Aggressive Dead Code Elimination

Aggressive Dead Code Elimination (以下简称 ADCE) 是一个 LLVM transform pass，用于消除冗余代码。该 ADCE 本质上是 liveness analysis 的应用，是一个 backward dataflow analysis。Aggressive 的意思是对于每一条指令，该 pass 假设该指令是 dead 除非该指令被证明是 live 的，在分析结束后所有的被认为是 dead 的指令都会被消除。

该 pass 的代码实现位于 `llvm-8.0.1.src/include/llvm/Transforms/Scalar/ADCE.h` 和 `llvm-8.0.1.src/lib/Transforms/Scalar/ADCE.cpp` 。

Implementation of ADCE

我们从 `ADCEPass` 的入口开始分析：

```
PreservedAnalyses ADCEPass::run(Function &F, FunctionAnalysisManager &FAM)
{
  // ADCE does not need DominatorTree, but require DominatorTree here
  // to update analysis if it is already available.
  auto *DT = FAM.getCachedResult<DominatorTreeAnalysis>(F);
  auto &PDT = FAM.getResult<PostDominatorTreeAnalysis>(F);
  if (!AggressiveDeadCodeElimination(F, DT,
PDT).performDeadCodeElimination())
    return PreservedAnalyses::all();

  PreservedAnalyses PA;
  PA.preserveSet<CFGAnalyses>();
  PA.preserve<GlobalsAA>();
```

```
  PA.preserve<DominatorTreeAnalysis>();
  PA.preserve<PostDominatorTreeAnalysis>();
  return PA;
}
```

可以看到 ADCE 实际是在类 `AggressiveDeadCodeElimination` 中实现的，而该类的构造函数有三个参数：待分析的函数，待分析函数的支配树 `DominatorTree` 和 后支配树 `PostDominatorTree` ，即，对于一个待分析的函数来说，执行 ADCE 分析需要 `DominatorTree` 和 `PostDominatorTree` （因为该分析是一个 backward 分析，实际上在分析时需要的是 `PostDominatorTree` ，而不需要 `DominatorTree` ，但是因为该 pass 删除了一些基本块，所以 `DominatorTree` 作为参数传进来是为了对 `DominatorTree` 进行更新）。

函数 `AggressiveDeadCodeElimination::performDeadCodeElimination()` 的定义很简单：

```
bool AggressiveDeadCodeElimination::performDeadCodeElimination() {
  initialize();
  markLiveInstructions();
  return removeDeadInstructions();
}
```

根据该函数的实现，可以看到该 ADCE pass 的流程很简单清晰：首先是初始化工作（选取哪些指令作为分析的起点），然后标记 live 的 指令，最后消除那些 dead 指令。

后续将按顺序分析这三个函数的实现。在这之前，先看下在这三个函数中会用到一些的变量和函数的定义。

1. `AggressiveDeadCodeElimination` 的成员变量 `MapVector<BasicBlock *, BlockInfoType> BlockInfo`

   将基本块 `BasicBlock` 映射到存储该基本块相关信息的 `BlockInfoType` ，`BlockInfoType` 的定义如下：

   ```
   struct BlockInfoType {
     /// True when this block contains a live instructions.
     bool Live = false;
     /// True when this block ends in an unconditional branch.
     bool UnconditionaUnlBranch = false;
     /// True when this block is known to have live PHI nodes.
     bool HasLivePhiNodes = false;
     /// Control dependence sources need to be live for this block.
     bool CFLive = false;
     /// Quick access to the LiveInfo for the terminator,
     /// holds the value &InstInfo[Terminator]
   ```

```
  InstInfoType *TerminatorLiveInfo = nullptr;
  /// Corresponding BasicBlock.
  BasicBlock *BB = nullptr;
  /// Cache of BB->getTerminator().
  Instruction *Terminator = nullptr;
  /// Post-order numbering of reverse control flow graph.
  unsigned PostOrder;
  bool terminatorIsLive() const { return TerminatorLiveInfo->Live; }
};
```

