CalledValuePropagation

Introduction

CalledValuePropagation 是一个 transform pass，对于一些间接调用点 (indirect call sites)，在调用点处添加了名为 !callee 的 metadata 来表示被调函数 (callees) 的可能取值。

CalledValuePropagation 在 LLVM 6.x 版本中被引入，Differential Revision: https://reviews.llvm.org/D37355

本文中对该 pass 的代码分析基于 LLVM 7.0.0 版本，其实现代码位于 `llvm-7.0.0.src/include/llvm/Transforms/IPO/CalledValuePropagation.h` 和 `llvm-7.0.0.src/lib/Transforms/IPO/CalledValuePropagation.cpp` (IPO 是 Inter-Procedural Optimization 的简写)。CalledValuePropagation 基于 SparsePropagation 实现。(`llvm-7.0.0.src/include/llvm/Analysis/SparsePropagation.h`)

可以通过 opt 命令来调用该 pass 来获取间接调用点的被调函数的可能取值。例如

```
➜opt -S -called-value-propagation simple-arguments.ll -o simple-arguments.opt.ll
```

如果打开生成的LLVM IR 文件 simple-arguments.opt.ll 会看到有如下的一条指令：

```
%tmp3 = call i1 %cmp(i64* %tmp1, i64* %tmp2), !callees !0
```

!callees !0 就是用于表示被调函数可能取值的 metadata，在文件 simple-arguments.opt.ll 的最后会看到如下的内容：

```
!0 = !{i1 (i64*, i64*)* @ugt, i1 (i64*, i64*)* @ule}
```

即 `%tmp3 = call i1 %cmp(i64* %tmp1, i64* %tmp2)` 中可能的被调函数是 `ugt` 和 `ule` 。

CalledValuePropagationPass

`CalledValuePropagationPass` 类的定义如下：

```
class CalledValuePropagationPass : public
PassInfoMixin<CalledValuePropagationPass>
{
public:
    PreservedAnalyses run(Module &M, ModuleAnalysisManager &);
};

PreservedAnalyses CalledValuePropagationPass::run(Module &M,
ModuleAnalysisManager &)
{
    runCVP(M);
    return PreservedAnalyses::all();
}
```

可以看到核心功能在函数 `runCVP()` 中实现：

```
static bool runCVP(Module &M)
{
    // Our custom lattice function and generic sparse propagation solver.
    CVPLatticeFunc Lattice;
    SparseSolver<CVPLatticeKey, CVPLatticeVal> Solver(&Lattice);

    // For each function in the module, if we can't track its arguments,
let the
    // generic solver assume it is executable.
    for (Function &F : M)
        if (!F.isDeclaration() && !canTrackArgumentsInterprocedurally(&F))
            Solver.MarkBlockExecutable(&F.front());

    // Solver our custom lattice. In doing so, we will also build a set of
    // indirect call sites.
    Solver.Solve();
```

```
    // Attach metadata to the indirect call sites that were collected
indicating
    // the set of functions they can possibly target.
    bool Changed = false;
    MDBuilder MDB(M.getContext());
    for (Instruction *C : Lattice.getIndirectCalls())
    {
        CallSite CS(C);
        auto RegI = CVPLatticeKey(CS.getCalledValue(),
IPOGrouping::Register);
        CVPLatticeVal LV = Solver.getExistingValueState(RegI);
        if (!LV.isFunctionSet() || LV.getFunctions().empty())
            continue;
        MDNode *Callees = MDB.createCallees(LV.getFunctions());
        C->setMetadata(LLVMContext::MD_callees, Callees);
        Changed = true;
    }

    return Changed;
}
```

这段代码的逻辑很清晰，首先创建了 LatticeFunction 和 SparseSolver，如果函数不是一个函数声明并且该函数的参数不能过程间地追踪，那么将该函数的入口基本块添加至集合 BBExecutable 和 BBWorkList 中。接着调用 `Solver.Solve();` 进行求解，在求解过程中对间接调用点的被调函数的可能取值进行了收集，最后将可能的被调函数以 call sites 的 metadata 的形式写入 LLVM IR。

所以 `CVPLatticeKey`，`CVPLatticeVal`，`CVPLatticeFunc` 都是怎么定义的?

