Enna1 clean up 2 years ago

Mem2Reg

mem2reg 是一个 LLVM transform pass，它用于将 LLVM IR 转换为 SSA 形式的 LLVM IR。具体是什么意思?

LLVM IR 借助 “memory 不是 SSA value” 的特点开了个后门。编译器前端在生成 LLVM IR 时，可以选择不生成真正的 SSA 形式，而是把局部变量生成为 alloca/load/store 形式：

- 用 alloca 指令来”声明“变量，得到一个指向该变量的指针
- 用 store 指令来把值存在变量里
- 用 load 指令来把值读出为 SSA value

LLVM 在 mem2reg 这个 pass 中，会识别出上述这种模式的 alloca，把它提升为 SSA value，在提升为 SSA value时会对应地消除 store 与 load，修改为 SSA 的 def-use/use-def 关系，并且在适当的位置安插 Phi 和 进行变量重命名。

实际上 Clang 生成的就是这样的 alloca/load/store 形式的 LLVM IR：

对于如下函数 foo：

```
int foo(int x, int cond)
{
    if (cond > 0)
        x = 1;
    else
        x = -1;
```

```
    return x;
}
```

`clang -Xclang -disable-O0-optnone -O0 -emit-llvm -S foo.c` 得到的 LLVM IR：

```
define dso_local i32 @foo(i32 %x, i32 %cond) #0 {
entry:
  %x.addr = alloca i32, align 4
  %cond.addr = alloca i32, align 4
  store i32 %x, i32* %x.addr, align 4
  store i32 %cond, i32* %cond.addr, align 4
  %0 = load i32, i32* %cond.addr, align 4
  %cmp = icmp sgt i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  store i32 1, i32* %x.addr, align 4
  br label %if.end

if.else:                                          ; preds = %entry
  store i32 -1, i32* %x.addr, align 4
  br label %if.end

if.end:                                           ; preds = %if.else,
%if.then
  %1 = load i32, i32* %x.addr, align 4
  ret i32 %1
}
```

可以看到，局部变量都是在函数的入口基本块通过 alloca 来“声明”的，并且后续对局部变量的赋值都是通过 store 指令，获取局部变量的值都是通过 load 指令，正是前面所说的 alloca/load/store 形式的 LLVM IR。

`opt -S -mem2reg -o foo.m2r.ll foo.ll` 对上述 LLVM IR 运行 mem2reg pass 后得到的新的 LLVM IR：

```
define dso_local i32 @foo(i32 %x, i32 %cond) #0 {
entry:
  %cmp = icmp sgt i32 %cond, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  br label %if.end

if.else:                                          ; preds = %entry
  br label %if.end
```

```
if.end:                                           ; preds = %if.else,
%if.then
  %x.addr.0 = phi i32 [ 1, %if.then ], [ -1, %if.else ]
  ret i32 %x.addr.0
}
```

可以看到 alloca/load/store 形式的局部变量被提升为了 SSA value，并且在 `if.end` 基本块中还安插了 Phi。

mem2reg pass 的代码实现位于： `llvm-7.0.0.src/lib/Transforms/Utils/Mem2Reg.cpp` `llvm-7.0.0.src/lib/Transforms/Utils/PromoteMemoryToRegister.cpp`

算法描述

下面详细先看一下 mem2reg 的算法描述（转自R大在知乎上的回答，见本文的参考链接）：

> 1. LLVM assumes that all locals are introduced in the entry basic block of a function with an alloca instruction. LLVM also assumes that all allocas appear at the start of the entry block continuously. This assumption could be easily violated by the front-end, but it is a reasonable assumption to make.
> 2. For each alloca seen in step 1, LLVM checks if it is promotable based on the use of this local. It is deemed promotable iff:
>   1. It is not used in a volatile instruction.
>   2. It is loaded or stored directly, i.e, its address is not taken.
> 3. For each local selected for promotion in step 2, a list of blocks which use it, and a list of block which define it are maintained by making a linear sweep over each instruction of each block.
> 4. Some basic optimizations are performed:
>   1. Unused allocas are removed.
>   2. If there is only one defining block for an alloca, all loads which are dominated by the definition are replaced with the value.
>   3. allocas which are read and written only in a block can avoid traversing CFG, and PHI-node insertion by simply inserting each load with the value from nearest store.
> 5. A dominator tree is constructed.
> 6. For each candidate for promotion, points to insert PHI nodes is computed as follows:

    1. A list of blocks which use it without defining it (live-in blocks or upward exposed blocks) are determined with the help of using and defining blocks created in Step 3.
    2. A priority queue keyed on dominator tree level is maintained so that inserted nodes corresponding to defining blocks are handled from the bottom of the dominator tree upwards. This is done by giving each block a level based on its position in the dominator tree.
    3. For each node — root, in the priority queue:
        1. Iterated dominance frontier of a definition is computed by walking all dominator tree children of root, inspecting their CFG edges with targets elsewhere on the dominator tree. Only targets whose level is at most root level are added to the iterated dominance frontier.
        2. PHI-nodes are inserted at the beginning in each block in the iterated dominance frontier computed in the previous step. There will be predecessor number of dummy argument to the PHI function at this point.
7. Once all PHI-nodes are prepared, a rename phase start with a worklist containing just entry block as follows:
    1. A hash table of IncomingVals which is a map from a alloca to its most recent name is created. Most recent name of each alloca is an undef value to start with.
    2. While (worklist != NULL)
        1. Remove block B from worklist and mark B as visited.
        2. For each instruction in B:
            1. If instruction is a load instruction from location L (where L is a promotable candidate) to value V, delete load instruction, replace all uses of V with most recent value of L i.e, IncomingVals[L].
            2. If instruction is a store instruction to location L (where L is a promotable candidate) with value V, delete store instruction, set most recent name of L i.e, IncomingVals[L] = V.
            3. For each PHI-node corresponding to a alloca — L , in each successor of B, fill the corresponding PHI-node argument with most recent name for that location i.e, IncomingVals[L].
        3. Add each unvisited successor of B to worklist.

