CSE 231 Project Final Report

Qingyu Zhou[*], **Mengting Wan**[†], **Peixin Li**[‡]
Computer Science and Engineering Department
University of California, San Deigo
[*]qyzhou@ucsd.edu, [†]m5wan@ucsd.edu, [‡]pel052@eng.ucsd.edu

1. INTRODUCTION

In this project, we designed an interface and implemented a dataflow analysis framework. Using this framework, we achieved several analysis and optimizations, which includes Constant Propagation Analysis, Available Expressions Analysis, Range Analysis and Intra-procedural Pointer Analysis. We also proposed several benchmark examples and apply our dataflow analysis on these examples. In this report, we will first introduce our interface, then we will introduce the implementation of above analysis and optimizations case-by-case.

Based on the interface, the source files for each analysis consists of two analysis files (".cpp" and ".h"), two analysis flow files (".cpp" and ".h") and a pass file (".cpp"). Here we also introduce our result structure. Outputs from our project consists of three parts:

Guardian. Guardian is the analysis for the program. For example, for pointer analysis, it will run Anderson algorithm based on IR of code.

Artisan. Artisan is a wrapper for Guardian result. It will display in text or graphic representation. For example, in pointer analysis, it gives the the graphic representation of the pointer relation with arrows.

Miner. Miner is the optimizer. According to the GuardianâĂŹs result, it does the optimization on the source file and output optimized file.

A glimpse of the workflow is showed in Figure 1.

2. INTERFACE DESIGN

2.1 Lattice

The very first for this project is how to build a easy-understood and easy-implemented lattice. To achieve it, we constructed three intuitive classes:

- **LatticeNode**: the node in the lattice;
- **LatticeEdge**: the edge connecting each node;
- **Flow**: the information stored in each edge.

2.1.1 LatticeNode

Class of `LatticeNode` is showed as below:

```
class LatticeNode {
public:
    int index;
    std::vector<LatticeEdge *> incoming;
    std::vector<LatticeEdge *> outgoing;
    Instruction * inst;
}
```

Each Node may point to may other node and may also be pointed by many nodes. Hence, we use two vector to denote the incoming edges and outcoming edges. The index is unique, which used to distinguish each Node.

2.1.2 LatticeEdge

Class of `LatticeEdge` is showed as below:

```
class LatticeEdge {
public:
    Flow * flow;
    LatticeNode * src;
    LatticeNode * dst;

};
```

For each LatticeEdge, we need to store which LatticeNode points to which LatticeNode. Two pointers src and dst help us to save this info. The information of each analysis is store in the flow. The flow is base class for any other subclass of flow which store different kind of information for the analysis.

2.1.3 Flow

Class of `Flow` is showed as below:

```
class Flow {
 public :
   // LatticeBase * base;
   //TriState is the world of lattice
   //Bottom =1 , TOP 2 or NOT Bottom and TOP 0
   int triPoint;
   virtual bool equals(Flow* other);
   //copy Flow
   virtual void copy(Flow *rhs);
   //join flow
   virtual Flow* join(Flow* other);
   Flow();
   Flow(int triPoint);
   Flow(Flow* flow);
   virtual ~Flow();
};
```

Flow class is tricky. It stores information of our analysis. It also must be a base class for our further analysis subclass because the analysis for pointer, Constant propagation and others are very different and we still needs unified worklist and lattice representation. However, the only thing each

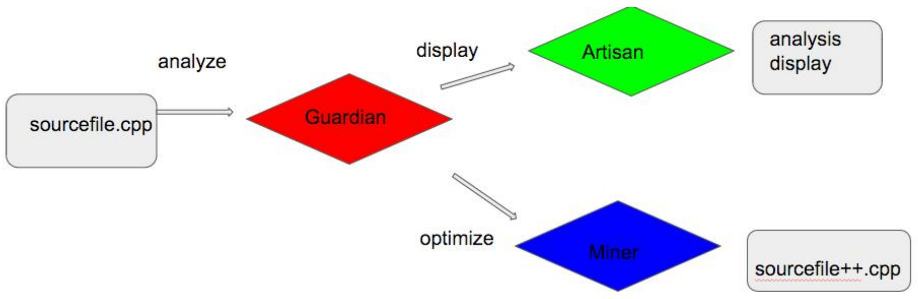


Figure 1: Workflow of this project.

