SparsePropagation

Introduction

数据流分析是一种用于在计算在某个程序点的程序状态（数据流值）的技术。基于数据流分析的典型例子有常量传播、到达定值等。

根据R大在知乎的回答（见参考链接），因为 SSA 形式贯穿于 LLVM IR，所以在 LLVM 中都针对 SSA Value 的数据流分析都是用 sparse 方式去做的，而不像传统 IR 那样迭代遍历每条指令去传播信息直到到达不同点（需要注意的是，在 LLVM IR 中 “memory” 不是 SSA value，所以对 “memory” 分析的话，就无法用 sparse 的方式了；但是 LLVM 有一个 memory SSA 的项目，我对 memory SSA 没有了解，后面有时间写篇文章填坑）。

- dense 分析：要用个容器携带所有变量的信息去遍历所有指令，即便某条指令不关心的变量信息也会携带过去
- sparse 分析：变量的信息直接在 def 与 use 之间传播，中间不需要遍历其他不相关的指令

在 LLVM 中提供了一个用于实现 sparse analysis 的 infrastructure，位于 `llvm-7.0.0.src/include/llvm/Analysis/SparsePropagation.h` 。

在标准的数据流分析框架中，应该有如下的组成部分:

- D: 数据流分析方向，forward 还是 backward，即是前向的数据流分析还是后向的数据流分析

- V, ^ : 即数据流值和交汇运算。(V, ^)需要满足半格的定义，即(V, ^)是一个半格
- F: V 到 V 的传递函数族。

基于 SparsePropagation 实例化一个分析时需要提供 LatticeKey, LatticeVal 和 LatticeFunction。其中 LatticeVal 对应数据流值，LatticeKey 用于将 LLVM Value 映射到 LatticeVal，而 LatticeFunction 对应传递函数。好像基于 SparsePropagation 实例化一个分析时，分析方向只能是前向的。

AbstractLatticeFunction

首先，需要继承 AbstractLatticeFunction 类来实现一个 LatticeFunction。

```
template <class LatticeKey, class LatticeVal> class AbstractLatticeFunction 
{
private:
  LatticeVal UndefVal, OverdefinedVal, UntrackedVal;

public:
  AbstractLatticeFunction(LatticeVal undefVal, LatticeVal overdefinedVal,
                          LatticeVal untrackedVal) {
    UndefVal = undefVal;
    OverdefinedVal = overdefinedVal;
    UntrackedVal = untrackedVal;
  }

  virtual ~AbstractLatticeFunction() = default;

  LatticeVal getUndefVal()       const { return UndefVal; }
  LatticeVal getOverdefinedVal() const { return OverdefinedVal; }
  LatticeVal getUntrackedVal()   const { return UntrackedVal; }

  /// IsUntrackedValue - If the specified LatticeKey is obviously 
uninteresting
  /// to the analysis (i.e., it would always return UntrackedVal), this
  /// function can return true to avoid pointless work.
  virtual bool IsUntrackedValue(LatticeKey Key) { return false; }

  /// ComputeLatticeVal - Compute and return a LatticeVal corresponding to 
the
  /// given LatticeKey.
  virtual LatticeVal ComputeLatticeVal(LatticeKey Key) {
    return getOverdefinedVal();
  }

  /// IsSpecialCasedPHI - Given a PHI node, determine whether this PHI node 
is
```

```
  /// one that the we want to handle through ComputeInstructionState.
  virtual bool IsSpecialCasedPHI(PHINode *PN) { return false; }

  /// MergeValues - Compute and return the merge of the two specified 
lattice
  /// values.  Merging should only move one direction down the lattice to
  /// guarantee convergence (toward overdefined).
  virtual LatticeVal MergeValues(LatticeVal X, LatticeVal Y) {
    return getOverdefinedVal(); // always safe, never useful.
  }

  /// ComputeInstructionState - Compute the LatticeKeys that change as a 
result
  /// of executing instruction \p I. Their associated LatticeVals are store 
in
  /// \p ChangedValues.
  virtual void
  ComputeInstructionState(Instruction &I,
                          DenseMap<LatticeKey, LatticeVal> &ChangedValues,
                          SparseSolver<LatticeKey, LatticeVal> &SS) = 0;

  /// PrintLatticeVal - Render the given LatticeVal to the specified 
stream.
  virtual void PrintLatticeVal(LatticeVal LV, raw_ostream &OS);

  /// PrintLatticeKey - Render the given LatticeKey to the specified 
stream.
  virtual void PrintLatticeKey(LatticeKey Key, raw_ostream &OS);

  /// GetValueFromLatticeVal - If the given LatticeVal is representable as 
an
  /// LLVM value, return it; otherwise, return nullptr. If a type is given, 
the
  /// returned value must have the same type. This function is used by the
  /// generic solver in attempting to resolve branch and switch conditions.
  virtual Value *GetValueFromLatticeVal(LatticeVal LV, Type *Ty = nullptr) 
{
    return nullptr;
  }
};
```

