实验三 冗余 obc 检测的移除

姓名：邱明阳 王思源

日期：2025 年 1 月 1 日

1 实验目的

1. 熟悉数据流分析的概念，掌握数据流分析框架设计实现过程
2. 掌握常量传播算法实现
3. 了解 LLVM Pass 的使用和编写方法

2 实验内容

1. 完善数据流分析框架
2. 设计实现过程内常量传播算法，实现状态数据结构、传递以及合并规则；并基于结果，移除实验二中添加的冗余 obc 检测代码
3. 在过程内常量传播算法的基础上，添加过程间状态传递，完成过程间常量传播算法

3 实验过程

3.1 前向数据流计算框架实现

Dataflow.h 文件定义了一个数据流分析框架，用于在 LLVM IR 中进行前向和后向数据流分析。笔者完善了其中的 compForwardDataflow 函数，用于计算前向数据流分析的固定点。它接受一个 LLVM 函数、一个数据流访问者、一个结果存储结构和一个初始数据流值作为参数。该函数工作流程如下：

1. 初始化工作列表：
   - 遍历函数的每个基本块。
   - 如果是 main 函数的入口块，设置其输入数据流值为初始值 initval。
   - 否则，调用 visitor->initGlobal 初始化输入数据流值。
   - 将每个基本块加入工作列表。
2. 迭代计算数据流值：

- 当工作列表不为空时，循环执行以下步骤：
- 从工作列表中弹出一个基本块 bb。
- 初始化输入数据流值 in。
- 如果是 main 函数的入口块，设置 in 为初始值 initval。
- 否则，遍历基本块的前驱，合并前驱的输出数据流值到 in。
- 设置基本块的输入数据流值为 in。
- 如果基本块的名称以 obc_err_ 开头，设置其输出数据流值等于输入数据流值。
- 否则，调用 visitor->compDFVal 计算基本块的输出数据流值。
- 如果输出数据流值发生变化，更新输出数据流值，将基本块的后继加入工作列表。

该函数源码如下：

```
template<class T>
void compForwardDataflow(llvm::Function *fn, DataflowVisitor<T> *visitor, typen
ame DataflowBBResult<T>::Type *result, const T & initval) {
    std::vector<llvm::BasicBlock *> worklist; // 工作列表

    // 初始化工作列表
    for (llvm::BasicBlock &bb : *fn) { // 遍历函数的每个基本块
        if (bb.getName() == "entry" && bb.getParent()->getName() == "main") {
            (*result)[&bb].first = initval;
        } else {
            visitor->initGlobal(&(*result)[&bb].first);
        }
        visitor->initGlobal(&(*result)[&bb].second);
        worklist.push_back(&bb); // 将基本块加入工作列表
    }

    // 迭代计算数据流值
    while (!worklist.empty()) { // 当工作列表不为空
        llvm::BasicBlock *bb = worklist.back(); // 获取工作列表的最后一个基本块
        worklist.pop_back(); // 弹出最后一个基本块

        T in, out;
        visitor->initGlobal(&in);
        if (bb->getName() == "entry" && bb->getParent()->getName() == "main") {
            in = initval;
        } else {
            for (llvm::pred_iterator pi = llvm::pred_begin(bb), pe = llvm::pred
_end(bb); pi != pe; ++pi) { // 遍历基本块的前驱
                llvm::BasicBlock *pred = *pi; // 获取前驱
```

```
                visitor->merge(&in, &(*result)[pred].second); // 合并
            }
        }
        (*result)[bb].first = in; // 设置基本块的输入数据流值

        // 计算基本块的输出数据流值
        if (bb->getName().startswith("obc_err_")) {
            (*result)[bb].second = in; // 输出数据流值等于输入数据流值
        } else {
            visitor->compDFVal(bb, &out, true); // 计算数据流值
            if (out != (*result)[bb].second) { // 如果输出数据流值发生变化
                (*result)[bb].second = out; // 更新输出数据流值
                for (llvm::succ_iterator si = llvm::succ_begin(bb), se = llvm::
succ_end(bb); si != se; ++si) { // 遍历基本块的后继
                    llvm::BasicBlock *succ = *si; // 获取后继
                    worklist.push_back(succ); // 将后继加入工作列表
                }
            }
        }
    }
}
```