注释很清晰： `Live` 用于说明该基本块中是否存在 live 的指令 (instructions)； `UnconditionaUnlBranch` 用于说明该基本块是否以一个无条件分支指令 (unconditional branch instruction) 结束； `HasLivePhiNodes` 用于说明该基本块是否存在 live 的 `PHINode` ； `CFLive` 用于说明该基本块所控制依赖的基本块应该是 live 的； `TerminatorLiveInfo` 指向该基本块的 `Terminator` 指令的相关信息 `InstInfoType` （见成员变量 `DenseMap<Instruction *, InstInfoType> InstInfo` ）； `BB` 就是该 `BlockInfoType` 所描述的基本块； `Terminator` 就是该基本块的 `Terminator` 指令，存储在 `BlockInfoType` 中起到一个 cache 的作用； `PostOrder` 是该基本块在 reverse control flow graph 中的 post-order 编号。

2. `AggressiveDeadCodeElimination` 的成员变量 `DenseMap<Instruction *, InstInfoType> InstInfo`

   将指令 `Instruction` 映射到存储该指令相关信息的 `InstInfoType` ， `InstInfoType` 的定义如下：

   ```
   struct InstInfoType {
     /// True if the associated instruction is live.
     bool Live = false;
     /// Quick access to information for block containing associated
   Instruction.
     struct BlockInfoType *Block = nullptr;
   };
   ```

   成员变量 `Live` 用于说明 `InstInfoType` 对应的指令是否为 live；成员变量 `Block` 指向的就是该指令存在的基本块所对应的 `BlockInfoType` 。

3. `AggressiveDeadCodeElimination` 的成员变量 `SmallPtrSet<BasicBlock *, 16> BlocksWithDeadTerminators`

   该成员变量存储那些基本块的 terminator 指令不是 live 的基本块。

4. `AggressiveDeadCodeElimination` 的成员变量 `SmallPtrSet<BasicBlock *, 16> NewLiveBlocks`

该成员变量的注释：The set of blocks which we have determined whose control dependence sources must be live and which have not had those dependences analyzed.

就是说，如果某个基本块所控制依赖的那些基本块应该是 live 的，但是这控制依赖还没有分析，就将该基本块暂时存储在 `NewLiveBlocks` 中 。

5. `AggressiveDeadCodeElimination` 的成员变量 `SmallVector<Instruction *, 128> Worklist`

   该成员变量用于存储已知是 live 的 Instruction，需要注意的是在函数 `initialize()`，`markLiveInstructions()` 执行完后，该变量为空，在函数 `removeDeadInstructions()` 中复用了该成员变量存储用于存储需要被消除的 dead instructions 。

6. 函数 `static bool isUnconditionalBranch(Instruction *Term);`

   该函数很简单，就是判断给定的 `Instruction` 是否是一个无条件分支指令，实现如下：

```
static bool isUnconditionalBranch(Instruction *Term) {
  auto *BR = dyn_cast<BranchInst>(Term);
  return BR && BR->isUnconditional();
}
```

   首先看参数 `Instruction *Term` 是否为 `BranchInst` 类型，如果该参数确实是一个 `BranchInst`，并且是 `Unconditional` 的 `BranchInst`，则该函数返回 true 。

7. 类 `AggressiveDeadCodeElimination` 的成员函数 `isAlwaysLive(Instruction &I)`

   从函数名就能看出该函数对于分析来说很关键，因为该函数确定了什么样的指令是 always live 的 。

```
bool AggressiveDeadCodeElimination::isAlwaysLive(Instruction &I) {
  // TODO -- use llvm::isInstructionTriviallyDead
  if (I.isEHPad() || I.mayHaveSideEffects()) {
    // Skip any value profile instrumentation calls if they are
    // instrumenting constants.
    if (isInstrumentsConstant(I))
      return false;
    return true;
  }
  if (!I.isTerminator())
    return false;
  if (RemoveControlFlowFlag && (isa<BranchInst>(I) || isa<SwitchInst>
(I)))
```

```
    return false;
  return true;
}
```