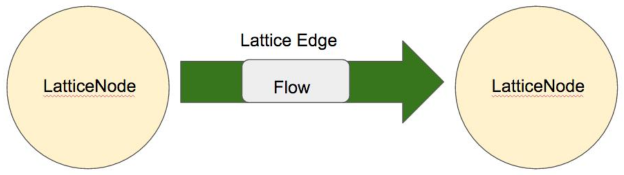


Figure 2: Structure of the interface.

analyses are in common is that their flow will have BOTTOM or TOP status, which is either empty or full set. If the flow is not BOTTOM or TOP, then it must contain some specific analysis info. In addition, each flow will have comparision and join operation. We can define a general interface function to let subclass override the join and comparison in base flow. In addition, the flow also needs to get a copy of input flow information. Otherwise, we need to create a new object on heap and assign the new combined information manually each time. Why bother?

When we start to build our Flow class, we think we need to build a LatticeBase class first. Because we need to store the BOTTOM and TOP info which is different from other value during the analysis. However, it is so painful and meaningless because only thing we want to check is that whether the flow is FLOW or BOTTOM. Then we realize that empty set and full set is no different. Indeed, empty set of pointer relation is different from empty set of Constant information because they are on different set. However, because we are not interested in these info, we can unify our TOP and BOTTOM to an abstract concept. Hence, we introduce triPoint here.

triPoint is amazing, we set $triPoint = 1$ if BOTTOM; $triPoint = 2$ if TOP; $triPoint = 0$ otherwise.

The very reason we set these specific values is that we can directly use triPoint to do condition decision.

- If (triPoint) true, we know we get a BOTTOM or TOP, we can do the fast basic analysis on input flow.
- If (triPoint) false, we immediately know that we are going to handle the value in dataflow.

The equal, join and copy function here should not be called, they are simply a interface left for subclass to implement it. Just in case, if some function miscalls these functions, they will print “you should not pass” to give user prompt to the error.

Mathematical notation(flow is just baseclass, no value but only B or T involved):
$F_{out} = (input == BOTTOM \&\& input2 == BOTTOM) : BOTTOM?TOP$

2.2 Control Flow Graph(CFG)

LLVM has its own control CFG, which links each instruction by each instruction. However, because we are now using our own lattice to do the analysis. So we need to find a way to copy the info in the LLVM to our own lattice structure. Well, how to do it? The CFG is a basically a directed graph. So this a classic graph clone problem. Usual solution is run the BFS or DFS on it. We choose DFS there. Traditionally, we need to call the DFS function recursively. However, we find that ,in fact, a stack is enough to do the DFS[1]. We keep looking for what is connected to the this instruction until we cannot find next one. Now we have a lattice ready for further analysis.

Code Implementation. The task was made by `CFGmaker()` We need to pay attention here is that we need to input the first edge for first Node. Because there is no input for root, but we start flow analysis from the input to the root. Hence, we need to add an edge for the root as our beginning input container for our all analysis.

During the CFG building, a counter will add the index to each Node to help us differentiate each node later.

2.3 Worklist

For worklist, we use following algorithm:

```
while (worklist.empty.not) do
  let n := worklist.remove_any;
  let info_in := m(n.incoming_edges);
  let info_out := F(n, info_in);
  for i := 0 .. info_out.length-1 do
    let new_info := m(n.outgoing_edges[i])
                    U info_out[i];
    if (m(n.outgoing_edges[i]) notEqualTo new_info])
      m(n.outgoing_edges[i]) := new_info;
      worklist.add(n.outgoing_edges[i].dst);
```

[1]clone graph, `http://bangbingsyb.blogspot.com/2014/11/leetcode-clone-graph.html`

Code Implementation. We use a queue here to go through the LatticeEdge vector. (we choose to use Edge. Because in this algorithm, it uses edge and all information flow is also stored in the Edge.) The difference is that we initialize the node by calling function `initialize()`. The reason we want to do this is that we hope we allow subclass has chance to choose which it wants to initialize with, BOTTOM or TOP.

we will call `executeFlowFucntion()`, which is the essence of the worklist algorithm. This function takes the current instruction and current input flow. This function will be overridden by subclass. It will decide adding or removing what kind of information to the current flow. If the output flow is changed, then it will be added back to the queue. The algorithm will not stop until the queue is empty. In another word, the program reaches the fixing point.

