Project 2 Report

Di Huang, Zhe Wang, Xiangyu Wang

December 11, 2015

1 Abstract

This project is aimed to implement a data flow analysis framework, and we can use this framework to implement several analysis and optimization. In this project, we design and implement four kind of analysis and one generic worklist algorithm used for all four analysis. Besides, we write up for Passes and some test cases to test the analysis. Output logs from the experiments show that our program is correct and efficient.

2 Overview

This project is to help us better understand the framework of LLVM. LLVM is a compiler infrastructure designed to be a set of reusable libraries with well-defined interfaces. In this framework, we can write analysis functions and do optimization based on this framework. LLVM is written in C++ and it is designed to do compile-tine, link-time and runtime optimization.

In our project, we do the following for analysis: Common Subexpression Elimination(CSE) analysis, May Pointer(MAYP) analysis, Constant Propagation(CP) analysis and also the Range analysis. In order to run multiple analysis, we have to define a generic framework for all analysis. In our program, we design four parts: Lattice, FlowFunction, Worklist and Pass. There is only one worklist but you can implement your lattice and flow functions to do analysis. Based on the lattice and flow function you implement, you can write passes to use these analysis functions and get the final output.

During our implementation, we find some interesting part of llvm. We find that all llvm IR code has been rewritten as SSA that we can directly use. For CSE and Range analysis, we can use the mem2reg optimization to do the pre-optimisation. Mem2reg pass has already removed instructions like 'alloca' and 'load'. It has already done the constant propagation and pointer optimization for us, which greatly simplifies our analysis for CSE and Range. However, we must not use the mem2reg pass in CP and MAYP analysis because 'alloca', 'load' and 'store' are important instructions in our analysis.

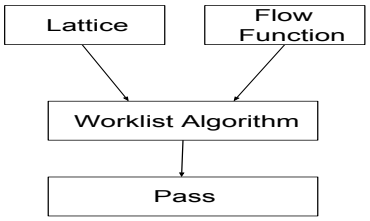

Figure 1: Project Framework

3 Interface Design

As shown in Figure. 1, there is one worklist algorithm, it takes the Lattice and Flow Function as input. It then get the output of a map, where the key is each instruction in the program and the value represents the input Lattice Node. The pass will invoke the worklist algorithm and get the analysis result.

Here the worklist algorithm is the core part in our project. We can never write a worklist algorithm for each analysis, which is too redundant. To avoid such kind of complexity, we design base classes for both Lattice and Flow Function. Anyone who want to run the worklist algorithm must write lattices and flow functions inherited from the base class.

3.1 Lattice

The base class in our project is called *LatticeNode* defined in LatticeNode.h. For convenience, all class members are public. There are 3 member variables in the class: *isTop*, *isBottom*, *type*. *isTop isBottom* respectively indicates whether this node is top or bottom node. *type* indicate the type of the lattice. Since we have four types of lattice, we define a enum variable called LatticeType containing four types: CPLATTICE, CSELATTICE, RALATTICE and MAYPLATTICE. Every subclass has to explicitly assign it's type.

We also define 3 virtual member functions in LatticeNode class.

- LatticeNode* join(LatticeNode* node). This function is rewritten by each lattice. For must analysis, it takes intersection operation. For may analysis, it takes union operation.
- bool equal(LatticeNode* node). This function takes one lattice node as input to compare whether it is equal to *this* lattice node.
- void print(). Print function is very important to get the standardized dump information. It can also be used as debug information. So we define it as the virtual function hoping that every kind of lattice node can output it's content.

3.2 Flow Function

Flow function has only one variable called *type* to indicate the flow function type. We define several virtual functions to be inherited in subclass.

- LatticeNode* operator()(Instruction* inst, vector¡LatticeNode*¿ input). We overload the operator (), which is used to provide convenient call API for worklist algorithm. The second parameter must be a vector since the merge node will have multiple input;
- void print(). It is used for standard output format.