算法实现

mem2reg pass 对应的类是 `PromoteLegacyPass`

```
struct PromoteLegacyPass : public FunctionPass {
  // Pass identification, replacement for typeid
  static char ID;

  PromoteLegacyPass() : FunctionPass(ID) {
    initializePromoteLegacyPassPass(*PassRegistry::getPassRegistry());
  }

  // runOnFunction - To run this pass, first we calculate the alloca
  // instructions that are safe for promotion, then we promote each one.
  bool runOnFunction(Function &F) override {
    if (skipFunction(F))
      return false;

    DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>
().getDomTree();
    AssumptionCache &AC =
        getAnalysis<AssumptionCacheTracker>().getAssumptionCache(F);
    return promoteMemoryToRegister(F, DT, AC);
  }

  void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<AssumptionCacheTracker>();
    AU.addRequired<DominatorTreeWrapperPass>();
    AU.setPreservesCFG();
  }
};
```

`PromoteLegacyPass` 是一个 `FunctionPass`，函数 `runOnFunction` 的内容很简单，就是对函数 `promoteMemoryToRegister` 的调用。

函数 `promoteMemoryToRegister` 的实现如下：

```
static bool promoteMemoryToRegister(Function &F, DominatorTree &DT,
                                    AssumptionCache &AC) {
  std::vector<AllocaInst *> Allocas;
  BasicBlock &BB = F.getEntryBlock(); // Get the entry node for the
function
  bool Changed = false;

  while (true) {
    Allocas.clear();

    // Find allocas that are safe to promote, by looking at all
instructions in
```

```
    // the entry node
    for (BasicBlock::iterator I = BB.begin(), E = --BB.end(); I != E; ++I)
      if (AllocaInst *AI = dyn_cast<AllocaInst>(I)) // Is it an alloca?
        if (isAllocaPromotable(AI))
          Allocas.push_back(AI);

    if (Allocas.empty())
      break;

    PromoteMemToReg(Allocas, DT, &AC);
    NumPromoted += Allocas.size();
    Changed = true;
  }
  return Changed;
}
```

该函数就是收集入口基本块中的所有 Promotable 的 AllocaInst，然后调用函数 `PromoteMemToReg` 。这里 LLVM 有一个假设：一个函数中所有的 AllocaInst 都是只出现在函数的入口基本块。所以编译器前端在生成 LLVM IR 时应该遵守该假设。

那么什么样的 AllocaInst 是 Promotable？简单来说如果该 AllocaInst 没有被用于 volatile instruction，并且它直接被用于 LoadInst 或 StoreInst（即没有被取过地址），那么就认为该 AllocaInst 是 Promotable。详细的可以看 `isAllocaPromotable` 函数的代码实现。

下面看函数 `PromoteMemToReg` 的实现：

```
void llvm::PromoteMemToReg(ArrayRef<AllocaInst *> Allocas, DominatorTree
&DT,
                           AssumptionCache *AC) {
  // If there is nothing to do, bail out...
  if (Allocas.empty())
    return;

  PromoteMem2Reg(Allocas, DT, AC).run();
}
```

如果没有 Promotable 的 AllocaInst，那么毫无疑问直接返回；否则构造一个结构体 `PromoteMem2Reg` 的对象，然后调用该对象的 `run` 函数，注意 `PromoteMem2Reg(Allocas, DT, AC).run()` 中的 `PromoteMem2Reg` 是一个结构体。

结构体 `PromoteMem2Reg` 的定义比较复杂，这里不给出完整的定义代码了，我们先看一下其构造函数：

```
PromoteMem2Reg(ArrayRef<AllocaInst *> Allocas, DominatorTree &DT,
               AssumptionCache *AC)
    : Allocas(Allocas.begin(), Allocas.end()), DT(DT),
      DIB(*DT.getRoot()->getParent()->getParent(), /*AllowUnresolved*/
false),
      AC(AC), SQ(DT.getRoot()->getParent()->getParent()->getDataLayout(),
                 nullptr, &DT, AC) {}
```

- 成员变量 `std::vector<AllocaInst *> Allocas` ，用于保存正在被 promoted 的 AllocaInst，其初始化为所有的 Promotable 的 AllocaInst
- 成员变量 `DIBuilder DIB` Debug Information Builder 用于在 LLVM IR 中创建调试信息
- 成员变量 `AssumptionCache *AC` ，对该成员变量的注释：A cache of @llvm.assume intrinsics used by SimplifyInstruction
- 成员变量 `const SimplifyQuery SQ` ，SimplifyQuery 用于将 instructions 变为更简的形式，例如：*(“add i32 1, 1” -> “2”)*，*(“and i32 %x, 0” -> “0”)*，*(“and i32 %x, %x” -> “%x”)*
- etc，该结构体还有很多其他的成员变量

函数 `PromoteMem2Reg::run()` 是前面提到的 mem2reg 算法的真正代码实现，该函数共200+行，这里不贴完整的代码了，而是一段一段的分析该函数。

```
void PromoteMem2Reg::run() {
  Function &F = *DT.getRoot()->getParent();
  ... // 略
  for (unsigned AllocaNum = 0; AllocaNum != Allocas.size(); ++AllocaNum) {
    AllocaInst *AI = Allocas[AllocaNum];

    assert(isAllocaPromotable(AI) && "Cannot promote non-promotable
alloca!");
    assert(AI->getParent()->getParent() == &F &&
           "All allocas should be in the same function, which is same as
DF!");

    removeLifetimeIntrinsicUsers(AI);

    if (AI->use_empty()) {
      // If there are no uses of the alloca, just delete it now.
      AI->eraseFromParent();

      // Remove the alloca from the Allocas list, since it has been
processed
      RemoveFromAllocasList(AllocaNum);
      ++NumDeadAlloca;
      continue;
```

```
    }
    ... // 略
  }
  ... // 略
}
```

F 是这些 AllocaInst 所在的函数，AllocaDbgDeclares 用于记录描述 AllocaInst 的 *dbg.declare intrinsic* ，在后续 AllocaInst 被 promoted 之后，就可以将 *dbg.declare intrinsic* 转换为 *dbg.value intrinsic* 。

for 循环依次处理每一个 AllocaInst ，在进行一些 assert 判断之后，调用函数 `removeLifetimeIntrinsicUsers` 在 LLVM IR 中删除该 AllocaInst 的 User 中除了 LoadInst 和 StoreInst 以外的 Instructions。如果某个 AllocaInst 没有 User，那么直接删除该 AllocaInst，并且将该 AllocaInst 从成员变量 `std::vector<AllocaInst *> Allocas` 中删除，因为该 AllocaInst 已经被处理完成。这段代码对应 mem2reg 算法的 4.1 部分。