核心函数是 `ComputeInstructionState` 和 `MergeValues` 。 `ComputeInstructionState` 对应数据流分析中的传递函数，当执行完一条 Instruction 后，应该怎么样更新数据流值。 `MergeValues` 对应数据流分析中的交汇运算，即怎么样处理数据流值的“合并”。

SparseSolver

除了需要继承 AbstractLatticeFunction 类来实现一个 LatticeFunction。还要创建一个 SparseSolver 对象来进行求解。

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
class SparseSolver {

  /// LatticeFunc - This is the object that knows the lattice and how to
  /// compute transfer functions.
  AbstractLatticeFunction<LatticeKey, LatticeVal> *LatticeFunc;

  /// ValueState - Holds the LatticeVals associated with LatticeKeys.
  DenseMap<LatticeKey, LatticeVal> ValueState;

  /// BBExecutable - Holds the basic blocks that are executable.
  SmallPtrSet<BasicBlock *, 16> BBExecutable;

  /// ValueWorkList - Holds values that should be processed.
  SmallVector<Value *, 64> ValueWorkList;

  /// BBWorkList - Holds basic blocks that should be processed.
  SmallVector<BasicBlock *, 64> BBWorkList;

  using Edge = std::pair<BasicBlock *, BasicBlock *>;

  /// KnownFeasibleEdges - Entries in this set are edges which have already 
had
  /// PHI nodes retriggered.
  std::set<Edge> KnownFeasibleEdges;

public:
  explicit SparseSolver(
      AbstractLatticeFunction<LatticeKey, LatticeVal> *Lattice)
      : LatticeFunc(Lattice) {}
  SparseSolver(const SparseSolver &) = delete;
  SparseSolver &operator=(const SparseSolver &) = delete;

  /// Solve - Solve for constants and executable blocks.
  void Solve();

  void Print(raw_ostream &OS) const;

  /// getExistingValueState - Return the LatticeVal object corresponding to 
the
  /// given value from the ValueState map. If the value is not in the map,
  /// UntrackedVal is returned, unlike the getValueState method.
```

```
  LatticeVal getExistingValueState(LatticeKey Key) const {
    auto I = ValueState.find(Key);
    return I != ValueState.end() ? I->second : LatticeFunc-
>getUntrackedVal();
  }

  /// getValueState - Return the LatticeVal object corresponding to the 
given
  /// value from the ValueState map. If the value is not in the map, its 
state
  /// is initialized.
  LatticeVal getValueState(LatticeKey Key);

  /// isEdgeFeasible - Return true if the control flow edge from the 'From'
  /// basic block to the 'To' basic block is currently feasible.  If
  /// AggressiveUndef is true, then this treats values with unknown lattice
  /// values as undefined.  This is generally only useful when solving the
  /// lattice, not when querying it.
  bool isEdgeFeasible(BasicBlock *From, BasicBlock *To,
                      bool AggressiveUndef = false);

  /// isBlockExecutable - Return true if there are any known feasible
  /// edges into the basic block.  This is generally only useful when
  /// querying the lattice.
  bool isBlockExecutable(BasicBlock *BB) const {
    return BBExecutable.count(BB);
  }

  /// MarkBlockExecutable - This method can be used by clients to mark all 
of
  /// the blocks that are known to be intrinsically live in the processed 
unit.
  void MarkBlockExecutable(BasicBlock *BB);

private:
  /// UpdateState - When the state of some LatticeKey is potentially 
updated to
  /// the given LatticeVal, this function notices and adds the LLVM value
  /// corresponding the key to the work list, if needed.
  void UpdateState(LatticeKey Key, LatticeVal LV);

  /// markEdgeExecutable - Mark a basic block as executable, adding it to 
the BB
  /// work list if it is not already executable.
  void markEdgeExecutable(BasicBlock *Source, BasicBlock *Dest);