3.3 Worklist Algorithm

As we mentioned above, the worklist algorithm takes the lattice and flow function as input and return a final map that represent each instruction's input lattice.

member variables

- map <Instruction*, LatticeNode*> output_map. This output map preserves each instruction's output lattice in the current state.
- map <Instruction*, vector<Instruction*>> predecessor. The actual control flow graph has been simplified by our definition of predecessor and successor. The predecessor means the predecessor instructions of the current instruction.
- map <Instruction*, vector<Instruction*>> successor. The successor map maps all of the successors of a instruction.
- map <Instruction*, LatticeNode*> finalMap. Final map is used to construct the final return of the worklist. It maps each input's converged input lattice node.
- queue<Instruction*> worklist. This worklist queue is used for monitoring whether any instruction need to be run again.

member functions

- void init(Function&, FlowFunction*, LatticeNode*). Every time the Run_Worklist function is called, the initialization function will be invoked for doing the following three things: empty all variables, construct predecessor and successor map.
- bool matchFlowFunc(Instruction*, FlowFunction*, LatticeNode*). matchFlowFunc is invoked by the Run_Worklist function. It first collect the instruction's all input lattice node and put it into a vector. Then it finds the corresponding FlowFunction type(using isa<>) and invoke the corresponding overloaded () operator defined in FlowFunction class.
- map<Instruction*, LatticeNode*> Run_Worklist(Function&, FlowFunction*, LatticeNode*). This function is the interface for Pass class. Pass can call the Run_Worklist function by taking the Instruction and a beginning node(usually the bottom node) as inputs and the function returns a finalMap.

3.4 Pass

Pass class is quite simple in our design. If we want to run any analysis in our framework, we just have to inherit the FunctionPass and invoke the Run_Worklist. The final map will be returned and automatically dumped after runing the pass.

4 Common Subexpression Elimination Analysis

4.1 Lattice

4.1.1 Definition

$(D, \sqsubseteq, \top, \bot, \sqcup, \sqcap) = (2^S, \supseteq, S, \emptyset, \cap, \cup)$ where $S = \{X \to E | X \in Var, E \in expr\}$

4.1.2 Implementation

The representation of each lattice node is simple. We implement the idea in CSELatticeNode class. **Member variable**

- map¡Value*, Instruction*¿ statements. This data structure is used to preserve every instruction's CSE information. The key is a Value* type to represent the current instruction and the value of that key represents the instruction that this current instruction points to.

Member function implementation

- Four types of constructors are implemented here: CSELatticeNode(), CSELatticeNode(bool bottom, bool top), CSELatticeNode(CSELatticeNode*), CSELatticeNode(CSELatticeNode).
- LatticeNode* join(LatticeNode* cseNode). This is the override of base class. Join here takes the intersection operation. Only the same information from two branches can be added in new lattice node.
- bool equal(LatticeNode*). This function returns true if two statements in lattice nodes are exactly same.

There is a tricky pitfall in each implementation inherited from basic class. If we would like to use the dyn_cast, isa functions provided, we need to implement one static **classof** function like other classes in llvm.

4.2 Flow Function

4.2.1 Definition

$$
\begin{aligned}
&F_{x:=y\text{op}z}(in) = in - \{x \to *\} - \{* \to x\} \cup \{x \to e | \\
&y \to e_1 \in in \wedge z \to e_2 \in in \wedge e = e_1 \text{op } e_2\} \\
&F_{x:=y}(in) = in - \{x \to *\} - \{* \to x\} \cup \{x \to e | y \to e \in in\}
\end{aligned} \tag{1a}
$$

4.2.2 Implementation

The implementation of CSE anaylsis is simplified by using the mem2reg pass. CSEFlowFunction inherits from two base classes: InstVisitor and FlowFunction. InstVisitor is used to override instruction vistors. All alloc and store instructions has been removed and $F_{X:=Y}(in)$ has been analyzed and replaced by constant. So we just need to implement the analysis for $F_{X:=Y \text{ op} Z}(in)$. To implement this flow function, we need to override *visitBinaryOperator*. In this function, we use the *Instruction→isIdenticalTo(Instruction*)* function to judge whether the two expression are the same expression.