紧接着是后续的这段代码对应 mem2reg 算法的 4.2 和 4.3 部分。

```
void PromoteMem2Reg::run() {
  AllocaDbgDeclares.resize(Allocas.size());
  AllocaInfo Info;
  ... // 略
  for (unsigned AllocaNum = 0; AllocaNum != Allocas.size(); ++AllocaNum) {
    AllocaInst *AI = Allocas[AllocaNum];

    ... // 略

    // Calculate the set of read and write-locations for each alloca.  This
is
    // analogous to finding the 'uses' and 'definitions' of each variable.
    Info.AnalyzeAlloca(AI);

    // If there is only a single store to this value, replace any loads of
```



```
        // The alloca has been processed, move on.
        RemoveFromAllocasList(AllocaNum);
        ++NumSingleStore;
        continue;
      }
    }

    // If the alloca is only read and written in one basic block, just
perform a
    // linear sweep over the block to eliminate it.
```

```
    if (Info.OnlyUsedInOneBlock &&
        promoteSingleBlockAlloca(AI, Info, LBI, SQ.DL, DT, AC)) {
      // The alloca has been processed, move on.
      RemoveFromAllocasList(AllocaNum);
      continue;
    }
    ... // 略
  }
  ... // 略
}
```

调用 `AllocaInfo::AnalyzeAlloca(AllocaInst **AI*)` （详见 Functionality:AllocaInfo）分析该 AllocaInst 的相关信息。根据 `AllocaInfo::AnalyzeAlloca(AllocaInst **AI*)` 的分析结果，如果对该 AllocaInst 的定值只有一处，那么通过函数 `rewriteSingleStoreAlloca` （详见 Functionality:rewriteSingleStoreAlloca）将所有的被该定值点 (def) 所支配的使用点 (use) 都替换为相应的定值，即如果对该 AllocaInst 的 StoreInst 只有一条，那么将所有被该 StoreInst 支配的用于获取 AllocaInst 的值的 LoadInst 替换为被 store 的值；如果对该 AllocaInst 的 def 和 use 都在同一个基本块内，则调用函数 `promoteSingleBlockAlloca` （详见 Functionality:promoteSingleBlockAlloca）通过线性扫描来消除 AllocaInst / StoreInst/ LoadInst 。

在完成对上述一些特别的情况的处理之后，则通过 IDF (Iterated Dominance Frontier) 和标准的 SSA 构建算法来将 alloca/load/store 形式的 LLVM IR 提升为真正的 SSA 形式的 LLVM IR。

```
void PromoteMem2Reg::run() {
  AllocaDbgDeclares.resize(Allocas.size());
  AllocaInfo Info;
  ... // 略
  for (unsigned AllocaNum = 0; AllocaNum != Allocas.size(); ++AllocaNum) {
    AllocaInst *AI = Allocas[AllocaNum];

    ... // 略

    // If we haven't computed a numbering for the BB's in the function, do
so
    // now.
    if (BBNumbers.empty()) {
      unsigned ID = 0;
      for (auto &BB : F)
        BBNumbers[&BB] = ID++;
    }

    // Remember the dbg.declare intrinsic describing this alloca, if any.
```

```
    if (!Info.DbgDeclares.empty())
      AllocaDbgDeclares[AllocaNum] = Info.DbgDeclares;

    // Keep the reverse mapping of the 'Allocas' array for the rename pass.
    AllocaLookup[Allocas[AllocaNum]] = AllocaNum;

    // At this point, we're committed to promoting the alloca using IDF's,
and
    // the standard SSA construction algorithm.  Determine which blocks
need PHI
    // nodes and see if we can optimize out some work by avoiding insertion
of
    // dead phi nodes.

    // Unique the set of defining blocks for efficient lookup.
    SmallPtrSet<BasicBlock *, 32> DefBlocks;
    DefBlocks.insert(Info.DefiningBlocks.begin(),
Info.DefiningBlocks.end());

    // Determine which blocks the value is live in.  These are blocks which
lead
    // to uses.
    SmallPtrSet<BasicBlock *, 32> LiveInBlocks;
    ComputeLiveInBlocks(AI, Info, DefBlocks, LiveInBlocks);

    // At this point, we're committed to promoting the alloca using IDF's,
and
    // the standard SSA construction algorithm.  Determine which blocks
need phi
    // nodes and see if we can optimize out some work by avoiding insertion
of
    // dead phi nodes.
    IDF.setLiveInBlocks(LiveInBlocks);
    IDF.setDefiningBlocks(DefBlocks);
    SmallVector<BasicBlock *, 32> PHIBlocks;
    IDF.calculate(PHIBlocks);
    if (PHIBlocks.size() > 1)
      llvm::sort(PHIBlocks.begin(), PHIBlocks.end(),
                 [this](BasicBlock *A, BasicBlock *B) {
                   return BBNumbers.lookup(A) < BBNumbers.lookup(B);
                 });

    unsigned CurrentVersion = 0;
    for (BasicBlock *BB : PHIBlocks)
      QueuePhiNode(BB, AllocaNum, CurrentVersion);

  }

  if (Allocas.empty())
```

```
    return; // All of the allocas must have been trivial!

  LBI.clear();

  // Set the incoming values for the basic block to be null values for all
of
  // the alloca's.  We do this in case there is a load of a value that has
not
  // been stored yet.  In this case, it will get this null value.
  RenamePassData::ValVector Values(Allocas.size());
  for (unsigned i = 0, e = Allocas.size(); i != e; ++i)
    Values[i] = UndefValue::get(Allocas[i]->getAllocatedType());

  // When handling debug info, treat all incoming values as if they have
unknown
  // locations until proven otherwise.
  RenamePassData::LocationVector Locations(Allocas.size());

  // Walks all basic blocks in the function performing the SSA rename
algorithm
  // and inserting the phi nodes we marked as necessary
  std::vector<RenamePassData> RenamePassWorkList;
  RenamePassWorkList.emplace_back(&F.front(), nullptr, std::move(Values),
                                  std::move(Locations));
  do {
    RenamePassData RPD = std::move(RenamePassWorkList.back());
    RenamePassWorkList.pop_back();
    // RenamePass may add new worklist entries.
    RenamePass(RPD.BB, RPD.Pred, RPD.Values, RPD.Locations,
RenamePassWorkList);
  } while (!RenamePassWorkList.empty());

  // The renamer uses the Visited set to avoid infinite loops.  Clear it
now.
  Visited.clear();

  ... // 略
}
```

