`getAnalysisUsage`

In LLVM, each pass can specify its dependencies on other passes by overriding the `getAnalysisUsage` function. This ensures that the pass manager runs the required analyses before executing the current pass, providing the necessary information for optimization.

1. `ConstantBranch`

   Following the homework instructions, we add a call to `Info.addRequired<ConstantOps>`. This is because the Constant Branch Folding optimization should only execute after constant propagation has run on the function being optimized, so that branch conditions that are constants can be identified and folded. `setPreservesCFG` is not specified for this pass since the edges of the control flow graph may be altered by creating unconditional branches (i.e., the entry block no longer has an edge to `%if.else`).

2. `DeadBlocks`

   Following the homework instructions, we add a call to `Info.addRequired<ConstantBranch>`. This is because Constant Branch Folding can leave an unreachable block by creating an unconditional branch.

3. `LICM`

   Following the homework instructions, we add a call to `Info.setPreservesCFG,` `Info.addRequired<DeadBlocks>,` `Info.addRequired<DominatorTreeWrapperPass>,` `Info.addRequired<LoopInfo>`. Since the pass does not modify the CFG, must execute after dead blocks are removed, and uses the built-in dominator tree and loop info for accurate hoisting of loop invariants.

`runOnFunction/runOnLoop`

These functions represent the actual optimization pass code for each optimization, which is only called once per function. Since they are only called once per function, all data must be local to each individual invocation.

1. `ConstantBranch`

Using the existing constant propagation code as an example to follow; we first have to initialize a `changed` boolean to indicate as a return value if we perform any folding. Then we create a `removeSet` to put the branch instructions that we'll remove later on. Next, iterate through each block, each instruction, and check if the instruction is a conditional branch with a constant condition instruction for folding and add the ones that are to the `removeSet`. The last step is to remove the instructions we flagged, if any, this part has to be separate since removing them immediately while iterating through the blocks would invalidate the iterator. While we remove the instructions we must also set the `changed` boolean to true and create a new unconditional branch to the target branch: either the then or else case based on the truthiness of the branch condition. It is also important to notify the removed block/successor and isolate it from the CFG, followed by erasing the old branch instruction.

2. `DeadBlocks`

   To find the dead code blocks we have to perform a DFS on the CFG, starting from the entry block. This is done by using a `visitedSet` and dfs_ext_iterator, which allows us to populate a set while performing DFS. Next we have to loop through all blocks and remove any that are not present in `visitedSet`, the ones that are isolated dead code blocks, by placing them in another set called `unreachableSet` for a similar purpose as `removeSet` in the previous optimization. Now if the `unreachableSet` is not empty then we have to indicate that with the return variable `changed` before we iterate through the set and go through all successors of each dead block, removing the dead block as a predecessor, and erasing the dead block from the containing function.

3. `LICM`

   a. `isSafeToHoistInstr`

      According to the assignment requirements, an instruction is eligible to be hoisted only if it satisfies ALL of the following conditions:

      1. Instruction class restriction:
         The instruction must belong to one of the allowed classes: `BinaryOperator`, *`CastInst`*, `SelectInst`, `GetElementPtrInst`, or `CmpInst`.

      2. Safety check:
         The instruction must be free of side effects, as determined by `isSafeToSpeculativelyExecute`.

3. Loop-invariant operands:
   The instruction must have only loop-invariant operands, as checked by `hasLoopInvariantOperands`.

b. `hoistInst`

To hoist the instruction, we move it into the loop's preheader block. This is done by first obtaining the loop preheader with `getLoopPreheader()`, then retrieving the preheader's terminator using `getTerminator()`. The instruction is then relocated to appear immediately before that terminator using `moveBefore`. Finally, we set `mChanged` to true to record that the loop was modified by the hoisting.

c. `hoistPreOrder`

The `hoistPreOrder` function performs a preorder traversal of the dominator tree and hoists eligible instructions out of the loop. For each basic block dominated by the current node, it iterates through the instructions and checks whether each one is safe to hoist. If so, it moves the instruction to the loop preheader. The iterator is advanced before hoisting to avoid iterator invalidation. After processing the current block, the function recursively visits all child nodes in the dominator tree.

d. `runOnLoop`

Set `mChanged` to false to track whether any optimizations occur. Then we save the current loop in mCurrLoop and retrieve the necessary analysis information using `getAnalysis` for loop structure information and the DominatorTree for understanding block dominance relationships. After initialization, it calls `hoistPreOrder` with the dominator tree node corresponding to the loop header (obtained via `mCurrLoop->getHeader()` and `mDomTree->getNode()`), which begins the recursive traversal to identify and hoist loop-invariant instructions. Finally, it returns `mChanged` to indicate whether the loop was modified.