5 May Pointer Analysis

5.1 Lattice

5.1.1 Definition

$D = 2^{\{x \to y \mid x \in \text{Vars} \wedge y \in \text{Vars}\}}$.
$\top = \{x \to y \mid x \in \text{Vars} \wedge y \in \text{Vars}\}$
$\bot = \emptyset$
$\sqsubseteq = \subseteq$
$\sqcup = \cup$
$\sqcap = \cap$

5.1.2 Implemention

Member variables

- map<Value*, set¡Value*>> point_to_info. It preserves each instruction's point-to information.
- set<Value*> point_to_all. This represents the pointers that points to anything. However, it is of no use in our analysis.

Member functions

- Four constructors as CSELatticeNode.
- join function inherited from LatticeNode. This function will do the union operation for lattice node. Every entry in the map and set will be added to a new LatticeNode.
- equal function inherited from LatticeNode.

5.2 Flow Function

5.2.1 Definition

$$
\begin{aligned}
&F_{x=k}(in) = in - \{x \to *\} \\
&F_{x=a+b}(in) = in - \{x \to *\} \\
&F_{x=y}(in) = in - \{x \to *\} \cup \{x \to z \,|\, y \to z \in in\} \\
&F_{x=\&y}(in) = in - \{x \to *\} \cup \{x \to y\} \\
&F_{x=*y}(in) = in - \{x \to *\} \cup \{x \to z \mid \exists y \to t \in in \wedge t \to z \in in\} \\
&F_{*x=y}(in) = \text{let } V := \{v | x \to v \in in\} \\
&S - V = \{v\}?v : \emptyset \cup \{v \to t | v \in V \wedge y \to t \in in\}
\end{aligned} \tag{2a}
$$

5.2.2 Implementation

In pointer analysis, we made the assumption that a pointer cannot be assigned to a constant value. So we do not deal with the first and second flow function in above definition.

Besides the member functions inherited from base class FlowFunction, we also implemented several visitor functions inherited from InstVisitor. There are several ways of implementation. One alternative is to deal with *alloca*, *load* and *store*. We can store all the information retrived from these instructions and analyze them based on the pre-store operation. The other alternative is to deal with *store* instruction. We can get the original varible of the two operand by several times cast and getOperand operation. This is easier way to analyze.
Member Functions

- void visitStoreInst(StoreInst I). This function can get two operands by getOperand(0) and getOperand(1). By analyzing the level of load operation in each operand and their level relationship, we can specify which kind of flow function it should apply.

6 Constant Propagation Analysis

6.1 Lattice

6.1.1 Definition

$(D, \sqsubseteq \top, \bot, \sqcup, \sqcap) = (2^S, \supseteq, S, \emptyset, \cap, \cup)$ where $S = \{X \to N | x \in Vars \text{ and } N \in Z\}$

6.1.2 Implementation

We implement the constant propagation in class CPLatticeNode, the detail information of that class is showed in figure 2. As figure 2 shows, class CPLatticeNode is inherited from class LatticeNode. In CPLatticeNode, we use a map data_info to store constant data information.