上述代码对 mem2reg 算法的步骤 6 和 步骤 7。

应用变量 `SmallPtrSet<BasicBlock *, 32> DefBlocks` 来存储所有的对某 AllocaInst 定值的基本块，用变量 `SmallPtrSet<BasicBlock *, 32> LiveInBlocks` 来存储函数 `ComputeLiveInBlocks` （详见 Functionality:ComputeLiveInBlocks）的返回结果。通过 IDF 计算出需要插入 Phi 的基本块集合 PHIBlocks，遍历 PHIBlocks 调用函数 `QueuePhiNode` ，创建待更新的 PHINode 。

所有需要插入 PHINode 的位置都已经插入了待更新的 PHINode，然后 worklist 算法调用函数 `RenamePass` （详见 Functionality:RenamePass）对 PHINode 进行更新。RenamePassWorkList 被初始化为首先处理函数 F 的入口基本块，然后从入口基本块开始沿着 CFG 不断迭代处理。

最后就是一些收尾的工作。

```
// Remove the allocas themselves from the function.
for (Instruction *A : Allocas) {
  // If there are any uses of the alloca instructions left, they must be in
  // unreachable basic blocks that were not processed by walking the
dominator
  // tree. Just delete the users now.
  if (!A->use_empty())
    A->replaceAllUsesWith(UndefValue::get(A->getType()));
  A->eraseFromParent();
}

// Remove alloca's dbg.declare instrinsics from the function.
for (auto &Declares : AllocaDbgDeclares)
  for (auto *DII : Declares)
    DII->eraseFromParent();
```

如果经过前面处理之后 在 LLVM IR 中还有 AllocaInst，那么将所有使用该 AllocaInst 的地方替换为 UndefValue。然后将 AllocaInst 从 LLVM IR 中删除。在 LLVM IR 中删掉 AllocaInst 的 *dbg.declare instrinsics*。

```
// Loop over all of the PHI nodes and see if there are any that we can get
// rid of because they merge all of the same incoming values.  This can
// happen due to undef values coming into the PHI nodes.  This process is
// iterative, because eliminating one PHI node can cause others to be
removed.
bool EliminatedAPHI = true;
while (EliminatedAPHI) {
  EliminatedAPHI = false;

  // Iterating over NewPhiNodes is deterministic, so it is safe to try to
  // simplify and RAUW them as we go.  If it was not, we could add uses to
  // the values we replace with in a non-deterministic order, thus creating
  // non-deterministic def->use chains.
  for (DenseMap<std::pair<unsigned, unsigned>, PHINode *>::iterator
            I = NewPhiNodes.begin(),
            E = NewPhiNodes.end();
       I != E;) {
```

```
    PHINode *PN = I->second;

    // If this PHI node merges one value and/or undefs, get the value.
    if (Value *V = SimplifyInstruction(PN, SQ)) {
      PN->replaceAllUsesWith(V);
      PN->eraseFromParent();
      NewPhiNodes.erase(I++);
      EliminatedAPHI = true;
      continue;
    }
    ++I;
  }
}
```

接着对 PHINode 进行一些优化，如果 PHINode 的 IncomingValue 中有 UndefValue，那么通过函数 `SimplifyInstruction` 简化该 PHINode，并相应地将使用该 PHINode 的地方替换为简化后的 Value。

```
// At this point, the renamer has added entries to PHI nodes for all
reachable
// code.  Unfortunately, there may be unreachable blocks which the renamer
// hasn't traversed.  If this is the case, the PHI nodes may not
// have incoming values for all predecessors.  Loop over all PHI nodes we
have
// created, inserting undef values if they are missing any incoming values.
for (DenseMap<std::pair<unsigned, unsigned>, PHINode *>::iterator
         I = NewPhiNodes.begin(),
         E = NewPhiNodes.end();
     I != E; ++I) {
  // We want to do this once per basic block.  As such, only process a
block
  // when we find the PHI that is the first entry in the block.
  PHINode *SomePHI = I->second;
  BasicBlock *BB = SomePHI->getParent();
  if (&BB->front() != SomePHI)
    continue;

  // Only do work here if there the PHI nodes are missing incoming values.
We
  // know that all PHI nodes that were inserted in a block will have the
same
  // number of incoming values, so we can just check any of them.
  if (SomePHI->getNumIncomingValues() == getNumPreds(BB))
    continue;

  // Get the preds for BB.
  SmallVector<BasicBlock *, 16> Preds(pred_begin(BB), pred_end(BB));
```

```
  // Ok, now we know that all of the PHI nodes are missing entries for some
  // basic blocks.  Start by sorting the incoming predecessors for
efficient
  // access.
  llvm::sort(Preds.begin(), Preds.end());

  // Now we loop through all BB's which have entries in SomePHI and remove
  // them from the Preds list.
  for (unsigned i = 0, e = SomePHI->getNumIncomingValues(); i != e; ++i) {
    // Do a log(n) search of the Preds list for the entry we want.
    SmallVectorImpl<BasicBlock *>::iterator EntIt = std::lower_bound(
        Preds.begin(), Preds.end(), SomePHI->getIncomingBlock(i));
    assert(EntIt != Preds.end() && *EntIt == SomePHI->getIncomingBlock(i)
&&
           "PHI node has entry for a block which is not a predecessor!");

    // Remove the entry
    Preds.erase(EntIt);
  }

  // At this point, the blocks left in the preds list must have dummy
  // entries inserted into every PHI nodes for the block.  Update all the
phi
  // nodes in this block that we are inserting (there could be phis before
  // mem2reg runs).
  unsigned NumBadPreds = SomePHI->getNumIncomingValues();
  BasicBlock::iterator BBI = BB->begin();
  while ((SomePHI = dyn_cast<PHINode>(BBI++)) &&
         SomePHI->getNumIncomingValues() == NumBadPreds) {
    Value *UndefVal = UndefValue::get(SomePHI->getType());
    for (BasicBlock *Pred : Preds)
      SomePHI->addIncoming(UndefVal, Pred);
  }
}

NewPhiNodes.clear();
```

经过前面 RenamePass 的处理后基本上 PHINode 都已经更新好了，但是因为 RenamePass 是沿着 CFG 进行处理的，所有可能对于那些 unreachable blocks 即那些沿着 CFG 不可达的基本块，PHINode 的 incoming values 还不完整，对于这种情况，将这些 incoming values 设置为 UndefValue。