3. CONSTANT PROPAGATION

Constant propagation refers to the constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.

3.1 Lattice

From the lecture, we can know the constant propagation analysis is a "must" analysis. The lattice can be constructed as below:

- $\mathcal{D} = powerset(\{X \to C \mid \forall X \in Vars \;\&\; C \in \mathbb{Z}\})$;
- $\top = \{X \to C \mid \forall X \in Vars \;\&\; C \in \mathbb{Z}\}$;
- $\bot = \emptyset$;
- $A \sqcup B = A \cap B$;
- $A \sqcap B = A \cup B$;
- $A \sqsubseteq B = A \supseteq B$.

3.2 Flow Function

In our project, we need to realize the bellowing functions:

- $F_{X=YopZ}(in) = (in - \{X \to *\}) \cup \{X \to C \mid Y \to C_1 \in in, Z \to C_2 \in in, C = C_1 op C_2\}$;
- $F_{(X==C)true-branch}(in) = (in - \{X \to *\}) \cup \{X \to C\}$;
- $F_{(X==C)false-branch}(in) = in$;
- $F_{merge} = (in_1, in_2) = in_1 \cup in_2$.

3.3 Implementation

Our Constant Propagation class was based on a map, which uses key to store the name of the register and a float to store the value of the constant. We use the same expression of TOP and BOTTOM as the Flow Class, which is stored in a int variable triPoint. The most important part of the function is the join method.

First, we initialized the input to the BOTTOM. If either of the two input flows is TOP (empty set), we will join the two input flows as top. If one of the flows is BOTTOM (full set), the join result will be the smaller set. Or if the two flows are neither TOP or BOTTOM, then we will have the merge the two input sets. The most difficult part when we do constant propagation is how to deal with the store, alloca and load instruction. We first tried to return to the top when we meet these instructions. However, we found the join result is not correct if we do so. So we try to and the value to the flow when we meet the instruction store. There is still a problem about how to deal with the information we get from the store instruction, but we make a improvement in Range Analysis and rewrite the `runStoreInst(RAFlow* in, Instruction* inst)` function.

The implementation of our constant propagation analysis is shown as below:

```
class ConstantPropAnalysisFlow: public Flow {
 class ConstantPropAnalysis : public WorkList {
   public :
    ConstantPropAnalysis(Function &F);
    Flow* executeFlowFunction(Flow *in,
                              Instruction *inst,
                              int NodeId);
        Flow* initialize();
        void print(raw_ostream &OS);
        void printHelper(raw_ostream &OS,
                             LatticeNode* node);
   protected:
        ConstantPropAnalysisFlow *runCastInst(
             ConstantPropAnalysisFlow* in,
             Instruction* inst);
        ConstantPropAnalysisFlow *returnTop();
        ConstantPropAnalysisFlow *runFOpInst(
              ConstantPropAnalysisFlow* in,
              Instruction* inst, unsigned opcode);
        ConstantPropAnalysisFlow *runOpInst(
              ConstantPropAnalysisFlow* in,
              Instruction* inst, unsigned opcode);
        ConstantPropAnalysisFlow *runPhiInst(
               ConstantPropAnalysisFlow* in,
               Instruction* inst);
  public:
        float computeOp(float leftVal,
                  float rightVal, unsigned opcode);
};
```

4. AVAILABLE EXPRESSIONS

available expressions is an analysis algorithm that determines for each point in the program the set of expressions that need not be recomputed. To be available on a program point, the operands of the expression should not be modified on any path from the occurrence of that expression to the program point. [2]

4.1 Lattice

The available expression analysis is also a "must" analysis. And in this part, we will construct the lattice of available expression analysis almost the same as constant propagation analysis:

- $\mathcal{D} = powerset(\{X \to E \mid \forall X \in values \;\&\; E \in instructions\})$;
- $\top = \{X \to E \mid \forall X \in values \;\&\; E \in instructions\}$;
- $\bot = \emptyset$;
- $A \sqcup B = A \cap B$;
- $A \sqcap B = A \cup B$;
- $A \sqsubseteq B = A \supseteq B$.