3.2 操作符重载

ConstantPropagator.h 文件定义了一个常量传播分析器，用于在 LLVM IR 中进行常量传播分析。其中笔者完善了 KSet 类中 operator == 操作符重载，其作用是比较两个 KSet 对象是否相等：

1. 如果两个集合的 top 状态不同，则不相等。
2. 如果两个集合的 top 状态相同且都是 TOP，则相等。
3. 如果两个集合的 top 状态相同且都不是 TOP，则比较 const_vals 是否相等。

源码如下：

```
    bool operator == (const KSet & other) const {
        // DONE: Overload operator == .
        // 如果两个集合不都是 TOP，则不相等
        if (top != other.top) {
            return false;
        }
        // 如果两个集合都是 TOP，则相等
        if (top && other.top) {
            return true;
```

```
        }
        // 如果两个集合都不是 TOP，则比较 const_vals
        return const_vals == other.const_vals;
    }
```

3.3 驱动函数 runOnModule

runOnModule 函数是 ConstValuePass 类的一部分，用于在整个 LLVM 模块上运行常量传播分析。该函数的工作流程如下：

1. 初始化访问者：
   - 创建对象 visitor，用于在数据流分析过程中访问和处理每个指令。
2. 构建函数工作列表：
   - 遍历模块中的每个函数，跳过内建函数和特定名称的函数。
   - 将符合条件的函数加入 f_worklist 集合。
3. 分析每个函数的数据流信息：
   - 遍历 f_worklist 中的每个函数 F：
   - 初始化入口状态 in_state。
   - 初始化全局变量。
   - 调用函数进行前向数据流分析，计算每个基本块的输入和输出状态。
   - 将分析结果存储在 result 中。
   - 检查基本块名称是否为 obc_check，是则将其加入 check_redundant 列表。
4. 移除冗余的 OBC 检查代码：
   - 调用 removeRedundant()函数，移除冗余的 obc 检查代码。

该函数完善部分如下：

```
    // DONE: Compute dataflow information for each function in f_worklist.
    // 分析每个函数的数据流信息
    for (auto *F : f_worklist) { // 遍历f_worklist
        // 初始化入口状态
        ConstValueState in_state; // 入口状态
        visitor.initGlobal(&in_state); // 初始化全局变量
        // 调用前向分析
        typename DataflowBBResult<ConstValueState>::Type out_state;
        compForwardDataflow(F, &visitor, &out_state, in_state); // 前向分析

        for (auto &BB : *F) { // 存储结果
```

```
        result[&BB] = out_state[&BB];
        }

        for (auto &BB : *F) { // 检查冗余
            if (BB.getName() == "obc_check") {
                check_redundant.push_back(&BB);
            }
        }
    }

    removeRedundant();
    return true;
```

3.4 常量集合合并函数 merge

merge 函数用于合并两个 ConstValueState 对象的状态。在常量传播分析中，合并操作通常发生在控制流图的分支点，多个前驱基本块的输出状态需要合并成一个输入状态。以下是 merge 函数的工作流程：

1. 遍历 src 的 cvmap：
   - src 是源状态，dest 是目标状态。
   - 遍历 src 的 cvmap，对于每个键值对(key, value)：
     - key 表示变量或指令。
     - value 表示该变量或指令的常量集合。
2. 检查 key 是否在 dest 的 cvmap 中：
   - 如果 key 不在 dest 的 cvmap 中：
     - 如果 value 是 top，在 dest 的 cvmap 中创建一个新的 KSet，标记为 top。
     - 否则，在 dest 的 cvmap 中创建一个新的 KSet，拷贝 value 的内容。
   - 如果 key 已经在 dest 的 cvmap 中：
     - 如果 dest 中对应的 KSet 或 value 是 top，则将 dest 中对应的 KSet 标记为 top，并清空其常量集合。
   - 否则，将 value 的常量集合插入到 dest 中对应的 KSet 的常量集合中。