Functionality

为了让 LLVM 的 mem2reg 算法实现更加易读，这里将函数 `PromoteMem2Reg::run()` 中用到的一些数据结构和函数单独进行分析。

AllocaInfo

```
struct AllocaInfo {
  SmallVector<BasicBlock *, 32> DefiningBlocks;
  SmallVector<BasicBlock *, 32> UsingBlocks;

  StoreInst *OnlyStore;
  BasicBlock *OnlyBlock;
  bool OnlyUsedInOneBlock;

  Value *AllocaPointerVal;
  TinyPtrVector<DbgInfoIntrinsic *> DbgDeclares;

  void clear() {
    DefiningBlocks.clear();
    UsingBlocks.clear();
    OnlyStore = nullptr;
    OnlyBlock = nullptr;
    OnlyUsedInOneBlock = true;
    AllocaPointerVal = nullptr;
    DbgDeclares.clear();
  }

  /// Scan the uses of the specified alloca, filling in the AllocaInfo used
  /// by the rest of the pass to reason about the uses of this alloca.
  void AnalyzeAlloca(AllocaInst *AI) {
    clear();

    // As we scan the uses of the alloca instruction, keep track of stores,
    // and decide whether all of the loads and stores to the alloca are
within
    // the same basic block.
    for (auto UI = AI->user_begin(), E = AI->user_end(); UI != E;) {
      Instruction *User = cast<Instruction>(*UI++);

      if (StoreInst *SI = dyn_cast<StoreInst>(User)) {
        // Remember the basic blocks which define new values for the alloca
        DefiningBlocks.push_back(SI->getParent());
        AllocaPointerVal = SI->getOperand(0);
        OnlyStore = SI;
      } else {
        LoadInst *LI = cast<LoadInst>(User);
        // Otherwise it must be a load instruction, keep track of variable
        // reads.
        UsingBlocks.push_back(LI->getParent());
        AllocaPointerVal = LI;
      }
```

```
      if (OnlyUsedInOneBlock) {
        if (!OnlyBlock)
          OnlyBlock = User->getParent();
        else if (OnlyBlock != User->getParent())
          OnlyUsedInOneBlock = false;
      }
    }

    DbgDeclares = FindDbgAddrUses(AI);
  }
};
```

`AllocaInfo::AnalyzeAlloca(AllocaInst **AI*)` 函数，记录了给定的一条 AllocaInst 的相关信息， AllocaInfo 的成员变量 DefiningBlocks 记录了所有对 AllocaInst 进行定值 (def) 的基本块；成员变量 UsingBlocks 记录了所有对 AllocaInst 进行使用 (use) 的基本块；成员变量 OnlyUsedInOneBlock 记录了是否所有对该条 AllocaInst 的 def 和 use 都在同一个基本块中，如果是，则将该基本块记录在成员变量 OnlyBlock 中；如果对 AllocaInst 的定值 (def) 即 StoreInst 只有一条，那么该 StoreInst 则存储在成员变量 OnlyStore 中。

LargeBlockInfo

```
/// This assigns and keeps a per-bb relative ordering of load/store
/// instructions in the block that directly load or store an alloca.
///
/// This functionality is important because it avoids scanning large basic
/// blocks multiple times when promoting many allocas in the same block.
class LargeBlockInfo {
  /// For each instruction that we track, keep the index of the
  /// instruction.
  ///
  /// The index starts out as the number of the instruction from the start
of
  /// the block.
  DenseMap<const Instruction *, unsigned> InstNumbers;

public:
  /// This code only looks at accesses to allocas.
  static bool isInterestingInstruction(const Instruction *I) {
    return (isa<LoadInst>(I) && isa<AllocaInst>(I->getOperand(0))) ||
           (isa<StoreInst>(I) && isa<AllocaInst>(I->getOperand(1)));
  }

  /// Get or calculate the index of the specified instruction.
  unsigned getInstructionIndex(const Instruction *I) {
    assert(isInterestingInstruction(I) &&
           "Not a load/store to/from an alloca?");
```

```
    // If we already have this instruction number, return it.
    DenseMap<const Instruction *, unsigned>::iterator It =
InstNumbers.find(I);
    if (It != InstNumbers.end())
      return It->second;

    // Scan the whole block to get the instruction.  This accumulates
    // information for every interesting instruction in the block, in order
to
    // avoid gratuitus rescans.
    const BasicBlock *BB = I->getParent();
    unsigned InstNo = 0;
    for (const Instruction &BBI : *BB)
      if (isInterestingInstruction(&BBI))
        InstNumbers[&BBI] = InstNo++;
    It = InstNumbers.find(I);

    assert(It != InstNumbers.end() && "Didn't insert instruction?");
    return It->second;
  }

  void deleteValue(const Instruction *I) { InstNumbers.erase(I); }

  void clear() { InstNumbers.clear(); }
};
```