4.2 Flow Function

In our project, we need to implement the bellowing functions:

[2] `https://en.wikipedia.org/wiki/Available_expression`

- $F_{X=YopZ}(in) = in \cup \{X \rightarrow W \mid \exists W \rightarrow E \in in, E = YopZ\} \cup \{X \rightarrow YopX \mid \forall WW \rightarrow E \in in, E \neq YopZ\}$;
- $F_{(*)true-branch}(in) = in$;
- $F_{(*)false-branch}(in) = in$;
- $F_{X=\phi(Y,Z)(in_1,in_2)} = (in_1 \cap in_2) \cup \{X \rightarrow E \mid Y \rightarrow E \in in_1 \& Z \rightarrow E \in in_2\}$;
- $F_{X=cmp(Y,Z)} = in$.

4.3 Implementation

Some parts of the available expression analysis is almost the same as constant propagation. For example, we still use the map to keep the value and its type. Using SSA (static single assignment form), which helps us find invariant computations in loop, makes this part easier. In order to get the correct result from our the implement of this part, we should deciding the phi nodes that have arbitrary numbers of arguments and join the two flows. The join method is almost the same as before. The implementation of our available expression analysis is shown as below:

```
class CSEAnalysis: public WorkList {
 public:
    CSEAnalysis(Function &F);
    Flow* executeFlowFunction(Flow *in,
                     Instruction *inst, int NodeId);
    Flow* initialize();
    void print(raw_ostream &OS);
    oid printHelper(raw_ostream &OS,
                    LatticeNode* node);
 protected:
    CSEAnalysisFlow *runUnaryInst(
            CSEAnalysisFlow* in, Instruction* inst,
            unsigned opcode);
    CSEAnalysisFlow *runFOpInst(
            CSEAnalysisFlow* in, Instruction* inst,
            unsigned opcode);s
    CSEAnalysisFlow *runOpInst(
            CSEAnalysisFlow* in, Instruction* inst,
            unsigned opcode);
    CSEAnalysisFlow *runPhiInst(
            CSEAnalysisFlow* in, Instruction* inst);
 private:
    string computeBinaryOp(string leftVal,
               string rightVal, unsigned opcode);
    string computeUnaryOp(string leftVal,
                     unsigned opcode);
    bool isEqual(CSEAnalysisFlow* in, Instruction *inst);
    map<string, string> analysisMap;
};
```

5. RANGE ANALYSIS

In this part, we will implement range analysis to support warning programmer if cannot show array access in bounds. The general idea is to track variables' ranges and update them by statement, which is a "may" analysis. However, in practical, we need to track these variables' ranges by instructions, which provides a challenge for our implementation.

5.1 Lattice

In this part, a map is used describe the range $[\min(X), \max(X)]$ of a variable X as follows:

$$m : X \rightarrow [\min(X), \max(X)],$$

where $\min(X)$ and $\max(X)$ indicate the min and max possible values of variable X. Suppose $Vars$ denotes the set of all variables in the program. Then we have following lattice

- $\mathcal{D} = powerset(\{X \rightarrow [\min(X), \max(X)] \mid X \in Vars\})$;
- $\top = \{X \rightarrow \emptyset \mid X \in Vars\}$;
- $\bot = \{X \rightarrow [-\infty, \infty] \mid X \in Vars\}$;
- $A \sqcup B = \{X \rightarrow [\min(a_1, b_1), \max(a_2, b_2)] \mid X \in Vars, m_A(X) = [a_1, a_2] \& m_B(X) = [b_1, b_2]\}$,
  where $m_A(X)$ indicates the map defined in A;
- $A \sqcap B = \{X \rightarrow [\max(a_1, b_1), \min(a_2, b_2)] \mid X \in Vars, m_A(X) = [a_1, a_2] \& m_B(X) = [b_1, b_2]\}$
- $A \sqsubseteq B \Leftrightarrow \forall X \in Vars, m_A(X) \subseteq m_B(X)$.