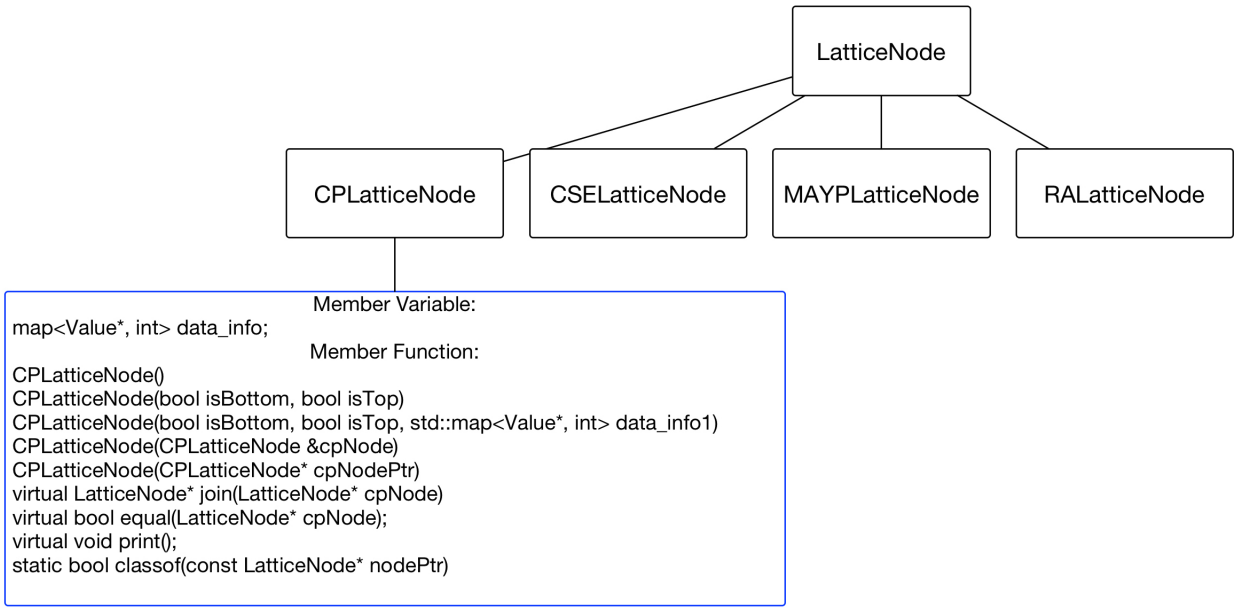

Figure 2: Constant Propagation Lattice Node

6.2 Flow Function

6.2.1 Classic Constant Propagation Flow Function

The classic flow functions introduced during lecture are listed as follows.

$$F_{x:=N}(in) = in - \{x \to *\} \cup \{x \to N\}$$
$$F_{x:=y\text{op }z}(in) = in - \{x \to *\} \cup \{x \to N | (y \to N_1) \in \text{in} \wedge (z \to N_2) \in \text{in} \wedge N = N_1 \text{op } N_2\} \tag{3a}$$

6.2.2 Flow Function Implementation

In our project, we choose not to run the mem2reg beforehand in order acquire the constant information as much as possible. However, without running mem2reg pass, the constant assignment instruction will be transformed into 2 instructions: *load* and *store*. We'll discuss the detail algorithm in the next section, now, we'll show the flow function relationship figure, which is showed as figure 3.

Since there is some difference between the classic constant propagation flow function and real situation (e.g: the assignment instruction are divided into two machine instructions: load and store) we need to design our own constant propagation algorithm to be compatible with these changes, which leads us to the next section.

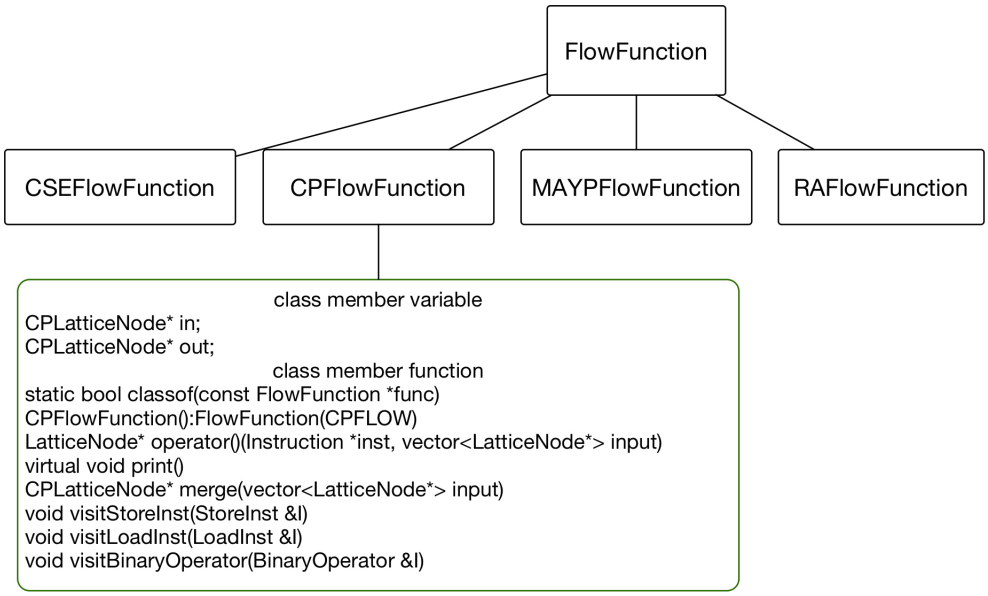

Figure 3: Constant Propagation Flow Function

6.2.3 Flow Function Algorithm

In this section, we'll show the primary flow function algorithms for constant propagation.