LargeBlockInfo 用于记录和获取同一基本块中出现的 LoadInst 和 StoreInst 先后顺序。

rewriteSingleStoreAlloca

```
/// Rewrite as many loads as possible given a single store.
///
/// When there is only a single store, we can use the domtree to trivially
/// replace all of the dominated loads with the stored value. Do so, and
return
/// true if this has successfully promoted the alloca entirely. If this
returns
/// false there were some loads which were not dominated by the single
store
/// and thus must be phi-ed with undef. We fall back to the standard alloca
/// promotion algorithm in that case.
static bool rewriteSingleStoreAlloca(AllocaInst *AI, AllocaInfo &Info,
                                     LargeBlockInfo &LBI, const DataLayout
&DL,
                                     DominatorTree &DT, AssumptionCache
*AC) {
  StoreInst *OnlyStore = Info.OnlyStore;
```

```
  bool StoringGlobalVal = !isa<Instruction>(OnlyStore->getOperand(0));
  BasicBlock *StoreBB = OnlyStore->getParent();
  int StoreIndex = -1;

  // Clear out UsingBlocks.  We will reconstruct it here if needed.
  Info.UsingBlocks.clear();

  for (auto UI = AI->user_begin(), E = AI->user_end(); UI != E;) {
    Instruction *UserInst = cast<Instruction>(*UI++);
    if (!isa<LoadInst>(UserInst)) {
      assert(UserInst == OnlyStore && "Should only have load/stores");
      continue;
    }
    LoadInst *LI = cast<LoadInst>(UserInst);

    // Okay, if we have a load from the alloca, we want to replace it with
the
    // only value stored to the alloca.  We can do this if the value is
    // dominated by the store.  If not, we use the rest of the mem2reg
machinery
    // to insert the phi nodes as needed.
    if (!StoringGlobalVal) { // Non-instructions are always dominated.
      if (LI->getParent() == StoreBB) {
        // If we have a use that is in the same block as the store, compare
the
        // indices of the two instructions to see which one came first.  If
the
        // load came before the store, we can't handle it.
        if (StoreIndex == -1)
          StoreIndex = LBI.getInstructionIndex(OnlyStore);

        if (unsigned(StoreIndex) > LBI.getInstructionIndex(LI)) {
          // Can't handle this load, bail out.
          Info.UsingBlocks.push_back(StoreBB);
          continue;
        }
      } else if (LI->getParent() != StoreBB &&
                 !DT.dominates(StoreBB, LI->getParent())) {
        // If the load and store are in different blocks, use BB dominance
to
        // check their relationships.  If the store doesn't dom the use,
bail
        // out.
        Info.UsingBlocks.push_back(LI->getParent());
        continue;
      }
    }

    // Otherwise, we *can* safely rewrite this load.
```

```
    Value *ReplVal = OnlyStore->getOperand(0);
    // If the replacement value is the load, this must occur in unreachable
    // code.
    if (ReplVal == LI)
      ReplVal = UndefValue::get(LI->getType());

    // If the load was marked as nonnull we don't want to lose
    // that information when we erase this Load. So we preserve
    // it with an assume.
    if (AC && LI->getMetadata(LLVMContext::MD_nonnull) &&
        !isKnownNonZero(ReplVal, DL, 0, AC, LI, &DT))
      addAssumeNonNull(AC, LI);

    LI->replaceAllUsesWith(ReplVal);
    LI->eraseFromParent();
    LBI.deleteValue(LI);
  }

  // Finally, after the scan, check to see if the store is all that is
left.
  if (!Info.UsingBlocks.empty())
    return false; // If not, we'll have to fall back for the remainder.

  // Record debuginfo for the store and remove the declaration's
  // debuginfo.
  for (DbgInfoIntrinsic *DII : Info.DbgDeclares) {
    DIBuilder DIB(*AI->getModule(), /*AllowUnresolved*/ false);
    ConvertDebugDeclareToDebugValue(DII, Info.OnlyStore, DIB);
    DII->eraseFromParent();
    LBI.deleteValue(DII);
  }
  // Remove the (now dead) store and alloca.
  Info.OnlyStore->eraseFromParent();
  LBI.deleteValue(Info.OnlyStore);

  AI->eraseFromParent();
  LBI.deleteValue(AI);
  return true;
}
```

`rewriteSingleStoreAlloca` 该函数的注释写的很清晰。在对于某条 AllocaInst 只有一处定值点 (def) 的情况下会调用该函数，该定值点即为 OnlyStore。因为只有一处定值点，所以可以将所有的被定值点**支配**的使用点 (use) 即 LoadInst 都替换该被定值点定义的值。但是这里有几个特殊情况需要处理：如果 def 和 use 在同一基本块内，那么需要保证 def 在 use 之前，这里就是通过 `LargeBlockInfo::getInstructionIndex()` 来计算 def 和 use 的先后顺序的，如果 use 在 def 之前，那么则该 use 则不能被定值所替换；如果 use 没有被 def 支配，当然该 use 也不能被定值所替换掉。

promoteSingleBlockAlloca

```
/// Many allocas are only used within a single basic block.  If this is the
/// case, avoid traversing the CFG and inserting a lot of potentially
useless
/// PHI nodes by just performing a single linear pass over the basic block
/// using the Alloca.
///
/// If we cannot promote this alloca (because it is read before it is
written),
/// return false.  This is necessary in cases where, due to control flow,
the
/// alloca is undefined only on some control flow paths.  e.g. code like
/// this is correct in LLVM IR:
///  // A is an alloca with no stores so far
///  for (...) {
///    int t = *A;
///    if (!first_iteration)
///      use(t);
///    *A = 42;
///  }
static bool promoteSingleBlockAlloca(AllocaInst *AI, const AllocaInfo
&Info,
                                     LargeBlockInfo &LBI, const DataLayout
&DL,
                                     DominatorTree &DT, AssumptionCache
*AC) {
  // The trickiest case to handle is when we have large blocks. Because of
this,
  // this code is optimized assuming that large blocks happen.  This does
not
  // significantly pessimize the small block case.  This uses
LargeBlockInfo to
  // make it efficient to get the index of various operations in the block.

  // Walk the use-def list of the alloca, getting the locations of all
stores.
  using StoresByIndexTy = SmallVector<std::pair<unsigned, StoreInst *>,
64>;
  StoresByIndexTy StoresByIndex;

  for (User *U : AI->users())
    if (StoreInst *SI = dyn_cast<StoreInst>(U))
      StoresByIndex.push_back(std::make_pair(LBI.getInstructionIndex(SI),
SI));

  // Sort the stores by their index, making it efficient to do a lookup
with a
```

```
  // binary search.
  llvm::sort(StoresByIndex.begin(), StoresByIndex.end(), less_first());

  // Walk all of the loads from this alloca, replacing them with the
nearest
  // store above them, if any.
  for (auto UI = AI->user_begin(), E = AI->user_end(); UI != E;) {
    LoadInst *LI = dyn_cast<LoadInst>(*UI++);
    if (!LI)
      continue;

    unsigned LoadIdx = LBI.getInstructionIndex(LI);

    // Find the nearest store that has a lower index than this load.
    StoresByIndexTy::iterator I = std::lower_bound(
        StoresByIndex.begin(), StoresByIndex.end(),
        std::make_pair(LoadIdx, static_cast<StoreInst *>(nullptr)),
        less_first());
    if (I == StoresByIndex.begin()) {
      if (StoresByIndex.empty())
        // If there are no stores, the load takes the undef value.
        LI->replaceAllUsesWith(UndefValue::get(LI->getType()));
      else
        // There is no store before this load, bail out (load may be
affected
        // by the following stores - see main comment).
        return false;
    } else {
      // Otherwise, there was a store before this load, the load takes its
      // value. Note, if the load was marked as nonnull we don't want to
lose
      // that information when we erase it. So we preserve it with an
assume.
      Value *ReplVal = std::prev(I)->second->getOperand(0);
      if (AC && LI->getMetadata(LLVMContext::MD_nonnull) &&
          !isKnownNonZero(ReplVal, DL, 0, AC, LI, &DT))
        addAssumeNonNull(AC, LI);

      // If the replacement value is the load, this must occur in
unreachable
      // code.
      if (ReplVal == LI)
        ReplVal = UndefValue::get(LI->getType());

      LI->replaceAllUsesWith(ReplVal);
    }

    LI->eraseFromParent();
    LBI.deleteValue(LI);
```

```
  }

  // Remove the (now dead) stores and alloca.
  while (!AI->use_empty()) {
    StoreInst *SI = cast<StoreInst>(AI->user_back());
    // Record debuginfo for the store before removing it.
    for (DbgInfoIntrinsic *DII : Info.DbgDeclares) {
      DIBuilder DIB(*AI->getModule(), /*AllowUnresolved*/ false);
      ConvertDebugDeclareToDebugValue(DII, SI, DIB);
    }
    SI->eraseFromParent();
    LBI.deleteValue(SI);
  }

  AI->eraseFromParent();
  LBI.deleteValue(AI);

  // The alloca's debuginfo can be removed as well.
  for (DbgInfoIntrinsic *DII : Info.DbgDeclares) {
    DII->eraseFromParent();
    LBI.deleteValue(DII);
  }

  ++NumLocalPromoted;
  return true;
}
```