Notice that the height of range is infinity. Therefore, we need to design a termination condition to ensure that our worklist algorithm can stop. Particularly, we set a threshold $thr = 3$. When we detect that some nodes have been visited more than thr times, we identify it as a loop and jump out of it.

5.2 Flow Function

We implemented our range analysis for following C++ operations: "+", "-", "*", "/", "%", ">>" and "<<". Specifically, our flow functions are showed as below:

- $F_{X=C}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow [C, C]\}$;
- $F_{X=Y}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow m_{in}(Y)\}$;
- $F_{X=Y+X}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow [a_1 + b_1, a_2 + b_2]\}$
  where $Y, Z \in in$ and $m_{in}(Y) = [a_1, a_2]$, $m_{in}(Z) = [b_1, b_2]$;
- $F_{X=Y-X}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow [a_1 - b_1, a_2 - b_2]\}$
  where $Y, Z \in in$ and $m_{in}(Y) = [a_1, a_2]$, $m_{in}(Z) = [b_1, b_2]$;
- $F_{X=Y*X}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow [\min(\mathcal{T}), \max(\mathcal{T})]\}$
  where $\mathcal{T} = \{a_1 * a_2, b_1 * a_2, a_1 * b_2, b_1 * b_2\}$, $Y, Z \in in$ and $m_{in}(Y) = [a_1, a_2]$, $m_{in}(Z) = [b_1, b_2]$;
- $F_{X=Y/X}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow [-\infty, \infty]\}$, if $a_2 b_2 < 0$;
  $F_{X=Y/X}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow [\min(\mathcal{T}), \max(\mathcal{T})]\}$, otherwise,
  where $\mathcal{T} = \{a_1/a_2, b_1/a_2, a_1/b_2, b_1/b_2\}$, $Y, Z \in in$ and $m_{in}(Y) = [a_1, a_2]$, $m_{in}(Z) = [b_1, b_2]$;
- $F_{X=Y\%X}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow [\min(\mathcal{T}), \max(\mathcal{T})]\}$
  where $\mathcal{T} = \{a_1\%a_2, b_1\%a_2, a_1\%b_2, b_1\%b_2\}$, $Y, Z \in in$ and $m_{in}(Y) = [a_1, a_2]$, $m_{in}(Z) = [b_1, b_2]$;
- $F_{X=Y>>C}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow [\min(\mathcal{T}), \max(\mathcal{T})]\}$
  where $\mathcal{T} = \{a_1 >> C, b_1 >> C\}$, $Y, Z \in in$ and $m_{in}(Y) = [a_1, a_2]$, $m_{in}(Z) = [b_1, b_2]$;
- $F_{X=Y<<C}(in) = (in - \{X \rightarrow *\}) \cup \{X \rightarrow [\min(\mathcal{T}), \max(\mathcal{T})]\}$
  where $\mathcal{T} = \{a_1 << C, b_1 << C\}$, $Y, Z \in in$ and $m_{in}(Y) = [a_1, a_2]$, $m_{in}(Z) = [b_1, b_2]$;

5.3 Implementation

Notice that our implementation is based on instructions rather than statements. Thus we have some issues to handle. We need to use different functions to handle different kinds of instructions: 1) we use `runStoreInst(RAFlow* in, Instruction* inst)` to handle `store` instruction; 2) we use `*runLoadInst(RAFlow* in, Instruction* inst)` to handle `load` instruction and create a temporary variable to store this range; 3) we use `*runOpInst(RAFlow* in, Instruction* inst, unsigned opcode, int type)` to handle arithmetic operators showed in the previous section and may need to use the temporary variable created in previous `load` instruction; 4) we use ***runRtnInst(RAFlow* in, Instruction* instruction)** to handle **return**; 5) we use ***runPhiInst(RAFlow* in, Instruction* inst)** to handle PHI nodes.