CVPLatticeKey

`CVPLatticeKey` 是 LatticeFunction 的 key type。

```
enum class IPOGrouping { Register, Return, Memory };
using CVPLatticeKey = PointerIntPair<Value *, 2, IPOGrouping>;
```

为了能够进行过程间分析，将 LLVM Values 分成了三类：Register, Return 和 Memory。Register用于表示 SSA registers，Return 用于表示函数的返回值，Memory 用于表示 in-memory values（ `StoreInst` 和 `LoadInst` 的 PointerOperand 是 `GlobalVariable` 时，会将该 PointerOperand 设置为 Memory 类型的 `CVPLatticeKey` ）。

`CVPLatticeKey` 是由 LLVM `Value*` 和 `IPOGrouping` 组成的 `PointerIntPair` 。

CVPLatticeVal

`CVPLatticeKey` 是 LatticeFunction 的 value type。

```
class CVPLatticeVal
{
public:
    enum CVPLatticeStateTy
    {
        Undefined,
        FunctionSet,
```



```
    CVPLatticeStateTy LatticeState;
    std::vector<Function *> Functions;
};
```

`CVPLatticeVal` 有两个成员变量：LatticeState，Functions。成员变量 Functions 用来存储 call sites 的被调函数的可能取值，成员变量 LatticeState 有四种可能取值：Undefined，FunctionSet， Overdefined， Untracked，当 LatticeState 是 FunctionSet 以外的其他三种状态时，Functions 为空。（Undefined 对应半格中的顶元素，根据半格的定义，对于任意数据流值 x，顶元素 ^ x = x，即顶元素与 x 的交汇运算的结果都是x；Overdefined 对应半格中的底元素，根据半格的定义，对于任意数据流值 x，底元素 ^ x = 底元素）

CVPLatticeFunc

`CVPLatticeFunc` 继承自 `AbstractLatticeFunction`： `class CVPLatticeFunc : public AbstractLatticeFunction<CVPLatticeKey, CVPLatticeVal>`

首先看一下 `CVPLatticeFunc` 是怎么重写的 `MergeValues()` 函数（对应数据流分析中的交汇运算）。

```
CVPLatticeVal MergeValues(CVPLatticeVal X, CVPLatticeVal Y) override {
  if (X == getOverdefinedVal() || Y == getOverdefinedVal())
    return getOverdefinedVal();
  if (X == getUndefVal() && Y == getUndefVal())
    return getUndefVal();
  std::vector<Function *> Union;
  std::set_union(X.getFunctions().begin(), X.getFunctions().end(),
                 Y.getFunctions().begin(), Y.getFunctions().end(),
```

```
                  std::back_inserter(Union), CVPLatticeVal::Compare{});
  if (Union.size() > MaxFunctionsPerValue)
    return getOverdefinedVal();
  return CVPLatticeVal(std::move(Union));
}
```

首先对需要进行交汇运算的两个操作数进行判断，如果是其中一个是底元素，那么交汇运算的结果就是底元素，直接返回 `getOverdefinedVal()`；如果两个操作数都是顶元素，那么交汇运算的结果就是顶元素，直接返回 `getUndefVal()`；其他情况就是对两个操作数的数据流值进行并集的操作。（注意到这里对并集的运算结果的大小进行判断，如果超过 MaxFunctionsPerValue (默认为 4)，就返回底元素 `getOverdefinedVal()`，代码的注释中的解释是： We likely can't do anything useful for call sites with a large number of possible targets, anyway.）

然后看一下 `CVPLatticeFunc` 是怎么重写的 `ComputeInstructionState()` 函数（对应数据流分析中的传递函数）。

```
void ComputeInstructionState(
    Instruction &I, DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
    SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) override {
  switch (I.getOpcode()) {
  case Instruction::Call:
    return visitCallSite(cast<CallInst>(&I), ChangedValues, SS);
  case Instruction::Invoke:
    return visitCallSite(cast<InvokeInst>(&I), ChangedValues, SS);
  case Instruction::Load:
    return visitLoad(*cast<LoadInst>(&I), ChangedValues, SS);
  case Instruction::Ret:
    return visitReturn(*cast<ReturnInst>(&I), ChangedValues, SS);
  case Instruction::Select:
    return visitSelect(*cast<SelectInst>(&I), ChangedValues, SS);
  case Instruction::Store:
    return visitStore(*cast<StoreInst>(&I), ChangedValues, SS);
  default:
    return visitInst(I, ChangedValues, SS);
  }
}
```

