean up the objects that the builder itself allocates: the SubMap and SubPHI objects stored as values in mVarDefs and mIncompletePhis.

This is done by iterating over each map and calling delete on the values. After deleting these heap-allocated objects, we call .clear() on each map and the set. Even though we do not own the LLVM objects, calling .clear() ensures that the containers release their internal storage and remove the raw pointers they hold, leaving the data structures in an empty state.

SSABuilder::addBlock

SSABuilder::addBlock is called whenever we need to add a new block to the maps of the builder. To do this we simply allocate SubMap and SubPHI pointers using new for both mVarDefs and mIncompletePhis respectively with the provided block. Lastly if the parameter isSealed is true, we want to call sealBlock.

SSABuilder::writeVariable

SSABuilder::writeVariable is called on a variable assignment to record its latest definition in a specific basic block. It updates mVarDefs[block], which maps variables to their current IR values, by indexing into the SubMap using the variable and storing the new value. This ensures future reads return the most recent assignment, implementing *local value numbering* as described in Section 2.1 of the Braun paper.

SSABuilder::readVariable

SSABuilder::readVariable retrieves the value assigned to a variable in a given basic block. It first looks up the variable in the current block, and if not found, recursively searches predecessor blocks to find the most recent definition.

SSABuilder::readVariableRecursive

SSABuilder::readVariableRecursive is responsible for recursively searching predecessor blocks to find the definition of a variable when it isn't defined locally in the requested block. Its main job is to handle incomplete CFGs and to create PHI nodes when multiple control-flow paths need to be merged.

First, we check if the block is unsealed using mSealedBlocks. This matters because an unsealed block doesn't yet know all of its predecessors, so we cannot build a complete PHI node. In that case, we create an incomplete PHI with zero operands as a placeholder. To meet LLVM's requirement that PHI nodes appear at the top of a block, we insert it before the first instruction using &block->front() if the block has instructions, or at the end if the block is empty.

We store this PHI inside mIncompletePhis. We do not call addPhiOperands yet—that happens later in sealBlock(). This also serves as a termination case to avoid infinite recursion in loops.

If the block has exactly one predecessor, no PHI is needed. We simply recurse into that predecessor and return its value, avoiding unnecessary PHI creation.

For sealed blocks with multiple predecessors, we must create a PHI node to merge incoming values. We begin by creating an empty PHI (zero operands) and inserting it at the beginning of the block, again using &block->front() if non-empty, otherwise appending it. We immediately write this PHI into the current block to break recursion cycles. Then we call addPhiOperands, which recursively gathers the variable from each predecessor and fills in the PHI's operands. readVariable on each predecessor may recurse further.

After any of these cases completes, we store the resulting value in the current block with writeVariable, and then return it.

SSABuilder::addPhiOperands

SSABuilder::addPhiOperands is used by SSABuilder::readVariableRecursive to populate a PHI node with the values coming from each of its predecessor blocks. It iterates over all predecessors of the current block using a pred_iterator, reads the variable's value from each predecessor via readVariable, and then adds each result to the PHI node using addIncoming(). Before returning, the function calls tryRemoveTrivialPhi to eliminate trivial PHI nodes, like cases where the PHI self-references or duplicates a single value.

SSABuilder::tryRemoveTrivialPhi

SSABuilder::tryRemoveTrivialPhi identifies and eliminates trivial PHI nodes or PHI nodes that do not actually merge different values. A PHI is considered trivial if all of its operands are either self-references or the same single value.

The function walks through all incoming operands using getNumIncomingValues() and getIncomingValue(i). For each operand, it skips self-references and skips duplicates of an already discovered candidate value. If it encounters a different value, the PHI is not trivial and the function simply returns it. If all operands are self-references with no meaningful incoming value, the replacement becomes an undefined value created with UndefValue::get().

To remove a trivial PHI, the function first collects all of its users (excluding self-references). It then updates all uses of the PHI by calling replaceAllUsesWith. After that, the PHI is removed from mVarDefs to avoid dangling pointers, and erased from its basic block using eraseFromParent().

Finally, the function recursively calls tryRemoveTrivialPhi on all former users that are themselves PHI nodes using dyn_cast<PHINode>. This is necessary because eliminating one PHI may cause another to become trivial. For example, if phi2 = $\phi(\text{phi1}, \text{phi1})$ and phi1 is replaced with the constant 5, then phi2 becomes $\phi(5, 5)$, which is now trivial and should also be removed.

SSABuilder::sealBlock

SSABuilder::sealBlock marks a block as fully known once all its predecessors have been discovered. It looks up all the incomplete PHI nodes stored in mIncompletePhis[block], calls addPhiOperands on each one to fill in the incoming values from every predecessor, and then adds the block to mSealedBlocks to mark that the block is now sealed and won't need any more PHI completion.

Identifier::readFrom, Identifier::writeTo

Both were trivial, requiring just calls to readVariable and writeVariable respectively.

ScopeTable::emitIR, ASTFunction

Removed the else case and replaced some setAddress calls to writeTo, accordingly by the homework document.

ASTIfStmt, ASTWhileStmt

In ASTIfStmt, after creating thenBlock, elseBlock (if it exists), and endBlock, we call addBlock() on each to register them with the SSABuilder. We seal thenBlock before emitting the then statement, seal elseBlock(if it exists) before emitting the else statement, and seal endBlock after all branches to it are created.

In ASTWhileStmt, after creating condBlock, bodyBlock, and endBlock, we call addBlock() on each. We seal bodyBlock before emitting the loop body, then seal both condBlock and endBlock after the back-edge branch is created, to ensure all predecessors are known before sealing.