可以看到可能有副作用的指令 (如 `StoreInst`) 是 always live 的。变量 `RemoveControlFlowFlag` 为 true (默认为true) 时，除了 `BranchInst` 和 `SwitchInst` 以外的 terminator 指令都 always live 的；如果 `RemoveControlFlowFlag` 为 false 的话，那么所有的 terminator 指令都 always live 的。

8. 类 `AggressiveDeadCodeElimination` 的成员函数 `void markLive(Instruction *I)` , `void markLive(BasicBlock *BB)` 和 `void markLive(BlockInfoType &BBInfo)` 。

   这里将这三个重载的函数一同说明了。

   - ```
     void AggressiveDeadCodeElimination::markLive(Instruction *I) {
       auto &Info = InstInfo[I];
       if (Info.Live)
         return;

       LLVM_DEBUG(dbgs() << "mark live: "; I->dump());
       Info.Live = true;
       Worklist.push_back(I);

       // Collect the live debug info scopes attached to this
     instruction.
       if (const DILocation *DL = I->getDebugLoc())
         collectLiveScopes(*DL);

       // Mark the containing block live
       auto &BBInfo = *Info.Block;
       if (BBInfo.Terminator == I) {
         BlocksWithDeadTerminators.erase(BBInfo.BB);
         // For live terminators, mark destination blocks
         // live to preserve this control flow edges.
         if (!BBInfo.UnconditionalBranch)
           for (auto *BB : successors(I->getParent()))
             markLive(BB);
       }
       markLive(BBInfo);
     }
     ```
     ```

首先就是将 `InstInfo[I].Live` 设置为 true，然后将该指令存储进成员变量 `Worklist` 中，没什么好说的。然后就是看该指令是否为其所在基本块的 terminator 指令，如果是的话，就更新 `BlocksWithDeadTerminators`（如果该指令在 `BlocksWithDeadTerminators` 中，就从中删除该指令）。如果该指令是其所在基本块的 terminator 指令，并且是一个无条件的 `BranchInst`，就调用 `void markLive(BasicBlock *BB)` 将其所在基本块的后继基本块都设置为 live，最后调用 `void markLive(BlockInfoType &BBInfo)` 将该指令所在的基本块设置为 live。

- `void markLive(BasicBlock *BB)` 的实现就是对 `void markLive(BlockInfoType &BBInfo)` 的一层封装。

```
void markLive(BasicBlock *BB) { markLive(BlockInfo[BB]); }
```

- `void markLive(BlockInfoType &BBInfo)` 的实现如下：

```
void AggressiveDeadCodeElimination::markLive(BlockInfoType &BBInfo)
{
  if (BBInfo.Live)
    return;
  LLVM_DEBUG(dbgs() << "mark block live: " << BBInfo.BB->getName()
<< '\n');
  BBInfo.Live = true;
  if (!BBInfo.CFLive) {
    BBInfo.CFLive = true;
    NewLiveBlocks.insert(BBInfo.BB);
  }

  // Mark unconditional branches at the end of live
  // blocks as live since there is no work to do for them later
  if (BBInfo.UnconditionalBranch)
    markLive(BBInfo.Terminator);
}
```

首先将 `BBInfo.Live` 设置为 true 。如果 `BBInfo.CFLive` 为 false，就将其设置为 true，并且将当前基本块存储进 `NewLiveBlocks` 中，即如果将当前基本块设置为 live 的，那么该基本块所控制依赖的基本块们也应该是 live 的。最后如果该基本块的 terminator 指令是一个无条件的 `BranchInst`，就对该基本块的 terminator 指令调用函数 `void markLive(Instruction *I)` 。

initialize()

函数 `initialize()` 用于确定选取哪些指令作为分析的起点。因函数的函数体很长，我们逐部分地分析：

第一部分代码如下：

```
void AggressiveDeadCodeElimination::initialize() {
  auto NumBlocks = F.size();