函数 `promoteSingleBlockAlloca` 用于处理 AllocaInst 的 LoadInst 和 StoreInst 只出现在同一基本块中的情况。对于每个 LoadInst 指令寻找在其之前出现的、相邻最近的 StoreInst，将所有通过 LoadInst 获取的值都替换为对应的 Stored value。存在几种特殊情况：如果对于某个 AllocaInst 来说，相应的 StoreInst 集合为空，那么将所有的通过 LoadInst 获取的值都替换为 UndefValue；如果对于某条 LoadInst 来说，没有在其之前出现的、相邻最近的 StoreInst，那么函数 `promoteSingleBlockAlloca` 返回 false，后续通过标准 SSA 构建算法来处理。

ComputeLiveInBlocks

```
/// Determine which blocks the value is live in.
///
/// These are blocks which lead to uses.  Knowing this allows us to avoid
/// inserting PHI nodes into blocks which don't lead to uses (thus, the
/// inserted phi nodes would be dead).
void PromoteMem2Reg::ComputeLiveInBlocks(
    AllocaInst *AI, AllocaInfo &Info,
    const SmallPtrSetImpl<BasicBlock *> &DefBlocks,
    SmallPtrSetImpl<BasicBlock *> &LiveInBlocks) {
  // To determine liveness, we must iterate through the predecessors of
```

```
blocks
  // where the def is live.  Blocks are added to the worklist if we need to
  // check their predecessors.  Start with all the using blocks.
  SmallVector<BasicBlock *, 64>
LiveInBlockWorklist(Info.UsingBlocks.begin(),

                                                  Info.UsingBlocks.end());

  // If any of the using blocks is also a definition block, check to see if
the
  // definition occurs before or after the use.  If it happens before the
use,
  // the value isn't really live-in.
  for (unsigned i = 0, e = LiveInBlockWorklist.size(); i != e; ++i) {
    BasicBlock *BB = LiveInBlockWorklist[i];
    if (!DefBlocks.count(BB))
      continue;

    // Okay, this is a block that both uses and defines the value.  If the
first
    // reference to the alloca is a def (store), then we know it isn't
live-in.
    for (BasicBlock::iterator I = BB->begin();; ++I) {
      if (StoreInst *SI = dyn_cast<StoreInst>(I)) {
        if (SI->getOperand(1) != AI)
          continue;

        // We found a store to the alloca before a load.  The alloca is not
        // actually live-in here.
        LiveInBlockWorklist[i] = LiveInBlockWorklist.back();
        LiveInBlockWorklist.pop_back();
        --i;
        --e;
        break;
      }

      if (LoadInst *LI = dyn_cast<LoadInst>(I)) {
        if (LI->getOperand(0) != AI)
          continue;

        // Okay, we found a load before a store to the alloca.  It is
actually
        // live into this block.
        break;
      }
    }
  }

  // Now that we have a set of blocks where the phi is live-in, recursively
```

```
add
  // their predecessors until we find the full region the value is live.
  while (!LiveInBlockWorklist.empty()) {
    BasicBlock *BB = LiveInBlockWorklist.pop_back_val();

    // The block really is live in here, insert it into the set.  If
already in
    // the set, then it has already been processed.
    if (!LiveInBlocks.insert(BB).second)
      continue;

    // Since the value is live into BB, it is either defined in a
predecessor or
    // live into it to.  Add the preds to the worklist unless they are a
    // defining block.
    for (BasicBlock *P : predecessors(BB)) {
      // The value is not live into a predecessor if it defines the value.
      if (DefBlocks.count(P))
        continue;

      // Otherwise it is, add to the worklist.
      LiveInBlockWorklist.push_back(P);
    }
  }
}
```

函数 `PromoteMem2Reg::ComputeLiveInBlocks` 注释很清晰。LiveInBlockWorklist 被初始化为所有对 AllocaInst 进行使用 (use) 的基本块，如果 LiveInBlockWorklist 存在对 AllocaInst 进行定义 (def) 的基本块，并且在该基本块中对该 AllocaInst 的第一条 StoreInst 出现在对该 AllocaInst 的第一条 LoadInst 之前，那么在 LiveInBlockWorklist 中去掉该基本块。然后以此时的 LiveInBlockWorklist 作为初始集合进行 worklist 算法迭代：对于 LiveInBlockWorklist 中的每个元素，如果其不在 LiveInBlocks 中，则将其添加至 LiveInBlocks，如果其前驱基本块不是对 AllocaInst 进行定义 (def) 的基本块，则将此前驱基本块也添加至 LiveInBlockWorklist，一直迭代至 LiveInBlockWorklist 为空。

QueuePhiNode

```
/// Queue a phi-node to be added to a basic-block for a specific Alloca.
///
/// Returns true if there wasn't already a phi-node for that variable
bool PromoteMem2Reg::QueuePhiNode(BasicBlock *BB, unsigned AllocaNo,
                                  unsigned &Version) {
  // Look up the basic-block in question.
  PHINode *&PN = NewPhiNodes[std::make_pair(BBNumbers[BB], AllocaNo)];

  // If the BB already has a phi node added for the i'th alloca then we're
```

```
done!
  if (PN)
    return false;

  // Create a PhiNode using the dereferenced type... and add the phi-node
to the
  // BasicBlock.
  PN = PHINode::Create(Allocas[AllocaNo]->getAllocatedType(),
getNumPreds(BB),
                       Allocas[AllocaNo]->getName() + "." +
Twine(Version++),
                       &BB->front());
  ++NumPHIInsert;
  PhiToAllocaMap[PN] = AllocaNo;
  return true;
}
```