The implementation of our range analysis class is showed as below. Here `getRange()` is used to implement the arithmetic operations we mentioned above.

```
class RangeAnalysis : public WorkList{
 public:
    RangeAnalysis(Function &F);
    Flow* executeFlowFunction(Flow *in,
                Instruction *inst, int NodeId);
    Flow* initialize();
    void print(raw_ostream &OS);
    void printHelper(raw_ostream &OS,
                         LatticeNode* node);

 protected:
    RAFlow *runCastInst(RAFlow* in,
                        Instruction* inst);
    RAFlow *runOpInst(RAFlow* in,
                      Instruction* inst,
                      unsigned opcode, int type);
    RAFlow *runPhiInst(RAFlow* in,
                       Instruction* inst);
    RAFlow *runStoreInst(RAFlow* in,
                         Instruction* inst);
    RAFlow *runLoadInst(RAFlow * in,
                        Instruction* inst);
    RAFlow *runCompInst(RAFlow* in,
                        Instruction* inst);
    RAFlow *runRtnInst(RAFlow* in,
                        Instruction* inst);
    map<int,node> nodeCount;

 public:
    Range getRange(Range leftRange, Range rightRange,
                   unsigned opcode);

};
```

We test our analysis on a benchmark example which all aboved operations are involved.

Limitation. Because of time limitation, we have successfully implement some operations but we didn't incorporate the condition checking or pointer analysis into our range analysis. If we can continue to develop this project, it would definitely be interesting to implement those analysis.

6. INTRA-PROCEDURAL POINTER ANALYSIS

Guardian. There are many ways to analyze pointer. There are flow sensitive or insensitive analysis. Also we have Andersen and Steensgaar. In order to avoid infinite lattice, we choose flow insensitive flow analysis here. The result of Steensgaar is not that precious, hence we choose to use Anderson here.

Here let's map our c++ instruction to the IR instruction here. We can find that all c++ operation related to the pointer can be mapped four types of IR instructions.
Type 1:

```
x = &y    store i32* %y, i32** %x, align 8
//Notice there is a subtype of this type:
x= NULL   store i32* null, i32** %a, align 8
```

Type 2:

```
*x =y        %0 = load i32** %y, align 8
             %1 = load i32*** %x, align 8
             store i32* %0, i32** %1, align 8
```

Type 3:

```
x=*y      %0 = load i32*** %y, align 8
          %1 = load i32** %0, align 8
          store i32* %1, i32** %x, align 8
```

Type 4:

```
x=y      %0 = load i32** %pb, align 8
         store i32* %0, i32** %pa, align 8
```

Type 5:

```
x=y+c  %arraydecay1 = getelementptr
           inbounds [2 x i32]* %y, i32 0, i32 0
       %add.ptr = getelementptr
           inbounds i32* %arraydecay1, i64 1
       store i32* %add.ptr, i32** %x, align 8
```

Type 5 is hidden boss. It is useful when we want to move the pointer. Especially when want array manipulation. However, the IR operation âĂIJgetelementptrâĂİ is so weird. Also, we can totally use array manipulation to avoid using pointer. Hence, we do not implement this. Also, in most case, there should be no two pointers points to the same array in most case. In general, optimization on x=y+c is rare. We may implement this in later version, but not now.

Now, we are going to handle how to let Guardian know what kind of pointer assignment it is handling. We notice that each operation has different order of load and store. Using this order difference, we can figure out what kind of operation we are dealing with.

x=*y and *x=y is tricky. Because there are one store following two store. The magic here is that we can find where the instruction in store has the variable. if two loads has variable name, then it is *x=y. If first load has variable name and the store also has variable name, then it is x=*y. Including this trick to our order analysis, we can easily classify each instruction to each type section.