  // We will have an entry in the map for each block so we grow the
  // structure to twice that size to keep the load factor low in the hash table.
  BlockInfo.reserve(NumBlocks);
  size_t NumInsts = 0;

  // Iterate over blocks and initialize BlockInfoVec entries, count
  // instructions to size the InstInfo hash table.
  for (auto &BB : F) {
    NumInsts += BB.size();
    auto &Info = BlockInfo[&BB];
    Info.BB = &BB;
    Info.Terminator = BB.getTerminator();
    Info.UnconditionalBranch = isUnconditionalBranch(Info.Terminator);
  }

  // Initialize instruction map and set pointers to block info.
  InstInfo.reserve(NumInsts);
  for (auto &BBInfo : BlockInfo)
    for (Instruction &I : *BBInfo.second.BB)
      InstInfo[&I].Block = &BBInfo.second;

  // Since BlockInfoVec holds pointers into InstInfo and vice-versa, we may not
  // add any more elements to either after this point.
  for (auto &BBInfo : BlockInfo)
    BBInfo.second.TerminatorLiveInfo = &InstInfo[BBInfo.second.Terminator];

  // Collect the set of "root" instructions that are known live.
  for (Instruction &I : instructions(F))
    if (isAlwaysLive(I))
      markLive(&I);
```

上面这一部分代码就是对 `AggressiveDeadCodeElimination` 的成员变量 `BlockInfo` 和 `InstInfo` 的初始化。

值得注意的地方主要是这三个函数调用： `Info.UnconditionalBranch = isUnconditionalBranch(Info.Terminator);` ， `isAlwaysLive(I)` 和 `markLive(&I)` 。这三个函数的实现已经在前面说明过了，并且函数命名很清晰，很直观的知道函数的作用。

第二部分代码如下：

```
if (!RemoveControlFlowFlag)
  return;

if (!RemoveLoops) {
  // This stores state for the depth-first iterator. In addition
  // to recording which nodes have been visited we also record whether
  // a node is currently on the "stack" of active ancestors of the current
  // node.
  using StatusMap = DenseMap<BasicBlock *, bool>;

  class DFState : public StatusMap {
  public:
    std::pair<StatusMap::iterator, bool> insert(BasicBlock *BB) {
      return StatusMap::insert(std::make_pair(BB, true));
    }

    // Invoked after we have visited all children of a node.
    void completed(BasicBlock *BB) { (*this)[BB] = false; }

    // Return true if \p BB is currently on the active stack
    // of ancestors.
    bool onStack(BasicBlock *BB) {
      auto Iter = find(BB);
      return Iter != end() && Iter->second;
    }
  } State;

  State.reserve(F.size());
  // Iterate over blocks in depth-first pre-order and
  // treat all edges to a block already seen as loop back edges
  // and mark the branch live it if there is a back edge.
  for (auto *BB: depth_first_ext(&F.getEntryBlock(), State)) {
    Instruction *Term = BB->getTerminator();
    if (isLive(Term))
      continue;

    for (auto *Succ : successors(BB))
      if (State.onStack(Succ)) {
        // back edge....
        markLive(Term);
```

```
        break;
      }
    }
  }
```

如果变量 `RemoveControlFlowFlag` 为 false (默认为 true)，则直接 return；如果 `RemoveLoops` 为 false (默认为 false) 的话，那么从函数 F 的入口基本块开始 depth-first pre-order 顺序遍历函数的基本块，如果存在基本块的 terminator 指令不是 live，并且该基本块的后继基本块已经被遍历过了，我们认为这样的边是一条回边，对将基本块的 terminator 指令调用函数 `void markLive(Instruction *I)` 。

第三部分代码如下：

```
// Mark blocks live if there is no path from the block to a
// return of the function.
// We do this by seeing which of the postdomtree root children exit the
// program, and for all others, mark the subtree live.
for (auto &PDTChild : children<DomTreeNode *>(PDT.getRootNode())) {
  auto *BB = PDTChild->getBlock();
  auto &Info = BlockInfo[BB];
  // Real function return
  if (isa<ReturnInst>(Info.Terminator)) {
    LLVM_DEBUG(dbgs() << "post-dom root child is a return: " << BB->getName()
                      << '\n';);
    continue;
  }