对于不同的 `Instruction` 实现不同的传递函数的逻辑。

visitSelect

下面先对 `SelectInst` 的传递函数 `visitSelect()` 进行分析， `SelectInst` 被用于实现基于条件的值的选择， `SelectInst` 不需要 LLVM IR 级别的分支指令的参与。语法如下：

```
<result> = select selty <cond>, <ty> <val1>, <ty> <val2>             ;
yields ty

selty is either i1 or {<N x i1>}
```

一个 SelectInst 的例子如下：

```
%X = select i1 true, i8 17, i8 42          ; yields i8:17
```

在这条 SelectInst 中， <cond> 为 true， <val1> 为 17，它的 <ty> 为 i8， <val2> 为 42，它的 <ty> 为 i8。 visitSelect() 函数的定义如下：

```
void visitSelect(SelectInst &I,
                 DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
                 SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  auto RegI = CVPLatticeKey(&I, IPOGrouping::Register);
  auto RegT = CVPLatticeKey(I.getTrueValue(), IPOGrouping::Register);
  auto RegF = CVPLatticeKey(I.getFalseValue(), IPOGrouping::Register);
  ChangedValues[RegI] =
      MergeValues(SS.getValueState(RegT), SS.getValueState(RegF));
}
```

首先为这条 SelectInst 创建了一个类型为 IPOGrouping::Register 的 CVPLatticeKey RegI，然后为 SelectInst 的 TrueValue 和 FalseValue 分别创建了类型为 IPOGrouping::Register 的 CVPLatticeKey RegT 和 RegF。RegI 对应的数据流值 CVPLatticeVal 是由 RegT 的数据流值和 RegF 的数据流值进行交汇运算 MergeValues() 后得到的。 RegT 的数据流值和 RegF 的数据流值是通过 SparseSolver 的成员函数 getValueState() 得到的， getValueState() 的定义如下：

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
LatticeVal
SparseSolver<LatticeKey, LatticeVal, KeyInfo>::getValueState(LatticeKey
Key) {
  auto I = ValueState.find(Key);
  if (I != ValueState.end())
    return I->second; // Common case, in the map

  if (LatticeFunc->IsUntrackedValue(Key))
    return LatticeFunc->getUntrackedVal();
  LatticeVal LV = LatticeFunc->ComputeLatticeVal(Key);

  // If this value is untracked, don't add it to the map.
  if (LV == LatticeFunc->getUntrackedVal())
```

```
    return LV;
  return ValueState[Key] = std::move(LV);
}
```

如果之前计算过某个 `LatticeKey` 对应的数据流值 `LatticeVal`，那么就会被存在 `DenseMap<LatticeKey, LatticeVal>` ValueState 中，如果是第一次查询这个 `LatticeKey` 对应的数据流值 `LatticeVal`，那么会调用函数 `LatticeFunc->ComputeLatticeVal()`，对于 CalledValuePropagationPass 来讲，LatticeFunc 就是 `CVPLatticeFunc`。 `CVPLatticeFunc` 的 `ComputeLatticeVal()` 函数的定义如下：