Everything is ready. We can start to use these instructions to run our worklist. But, let's first do some math here.

6.1 Lattice

Similar to previous sections, we thus have following lattice:

- $\mathcal{D} = powerset(\{x \to y \mid x, y \in Vars\})$;
- $\top = \{x \to y \mid \forall x, y \in Vars\}$;
- $\bot = \emptyset$;
- $A \sqcup B = A \cup B$;
- $A \sqcap B = A \cap B$;
- $A \sqsubseteq B = A \subseteq B$.

6.2 Flow function

Then we have following flow functions

- $F_{x=\&y}(in) = in \cup \{x \to y\}$;
- $F_{x=NULL}(in) = in - \{x \to *\}$;
- $F_{x=*y}(in) = in \cup \{x \to v \mid y \to w \ \& \ w \to v\}$;
- $F_{*x=y}(in) = in \cup \{w \to z \mid x \to w \ \& \ y \to z\}$;
- $F_{x=y}(in) = in \cup \{x \to v \mid y \to v\}$.

6.3 Implementation

We have `PointerAnalysisFlow` which inherit from Flow. We add a map of sets here to store the opinter relation, called value.
`map<string, set<string> > value`
The `join()` and `equal()` in flow is nothing but check where two map are same or not. Only tedious iteration operation, we are not going to cover it here.

For PointerAnalysis, which is a subclass of the Worklist. As we mentioned before, we need to and only need to override the `executeFlowFunction()`. We use a function called `whoAmI()` to identity which type of pointer operation we are going to handle. The result will be passed to a switch case which calls each function handle each different operation. The flow function operation follows the matched representation we mentioned previously.

To get the order information, we can use `instrution->getNextNode()` to get next instruction. Using `isa<targetIntructionType>` and `inst->getopcode()` to get what type of we have. Using `isPointer()` and `isVariable()` to get where IR is operating on variable or temp variable.

Artisan. The output of our analysis result will be series of key pair. Like

```
#Edge incoming
pa->a
pb->b
pa->a
pb->b

#Edge outcoming
pa->a
pb->b
```

To get a more intuitive representation,

```
#define NULL 0

int main() {
        int a;
        int b;
        int c;
        int d;
        int * massive_p1;
        int x;
        int y;
        int z;
        int *crazy_p2;
        massive_p1 = $a;
        massive_p1 = $b;
        massive_p1 = $c;
        massive_p1 = $d;
        crazy_p2 = $x;
        crazy_p2 = $y;
```

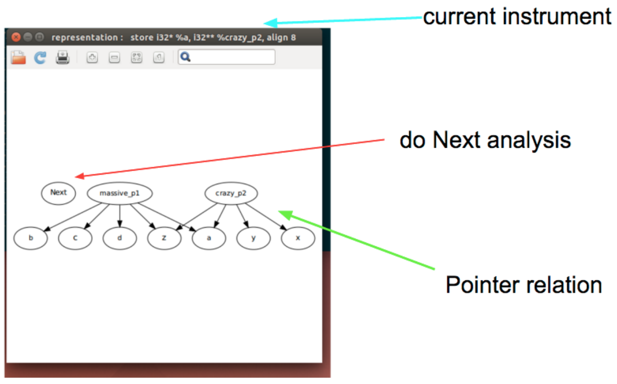

representation : store i32* %a, i32** %crazy_p2, align 8

Figure 3: Draw in Artisan.

```
        crazy_p2 = $z;

        massive_p1=$z;
        crazy_p2= $a;


        return 0;
}
```

After artisan, we can get kind of pretty diagram for our pointer output. We can even track the changes of the current. We call the draw in artisan, which will display the current value step by step. When you click **next**, it gives you the next stepâĂŹs DF analysis. The **title** of the window is the current instruction we analysis, which gives you a clear tracking info. The current version of artisan has a little problem when comes to the end of program because ending info is different from previous part. However, big name always has some special personality.We may improve him later.