  // This child is something else, like an infinite loop.
  for (auto DFNode : depth_first(PDTChild))
    markLive(BlockInfo[DFNode->getBlock()].Terminator);
}
```

如果存在某些基本块，这些基本块没有路径到达函数的 return 指令，那么就将这些基本块设置为 live 。具体实现时是这样的，首先看后支配树的根节点，如果根节点的子节点基本块的 terminator 指令是 `ReturnInst` 则跳过 (后支配树上 `ReturnInst` 所在的基本块的子孙节点一定能到达该 `ReturnInst` 所在的基本块)，对于根节点的子节点基本块，如果其 terminator 指令不是 `ReturnInst` ，则以此基本块为起点 depth-first 遍历，对所有遍历到的基本块，对其 terminator 指令调用函数 `void markLive(Instruction *I)` 。

第四部分代码：

```
// Treat the entry block as always live
auto *BB = &F.getEntryBlock();
auto &EntryInfo = BlockInfo[BB];
EntryInfo.Live = true;
if (EntryInfo.UnconditionalBranch)
  markLive(EntryInfo.Terminator);

// Build initial collection of blocks with dead terminators
for (auto &BBInfo : BlockInfo)
  if (!BBInfo.second.terminatorIsLive())
    BlocksWithDeadTerminators.insert(BBInfo.second.BB);
```

> }

函数 `initialize()` 的最后一部分代码，将函数的入口基本块设置为 live，如果入口基本块的 terminator 指令是无条件的 `BranchInst`，则对此 `BranchInst` 调用函数 `void markLive(Instruction *I)`，最后更新成员变量 `BlocksWithDeadTerminators` 。

markLiveInstructions()

`markLiveInstructions()` 就是标记所有 live 的 instructions 。

```
void AggressiveDeadCodeElimination::markLiveInstructions() {
  // Propagate liveness backwards to operands.
  do {
    // Worklist holds newly discovered live instructions
    // where we need to mark the inputs as live.
    while (!Worklist.empty()) {
      Instruction *LiveInst = Worklist.pop_back_val();
      LLVM_DEBUG(dbgs() << "work live: "; LiveInst->dump(););

      for (Use &OI : LiveInst->operands())
        if (Instruction *Inst = dyn_cast<Instruction>(OI))
          markLive(Inst);

      if (auto *PN = dyn_cast<PHINode>(LiveInst))
        markPhiLive(PN);
    }

    // After data flow liveness has been identified, examine which branch
    // decisions are required to determine live instructions are executed.
    markLiveBranchesFromControlDependences();
```

```
  } while (!Worklist.empty());
}
```

在函数 `initialize()` 执行后， `Worklist` 中存储了被标记为 live 的 instructions (每次调用函数 `void markLive(Instruction *I)` 时，该函数将指令 `I` 存储进 `Worklist`)。在函数 `markLiveInstructions()` 中通过 worklist 算法不断标记新的 live instructions，沿着 use-def 方向，如果一个指令被标记为 live，那么对其操作数 def 的 instruction 也被标记为 live。如果 `Worlist` 中某条指令是 `PHINode` ，则调用函数 `void markPhiLive(PHINode *PN)` 进行特殊处理。

```
void AggressiveDeadCodeElimination::markPhiLive(PHINode *PN) {
  auto &Info = BlockInfo[PN->getParent()];
  // Only need to check this once per block.
  if (Info.HasLivePhiNodes)
```