该函数源码如下：

```
void ConstantPropagatorVisitor::merge(ConstValueState *dest, ConstValueState *s
rc) {
    // 合并两个ConstValueState
```

```
    // 遍历src的cvmap
    for (auto it = src->cvmap.begin(); it != src->cvmap.end(); it++) {
        llvm::Value* key = it->first; // key
        KSet* value = it->second; // value
        if (dest->cvmap.find(key) == dest->cvmap.end()) {
            // 如果key不在dest的cvmap中
            if (value->top) { // 如果value是top
                dest->cvmap[key] = new KSet(); // 将key加入dest的cvmap
                dest->cvmap[key]->top = true; // key的KSet是top
            } else { // 否则
                dest->cvmap[key] = new KSet(*value); // 将key加入dest的cvmap
            }
        } else { // 否则
            if (dest->cvmap[key]->top || value->top) {
                // 如果key的KSet或value是top
                dest->cvmap[key]->top = true; // key的KSet是top
                dest->cvmap[key]->const_vals.clear(); // 清空key的KSet
            } else { // 否则
                dest->cvmap[key]->const_vals.insert(value->const_vals.begin(),
value->const_vals.end()); // 将value的常数加入key的KSet
            }
        }
    }
}
```

3.5 指令常量传播分析函数 compDFVal

compDFVal 函数在常量传播分析中用于计算给定指令的常量传播数据流值。它处理不同类型的指令，包括二元操作指令、比较指令、加载指令、存储指令和 GetElementPtr 指令。通过分析指令的操作数和操作类型，compDFVal 函数更新数据流状态，确保常量传播分析能够正确进行并优化代码。

对于不同类型指令的处理，概括如下：

1. 二元操作指令
   - 获取二元操作指令的两个操作数。
   - 如果操作数是常量，将值插入到对应的 KSet 中；否则从 cvmap 中获取 KSet。
   - 如果任一操作数的 KSet 是 top，则结果 KSet 也是 top。
   - 否则，根据二元操作符计算结果，并将结果插入到结果 KSet 中。
   - 将结果 KSet 存储在 cvmap 中。

2. 比较指令
   - 获取比较指令的两个操作数。
   - 如果操作数是常量，将值插入到对应的 KSet 中；否则从 cvmap 中获取 KSet。
   - 如果任一操作数的 KSet 是 top，则结果 KSet 也是 top。
   - 否则，根据比较操作符计算结果，并将结果插入到结果 KSet 中。
   - 将结果 KSet 存储在 cvmap 中。
3. 加载指令
   - 获取加载指令的指针操作数。
   - 如果指针在 cvmap 中，获取对应的 KSet 并设置结果 KSet。
   - 如果指针是全局变量，获取其初始值并设置结果 KSet。
   - 否则，将结果 KSet 标记为 top。
   - 将结果 KSet 存储在 cvmap 中。
4. 存储指令
   - 获取存储指令的指针和值操作数。
   - 如果指针在 cvmap 中，获取对应的 KSet；否则，创建一个新的 KSet 并存储在 cvmap 中。
   - 如果值是常量，将其值插入到指针的 KSet 中，并将 KSet 标记为非 top。
   - 如果值在 cvmap 中，获取对应的 KSet 并更新指针的 KSet。
   - 否则，将指针的 KSet 标记为 top 并清空其常量集合。
5. 获取元素指针指令
   - 获取 GEP 指令的索引和指针操作数。
   - 如果索引或指针是常量，将值插入到对应的 KSet 中；否则从 cvmap 中获取 KSet。
   - 如果任一操作数的 KSet 是 top，则结果 KSet 也是 top。
   - 否则，计算结果的常量集合并存储在结果 KSet 中。
   - 将结果 KSet 存储在 cvmap 中。

4 实验问题

ConstantPropagator.cpp 的 compDFVal 函数中，还存在其他类型的指令需要进

行处理。但由于时间原因，笔者目前仅完成了 3.5 节所述指令的处理，这导致后面涉及到函数调用的测例无法通过。

compDFVal 函数中，对于某些类型指令的处理的 if 语句，笔者的实现还存在商榷。

另外，某些算法处理逻辑可能需要笔者在实验框架外做适当的添加。但是笔者仅在必要的 TODO 部分进行了完善，其他需要调整的部分还没有做恰当地处理。

5 实验总结

在本次实验中，笔者深入学习了数据流分析的概念，并掌握了常量传播算法的实现。通过设计和实现前向数据流分析框架，笔者成功地在 LLVM IR 中识别并移除了冗余的 OBC 检测代码。通过本次实验，笔者不仅熟悉了 LLVM Pass 的使用和编写方法，还加深了对数据流分析和常量传播算法的理解，为后续的编译器优化研究打下了坚实的基础。