For the load instruction, first we delete the variable being loading in the instruction if it exists in input map. Then if the loading operand is a constant, we save it into the map, if not, we check if the operand already exists in the map, if true, we replace the loading operand with its corresponding value and save the loaded variable with the value into the map.

For the store instruction, first we erase the variable that is going to be stored from the map. Then check if the value operand is a constant, if it is, we save the value into the map. If the value operand is not a constant, find its value from the map, and store it into the map if it exists.

For the binary operator instruction, first we erase the temporary variable that is going to store the result value from the map. After that, check if the two operands are constant or variable that are stored in the map. If we can acquire the value of both operands, preform the corresponding binary operation on the two values, and store the result value to the map.

7 Range Analysis

7.1 Lattice

7.1.1 Definition

Suppose A's range is [a, b] and B's range is [c, d], the lattice is:
$D = \text{Vars} \to \mathbb{Z}^{\infty} \times \mathbb{Z}^{\infty}$
$\top(x) = [-\infty, \infty]$
$\bot(x) = \emptyset$
$\sqsubseteq = \subseteq$
$\sqcup = \cup$
$\sqcap = \cap$

7.1.2 Implementation

The implementation of range analysis lattice class is based on the common LatticeNode class. According to its lattice definition, we implemented the following member variables and functions.

Member variables

- RangeMap<Value*, ConstantRange*> The map is used to store the core information of the range analysis lattice which is a map from variables to their possible ranges. Based on this understanding, we looked through the class reference of LLVM to find a suitable data structure for both the variable and the range. We found a explanation that in LLVM, a variable is represented by the instruction who defined it and all the use of the variable will refer to the def-instruction. In addition, we noticed that LLVM recommends us not to use variable label to keep track of its value. Therefore, it's pretty natural for us to choose the pointer to instruction as the key of the map to represent the variable we want to analyze. As for the range, we found that the class ConstantRange can pretty much mimic behavior of our analysis target-"the range".

Member functions

- **Constructors: RALatticeNode()** We implemented several different constructors to provide different ways of class initialization in different uses.
- **Join: virtual LatticeNode* join(LatticeNode* raNode)** The join function is a functional implementation of the lattice's join. As is used in worklist algorithm, the join function is also a core function in the RALattice class. The operation of join is done by combining the two RangeMap in the two lattice node. For same keys between two map, we union their range, while for different keys between two map, we simply add all of them in to the final joined map.

$$
\begin{aligned}
F_{x:=y\ op\ z}(in) = in - \{x \to *\} \\
&\cup \{x \to (a,b) \mid y \to (a_y, b_y) \in in \\
&\wedge\ z \to (a_z, b_z) \in in \\
&\wedge\ a = min(a_y\ op\ a_z, a_y\ op\ b_z, b_y\ op\ a_z, b_y\ op\ b_z) \\
&\wedge\ b = max(a_y\ op\ a_z, a_y\ op\ b_z, b_y\ op\ a_z, b_y\ op\ b_z)\}
\end{aligned}
$$

$$
\begin{aligned}
F_{x:=y\ op\ C}(in) = in - \{x \to *\} \\
&\cup \{x \to (a,b) \mid y \to (a_y, b_y) \in in \\
&\wedge\ a = min(a_y\ op\ C, b_y\ op\ C) \\
&\wedge\ b = max(a_y\ op\ C, b_y\ op\ C)\}
\end{aligned}
$$

$$
\begin{aligned}
F_{x:=C\ op\ y}(in) = in - \{x \to *\} \\
&\cup \{x \to (a,b) \mid y \to (a_y, b_y) \in in \\
&\wedge\ a = min(C\ op\ a_y, C\ op\ b_y) \\
&\wedge\ b = max(C\ op\ a_y, C\ op\ b_y)\}
\end{aligned}
$$

$$
\begin{aligned}
F_{x:=C_1\ op\ C_1}(in) = in - \{x \to *\} \\
&\cup \{x \to (a,b) \mid a = b = C_1\ op\ C_2\}
\end{aligned}
$$