函数 `QueuePhiNode` 用于在基本块 BB 入口处为第 AllocaNo 条 AllocaInst 创建一个待更新的 PHINode。待更新指的是这里 PHINode 指令的操作数还不完全，需要后续对操作数进行更新（在函数 `PromoteMem2Reg::RenamePass` 中对此处插入的 PHINode 进行更新）。

RenamePass

```
/// Recursively traverse the CFG of the function, renaming loads and
/// stores to the allocas which we are promoting.
///
/// IncomingVals indicates what value each Alloca contains on exit from the
/// predecessor block Pred.
void PromoteMem2Reg::RenamePass(BasicBlock *BB, BasicBlock *Pred,
                                RenamePassData::ValVector &IncomingVals,
                                RenamePassData::LocationVector
&IncomingLocs,
                                std::vector<RenamePassData> &Worklist) {
NextIteration:
  // If we are inserting any phi nodes into this BB, they will already be
in the
  // block.
  if (PHINode *APN = dyn_cast<PHINode>(BB->begin())) {
    // If we have PHI nodes to update, compute the number of edges from
Pred to
    // BB.
    if (PhiToAllocaMap.count(APN)) {
      // We want to be able to distinguish between PHI nodes being inserted
by
      // this invocation of mem2reg from those phi nodes that already
existed in
      // the IR before mem2reg was run.  We determine that APN is being
inserted
```

```
      // because it is missing incoming edges.  All other PHI nodes being
      // inserted by this pass of mem2reg will have the same number of
incoming
      // operands so far.  Remember this count.
      unsigned NewPHINumOperands = APN->getNumOperands();

      unsigned NumEdges = std::count(succ_begin(Pred), succ_end(Pred), BB);
      assert(NumEdges && "Must be at least one edge from Pred to BB!");

      // Add entries for all the phis.
      BasicBlock::iterator PNI = BB->begin();
      do {
        unsigned AllocaNo = PhiToAllocaMap[APN];

        // Update the location of the phi node.
        updateForIncomingValueLocation(APN, IncomingLocs[AllocaNo],
                                       APN->getNumIncomingValues() > 0);

        // Add N incoming values to the PHI node.
        for (unsigned i = 0; i != NumEdges; ++i)
          APN->addIncoming(IncomingVals[AllocaNo], Pred);

        // The currently active variable for this block is now the PHI.
        IncomingVals[AllocaNo] = APN;
        for (DbgInfoIntrinsic *DII : AllocaDbgDeclares[AllocaNo])
          ConvertDebugDeclareToDebugValue(DII, APN, DIB);

        // Get the next phi node.
        ++PNI;
        APN = dyn_cast<PHINode>(PNI);
        if (!APN)
          break;

        // Verify that it is missing entries.  If not, it is not being
inserted
        // by this mem2reg invocation so we want to ignore it.
      } while (APN->getNumOperands() == NewPHINumOperands);
    }
  }

  // Don't revisit blocks.
  if (!Visited.insert(BB).second)
    return;

  for (BasicBlock::iterator II = BB->begin(); !isa<TerminatorInst>(II);) {
    Instruction *I = &*II++; // get the instruction, increment iterator

    if (LoadInst *LI = dyn_cast<LoadInst>(I)) {
      AllocaInst *Src = dyn_cast<AllocaInst>(LI->getPointerOperand());
```

```
      if (!Src)
        continue;

      DenseMap<AllocaInst *, unsigned>::iterator AI =
AllocaLookup.find(Src);
      if (AI == AllocaLookup.end())
        continue;

      Value *V = IncomingVals[AI->second];

      // If the load was marked as nonnull we don't want to lose
      // that information when we erase this Load. So we preserve
      // it with an assume.
      if (AC && LI->getMetadata(LLVMContext::MD_nonnull) &&
          !isKnownNonZero(V, SQ.DL, 0, AC, LI, &DT))
        addAssumeNonNull(AC, LI);

      // Anything using the load now uses the current value.
      LI->replaceAllUsesWith(V);
      BB->getInstList().erase(LI);
    } else if (StoreInst *SI = dyn_cast<StoreInst>(I)) {
      // Delete this instruction and mark the name as the current holder of
the
      // value
      AllocaInst *Dest = dyn_cast<AllocaInst>(SI->getPointerOperand());
      if (!Dest)
        continue;

      DenseMap<AllocaInst *, unsigned>::iterator ai =
AllocaLookup.find(Dest);
      if (ai == AllocaLookup.end())
        continue;

      // what value were we writing?
      unsigned AllocaNo = ai->second;
      IncomingVals[AllocaNo] = SI->getOperand(0);

      // Record debuginfo for the store before removing it.
      IncomingLocs[AllocaNo] = SI->getDebugLoc();
      for (DbgInfoIntrinsic *DII : AllocaDbgDeclares[ai->second])
        ConvertDebugDeclareToDebugValue(DII, SI, DIB);
      BB->getInstList().erase(SI);
    }
  }

  // 'Recurse' to our successors.
  succ_iterator I = succ_begin(BB), E = succ_end(BB);
  if (I == E)
    return;
```

```
  // Keep track of the successors so we don't visit the same successor
twice
  SmallPtrSet<BasicBlock *, 8> VisitedSuccs;

  // Handle the first successor without using the worklist.
  VisitedSuccs.insert(*I);
  Pred = BB;
  BB = *I;
  ++I;

  for (; I != E; ++I)
    if (VisitedSuccs.insert(*I).second)
      Worklist.emplace_back(*I, Pred, IncomingVals, IncomingLocs);

  goto NextIteration;
}
```

该函数的代码主要对应 mem2reg 算法中的步骤7，不再赘述。

While (worklist != NULL)

1. Remove block B from worklist and mark B as visited.
2. For each instruction in B:
    1. If instruction is a load instruction from location L (where L is a promotable candidate) to value V, delete load instruction, replace all uses of V with most recent value of L i.e, IncomingVals[L].
    2. If instruction is a store instruction to location L (where L is a promotable candidate) with value V, delete store instruction, set most recent name of L i.e, IncomingVals[L] = V.
    3. For each PHI-node corresponding to a alloca — L , in each successor of B, fill the corresponding PHI-node argument with most recent name for that location i.e, IncomingVals[L].
3. Add each unvisited successor of B to worklist.

参考链接

https://www.zhihu.com/question/41999500/answer/93243408