  /// getFeasibleSuccessors - Return a vector of booleans to indicate which
  /// successors are reachable from a given terminator instruction.
  void getFeasibleSuccessors(TerminatorInst &TI, SmallVectorImpl<bool> 
```

```
&Succs,
                             bool AggressiveUndef);

  void visitInst(Instruction &I);
  void visitPHINode(PHINode &I);
  void visitTerminatorInst(TerminatorInst &TI);
};
```

SparseSolver 通过 `Solve()` 函数求解数据流方程， `Solve()` 函数实现了 worklist 算法：

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
void SparseSolver<LatticeKey, LatticeVal, KeyInfo>::Solve() {
  // Process the work lists until they are empty!
  while (!BBWorkList.empty() || !ValueWorkList.empty()) {
    // Process the value work list.
    while (!ValueWorkList.empty()) {
      Value *V = ValueWorkList.back();
      ValueWorkList.pop_back();

      LLVM_DEBUG(dbgs() << "\nPopped off V-WL: " << *V << "\n");

      // "V" got into the work list because it made a transition. See if
any
      // users are both live and in need of updating.
      for (User *U : V->users())
        if (Instruction *Inst = dyn_cast<Instruction>(U))
```



```
    while (!BBWorkList.empty()) {
      BasicBlock *BB = BBWorkList.back();
      BBWorkList.pop_back();

      LLVM_DEBUG(dbgs() << "\nPopped off BBWL: " << *BB);

      // Notify all instructions in this basic block that they are newly
      // executable.
      for (Instruction &I : *BB)
        visitInst(I);
    }
  }
}
```

在调用 `Solve()` 函数之前通过 `MarkBlockExecutable()` 设置 BBWorkList 和 BBExecutable，因此初始状态下 ValueWorkList 为空，BBWorkList 不为空。然后会执行到 `while (!BBWorkList.empty())` 这个循环中，对 BBWorkList 中的每一个 `BasicBlock` 中的每一条 `Instruction` 调用 `visitInst()` 函数。

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
void SparseSolver<LatticeKey, LatticeVal, KeyInfo>::visitInst(Instruction 
&I) {
  // PHIs are handled by the propagation logic, they are never passed into 
the
  // transfer functions.
  if (PHINode *PN = dyn_cast<PHINode>(&I))
    return visitPHINode(*PN);

  // Otherwise, ask the transfer function what the result is.  If this is
  // something that we care about, remember it.
  DenseMap<LatticeKey, LatticeVal> ChangedValues;
  LatticeFunc->ComputeInstructionState(I, ChangedValues, *this);
  for (auto &ChangedValue : ChangedValues)
    if (ChangedValue.second != LatticeFunc->getUntrackedVal())
      UpdateState(ChangedValue.first, ChangedValue.second);

  if (TerminatorInst *TI = dyn_cast<TerminatorInst>(&I))
    visitTerminatorInst(*TI);, 
}
```

值得注意的是，在对 `TerminatorInst` 处理时会调用 `visitTerminatorInst()` 函数，该函数将 `TerminatorInst` 所在基本块的可达后继基本块加入到 BBWorkList 和 BBExecutable 中。

SparseSolver 通过 `UpdateState()` 函数对数据流值进行更新：

```
template <class LatticeKey, class LatticeVal, class KeyInfo>
void SparseSolver<LatticeKey, LatticeVal, KeyInfo>::UpdateState(LatticeKey 
Key,
                                                                LatticeVal 
LV) {
  auto I = ValueState.find(Key);
  if (I != ValueState.end() && I->second == LV)
    return; // No change.

  // Update the state of the given LatticeKey and add its corresponding 
LLVM
  // value to the work list.
  ValueState[Key] = std::move(LV);
  if (Value *V = KeyInfo::getValueFromLatticeKey(Key))
```

```
    ValueWorkList.push_back(V);
}
```

如果数据流值被更新了，那么会将该数据流值对应的 LLVM Value 加入到 ValueWorkList 中，所以在 `Solve()` 函数的 `while (!BBWorkList.empty() || !ValueWorkList.empty())` 循环的下一轮迭代时，会进入到 `while (!ValueWorkList.empty())` 这个循环中对每一个 Value 的每一次使用 调用 `visitInst()` 函数进行处理。

`Solve()` 函数就这样不断地进行迭代直至达到不动点位置。

Example

CalledValuePropagation 是一个 transform pass，基于 SparsePropagation 实现了对间接调用点 (indirect call sites)的被调函数的可能取值进行分析。

Reference

https://www.zhihu.com/question/41959902/answer/93087273