Figure 4: Range Anaylsis flow function

- **Equal: virtual bool equal(LatticeNode* raNode)** This function checks if two lattice node is equal. It's also used by worklist algorithem to determine whether or not to add subsequence instructions to the worklist. It's done by checking isTop isBottom lable as well as the RangeMap.
- **Print: virtual bool equal(LatticeNode* raNode)** This is used to output the final result.
- **CheckDifferences: std::vector<Value*> differInRange(RALatticeNode* in)** This is used to check the differences of two lattice node. This is used to help terminate the worklist algorithm for range analysis. Because of the lattice of range analysis is infinite high, we need a way to limit the number of iterations. Therefore, we record the times each flow function change the input and put a upper limit on it.

7.2 Flow Function

7.2.1 Definition

Figure. 4 illustrates the flow functions for range analysis

7.2.2 Implementation

In range analysis, we implemented flow function for two kind of operators, the binaryoperator and the PHI operator according to their abstract flow function definition. Note that we inherited **InstVistor** to automatically switch flow functions for different instructions.

Member Variables

- **Input and output: RALatticeNode* in; RALatticeNode* out** These two lattice node is used to store input and computed output information.
- **Counter: std::map<Value *, int> counter_map** This is used store how many times the flow function has change ranges of all variables. It works with RALatticeNode's diffInRange member function.

Member Functions

- **Opearator (): LatticeNode* operator()(Instruction *inst, vector<LatticeNode*> input)** This is the interface of flow function. It accepts input lattice node and instruction and distributes tasks to different visitXXXOperator functions which do the actual work. It's also responsible for limiting the times range can be changed.
- **Binary Operator function: void visitBinaryOperator(BinaryOperator I)** This is the actual function which do the work of processing input lattice node and ouput the result lattice node. It's for binary operator, add, sub, multiply and unsign div, as well as left binary AND, OR, LShr, Shl. The process is done by finding possible range of result of binary range. It's acutually done by the ConstantRange's sevral methods. They take care of it very well.
- **PHI node function: void visitPHINode(PHINode PHI)** This actually process the PHI node in SSA. As we use the mem2reg pass before doing our own analysis. The PHI node appears a lot in the IR code. We get the final range by union the ranges of the two candidates in the node.

8 Optimization

Due to time limitation, we do not implement the transformation part. We just present our idea on how to do it.

In these four types of analysis, it is easy to implement the optimization with the combination of CP analysis and CSE analysis. For example, for code in the following:

```
1 int main() {
        int a = 1;
3       int b = 2;
        int c = a+b;
5       int d = c;
        int e = d + 1;
7       return 0;
}
```

Listing 1: Sample Code

We can get c = 3 with constant propagation. Then we can remove d = c and replace e = d+1 with e = c+1. All of these can be done with IRBuilder insert and delete functions.

For the range analysis, it is more useful for checking whether there are exceptions with each variable. Our analysis program output each variable's range and can be used to check whether the variable or the array indicator is out of bound.

9 Discussion

In the constant propagation analysis, we used a (Value*, ConstantInt*) map at first time, where we used the a pointer of ConstantInt to store the constant value. For implementing load and store instruction, this is completely OK. However, for binary operation, after computed the result value, we need to check the entire map to find an element whose constant value information is equal to the result. However, iterating over the entire map takes too much time, thus we change the above map into a (Value*, int) map, and if we compute a new result value what we need to do is just insert the value into the map.

In two must analysis, we did not implement the PHI node visitor function. Some information are lost from our analysis. Our programs are too simple to check the correctness after information loss. From this aspect, we do not know whether our analysis is right or wrong.

10 Conclusion/Challenges

In this project, we only implement 3 operations: (ADD, SUB, and MUL) for integers. In the future, it is also possible to design and implement particular algorithms and data structures to compute and store float point constant information. One possible way is to use string to represent float values and design algorithms for string number calculating.