```
CVPLatticeVal ComputeLatticeVal(CVPLatticeKey Key) override {
  switch (Key.getInt()) {
  case IPOGrouping::Register:
    if (isa<Instruction>(Key.getPointer())) {
      return getUndefVal();
    } else if (auto *A = dyn_cast<Argument>(Key.getPointer())) {
      if (canTrackArgumentsInterprocedurally(A->getParent()))
        return getUndefVal();
    } else if (auto *C = dyn_cast<Constant>(Key.getPointer())) {
      return computeConstant(C);
    }
    return getOverdefinedVal();
  case IPOGrouping::Memory:
  case IPOGrouping::Return:
    if (auto *GV = dyn_cast<GlobalVariable>(Key.getPointer())) {
      if (canTrackGlobalVariableInterprocedurally(GV))
        return computeConstant(GV->getInitializer());
    } else if (auto *F = cast<Function>(Key.getPointer()))
      if (canTrackReturnsInterprocedurally(F))
        return getUndefVal();
  }
  return getOverdefinedVal();
}
```

我们还是继续 `visitSelect` 函数的逻辑来跟进代码，在 `visitSelect` 函数中，通过 `SparseSolver<LatticeKey, LatticeVal, KeyInfo>::getValueState(LatticeKey Key)` 获取 RegT 的数据流值和 RegF 的数据流值时，如果是第一次查询它们的数据流值，就会调用 `CVPLatticeFunc` 的 `ComputeLatticeVal()` 函数，因为 RegT 和 RegF 都是 IPOGrouping::Register 所以会进入 `case IPOGrouping::Register:` 这个分支，可以看到当 SelectInst 的 TrueValue(RegT) 或者 FalseValue(RegF) 是 `Constant` 时，会调用 `computeConstant()` 函数。

```
CVPLatticeVal computeConstant(Constant *C) {
  if (isa<ConstantPointerNull>(C))
    return CVPLatticeVal(CVPLatticeVal::FunctionSet);
```

```
  if (auto *F = dyn_cast<Function>(C->stripPointerCasts()))
    return CVPLatticeVal({F});
  return getOverdefinedVal();
}
```

在 `computeConstant()` 函数中，如果 Constant *C 是 Function，最终返回这个函数。

举个例子，应该会更直观，我们有这样一条 SelectInst：

```
%func = select i1 %cond, i1 (i64, i64)* @ugt, i1 (i64, i64)* @ule
```

%cond 是某条件，TrueValue 是这个 CalledValuePropagationPass 的分析对象 LLVM IR 中的函数 ugt，FalseValue 是函数 ugt。然后我们想要计算在执行完这条指令后的数据流值，因为这是一个 SelectInst，所以应用传递函数时执行调用的是 `visitSelect` 函数，然后我们是第一次查询该 SelectInst 的 TrueValue 和 FalseValue 对应的 `CVPLatticeVal`，所以会调用函数 `ComputeLatticeVal()`，最终得到的就是 `CVPLatticeVal{@ugt}` 和 `CVPLatticeVal{@ule}`，所以更新后的 %func 对应的 `CVPLatticeVal` 就是 `MergeValues(CVPLatticeVal{@ugt}, CVPLatticeVal{@ule})` 即 `CVPLatticeVal{@ugt, @ule}`。

visitLoad

LoadInst 对应的传递函数是 `visitLoad()`：

```
void visitLoad(LoadInst &I,
               DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
               SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  auto RegI = CVPLatticeKey(&I, IPOGrouping::Register);
  if (auto *GV = dyn_cast<GlobalVariable>(I.getPointerOperand())) {
    auto MemGV = CVPLatticeKey(GV, IPOGrouping::Memory);
    ChangedValues[RegI] =
        MergeValues(SS.getValueState(RegI), SS.getValueState(MemGV));
  } else {
    ChangedValues[RegI] = getOverdefinedVal();
  }
}
```

可以发现，只有当 LoadInst 的 PointerOperand 是 GlobalVariable 时才进行分析，可见该 CalledValuePropagationPass 还是比较保守或者说是比较简单的。 该传递函数很简单，RegI 对应的新数据流值 `CVPLatticeVal` 是由 RegI 的原数据流值和 MemGV 的数据流值进行交汇运算 `MergeValues()` 得到。

visitStore

StoreInst 对应的传递函数是 `visitStore()`：

```
void visitStore(StoreInst &I,
                DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
                SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  auto *GV = dyn_cast<GlobalVariable>(I.getPointerOperand());
  if (!GV)
    return;
  auto RegI = CVPLatticeKey(I.getValueOperand(), IPOGrouping::Register);
  auto MemGV = CVPLatticeKey(GV, IPOGrouping::Memory);
  ChangedValues[MemGV] =
      MergeValues(SS.getValueState(RegI), SS.getValueState(MemGV));
}
```

visitReturn

ReturnInst 对应的传递函数是 `visitReturn()`：