```
  // which will trigger marking live branches upon which
  // that block is control dependent.
  for (auto *PredBB : predecessors(Info.BB)) {
    auto &Info = BlockInfo[PredBB];
    if (!Info.CFLive) {
      Info.CFLive = true;
      NewLiveBlocks.insert(PredBB);
    }
  }
}
```

首先将 `PHINode` 所在基本块对应的 `BlockInfoType` 的 `HasLivePhiNodes` 域设置为 true。如果该 `PHINode` 所在基本块的的前驱基本块对应的 `BlockInfoType` 的 `CFLive` 域为 false，则将该域设置为 true，并将此前驱基本块放入 `NewLiveBlocks` 中，即该前驱基本块所控制依赖的基本块也应该是 live 的 。 `markLiveBranchesFromControlDependences()` 函数就是用于将 live 基本块所控制依赖的基本块也标记为 live 的。
`markLiveBranchesFromControlDependences()` 函数的实现如下：

```
void
AggressiveDeadCodeElimination::markLiveBranchesFromControlDependences() {
  if (BlocksWithDeadTerminators.empty())
    return;

  // The dominance frontier of a live block X in the reverse
  // control graph is the set of blocks upon which X is control
  // dependent. The following sequence computes the set of blocks
  // which currently have dead terminators that are control
```

```
  // dependence sources of a block which is in NewLiveBlocks.

  SmallVector<BasicBlock *, 32> IDFBlocks;
  ReverseIDFCalculator IDFs(PDT);
  IDFs.setDefiningBlocks(NewLiveBlocks);
  IDFs.setLiveInBlocks(BlocksWithDeadTerminators);
  IDFs.calculate(IDFBlocks);
  NewLiveBlocks.clear();

  // Dead terminators which control live blocks are now marked live.
  for (auto *BB : IDFBlocks) {
    LLVM_DEBUG(dbgs() << "live control in: " << BB->getName() << '\n');
    markLive(BB->getTerminator());
  }
}
```

首先计算 `NewLiveBlocks` 的 Post Iterated Dominance Frontier (即 `NewLiveBlocks` 中的基本块所控制依赖的基本块)，然后对得到的基本块的 terminator 指令调用函数 `void markLive(Instruction *I)` 。

removeDeadInstructions()

`removeDeadInstructions()` 函数将所有没有被标记为 live 的instructions 消除。

```
bool AggressiveDeadCodeElimination::removeDeadInstructions() {
  // Updates control and dataflow around dead blocks
  updateDeadRegions();

  // The inverse of the live set is the dead set.  These are those
instructions
  // that have no side effects and do not influence the control flow or
return
  // value of the function, and may therefore be deleted safely.
  // NOTE: We reuse the Worklist vector here for memory efficiency.
  for (Instruction &I : instructions(F)) {
    // Check if the instruction is alive.
    if (isLive(&I))
      continue;

    if (auto *DII = dyn_cast<DbgInfoIntrinsic>(&I)) {
      // Check if the scope of this variable location is alive.
      if (AliveScopes.count(DII->getDebugLoc()->getScope()))
        continue;

      // Fallthrough and drop the intrinsic.
    }
```

```
    // Prepare to delete.
    Worklist.push_back(&I);
    I.dropAllReferences();
  }

  for (Instruction *&I : Worklist) {
    ++NumRemoved;
    I->eraseFromParent();
  }

  return !Worklist.empty();
}
```

这里重用了成员变量 `Worklist` ，因为执行至该函数时， `Worklist` 已经是空的了，这里复用它来存储需要被消除的指令。该函数中值得注意的是对 `updateDeadRegions()` 函数的调用，因为我们删除了一些基本块，所以需要对 `DominatorTree` 进行更新，这就是在 `updateDeadRegions()` 中实现的，这里不再详细分析了。

Summary

大体上来说，该分析的流程如下：

1. 该算法的起点是所有 terminator 指令 (例如 `ReturnInst`), may side effecting 指令 (例如 `StoreInst`)，这些认为是 live 的
2. 然后，利用 SSA 形式的 use-def 信息，从上述起点出发迭代，把所有能通过 use-def 链到达的指令都标记为 live
3. 最后，没有被标记为 live 的指令就是 dead，遍历一次所有指令，把没有被标记为 live 的指令删除，DCE就完成了

该分析可以看作是一个使用了只有 2 个元素的 lattice (bottom 是 live，top 是 dead) 的 backward 数据流分析。