```
void visitReturn(ReturnInst &I,
                 DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
                 SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  Function *F = I.getParent()->getParent();
  if (F->getReturnType()->isVoidTy())
    return;
  auto RegI = CVPLatticeKey(I.getReturnValue(), IPOGrouping::Register);
  auto RetF = CVPLatticeKey(F, IPOGrouping::Return);
  ChangedValues[RetF] =
      MergeValues(SS.getValueState(RegI), SS.getValueState(RetF));
}
```

该传递函数稍微有一点特殊，因为 CalledValuePropagationPass 是过程间分析的。所以对于 ReturnInst，会对该函数的 IPOGrouping::Register 类型的 数据流值 `CVPLatticeVal` 进行更新，这样的话，当有 callsite 调用该函数时，就能计算出该 callsite 的返回值的数据流值。

visitCallSite

CallInst 和 InvokeInst 对应的传递函数都是 `visitCallSite()`：

```
void visitCallSite(CallSite CS,
                   DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
                   SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  Function *F = CS.getCalledFunction();
  Instruction *I = CS.getInstruction();
  auto RegI = CVPLatticeKey(I, IPOGrouping::Register);
```

```
  // If this is an indirect call, save it so we can quickly revisit it when
  // attaching metadata.
  if (!F)
    IndirectCalls.insert(I);

  // If we can't track the function's return values, there's nothing to do.
  if (!F || !canTrackReturnsInterprocedurally(F)) {
    // Void return, No need to create and update CVPLattice state as no one
    // can use it.
    if (I->getType()->isVoidTy())
      return;
    ChangedValues[RegI] = getOverdefinedVal();
    return;
  }

  // Inform the solver that the called function is executable, and perform
  // the merges for the arguments and return value.
  SS.MarkBlockExecutable(&F->front());
  auto RetF = CVPLatticeKey(F, IPOGrouping::Return);
  for (Argument &A : F->args()) {
    auto RegFormal = CVPLatticeKey(&A, IPOGrouping::Register);
    auto RegActual =
        CVPLatticeKey(CS.getArgument(A.getArgNo()), IPOGrouping::Register);
    ChangedValues[RegFormal] =
        MergeValues(SS.getValueState(RegFormal),
SS.getValueState(RegActual));
  }

  // Void return, No need to create and update CVPLattice state as no one
can
  // use it.
  if (I->getType()->isVoidTy())
    return;

  ChangedValues[RegI] =
      MergeValues(SS.getValueState(RegI), SS.getValueState(RetF));
}
```

对函数调用的参数和返回值进行处理，背后的逻辑很简单：对于参数来讲，被调函数的形参的可能取值就是对该函数的所有调用点的实参的并集，因此 `visitCallSite` 就是把当前 call site 的实参的数据流值并入被调函数的形参的数据流值中；而该 call site 的返回值就是所有可能被调函数的返回值的并集，所以 `visitCallSite` 就是把当前 call site 的被调函数的返回值的数据流值（在 VistieReturn 中被设置）并入当前 call site 的返回值的数据流值中。

值得注意的是： `SS.MarkBlockExecutable(&F->front());` ，将被调函数的入口基本块添加至 SparseSolver 的 BBWorkList 和 BBExecutable 集合中。因为这里更新了被调函数的形参的数据流值，所以需要再次对被调函数中的数据流值进行迭代更新。

visitReturn 和 visitCallSite 的实现使得该 CalledValuePropagationPass 是过程间的分析。

visitInst

该函数是对除了上述指令外的其他指令的传递函数，就是简单的设置为数据流值设置为 `getOverdefinedVal()` 。

```
void visitInst(Instruction &I,
               DenseMap<CVPLatticeKey, CVPLatticeVal> &ChangedValues,
               SparseSolver<CVPLatticeKey, CVPLatticeVal> &SS) {
  auto RegI = CVPLatticeKey(&I, IPOGrouping::Register);
  ChangedValues[RegI] = getOverdefinedVal();
}
```

How to call CalledValuePropagationPass in your code

除了可以通过 opt 命令来调用该 pass 来获取间接调用点的被调函数的可能取值，还可以通过 PassManager 在你的代码中调用 CalledValuePropagationPass 。

可以参考：https://github.com/Enna1/LLVM-Clang-Examples/tree/master/use-calledvaluepropagation-in-your-tool