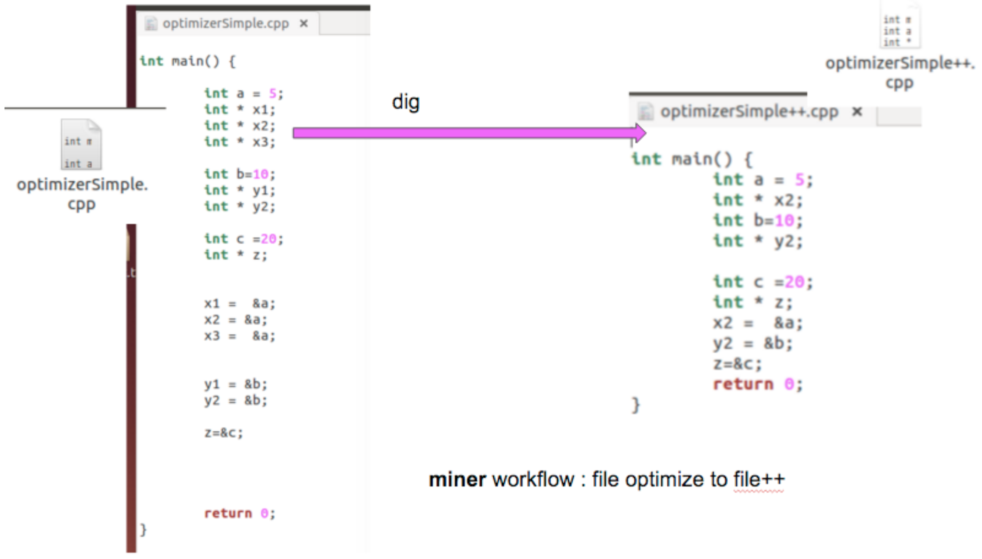
Code Implementation. In order to save life, we use python to implement artisan. The UI is based on gtk, which is runnable on the Ubuntu 14.04 with python 2.7.6. python 3.0 also works. The graph component is **xdot**. `xdot.py` is an interactive viewer for graphs written in Graphviz's dot language[3]. In fact, xdot also supports svg, which is used in LLVM to draw graph. However, to directly call its API based on the Guardian result.

Miner. We call our optimizer miner because it digs deeply into the source file and change anything he can changes.

FFor pointer analysis, miner will look for the final result of Guardian. Base on this info, it will replace the variable and expression in file. Here we have two solution.

- replace the source file;
- replace .ll file.

[3]`https://github.com/jrfonseca/xdot.py`



```
int main() {

        int a = 5;
        int * x1;
        int * x2;
        int * x3;

        int b=10;
        int * y1;
        int * y2;

        int c =20;
        int * z;


        x1 =  &a;
        x2 = &a;
        x3 =  &a;


        y1 = &b;
        y2 = &b;

        z=&c;



        return 0;
}
```

```
int main() {
        int a = 5;
        int * x2;
        int b=10;
        int * y2;

        int c =20;
        int * z;
        x2 =  &a;
        y2 = &b;
        z=&c;
        return 0;
}
```

Figure 4: Miner Workflow.

Well, both ways has their own advantage. However, we decide to work on source file because we can see the changes. Using .ll file, the only way to check whether miner has worked correctly is to check output of executable file. This is kind of detour to check the result. Hence, we choose to work on the cpp file directly. It generate a file which appends ++ to the name of origin file. I must admit that this is not a good choose. Because âĂIJit will be used before it has been made betterâĂİ. Just like C++, which should called ++C [4].

Suppose we have following example:

```
int main() {
        int a = 5;
        int * x1;
        int * x2;
        int * x3;

        int b=10;
        int * y1;
        int * y2;

        int c =20;
        int * z;
        x1 =  &a;
        x2 = &a;
        x3 =  &a;
        y1 = &b;
        y2 = &b;
        z=&c;
        return 0;
}
```

After optimization, we will get

```
int main() {
        int a = 5;
        int * x2;
        int b=10;
        int * y2;

        int c =20;
        int * z;
        x2 =  &a;
        y2 = &b;
        z=&c;
        return 0;
}
```

Code Implementation. The miner is implement in python. It keep a list of set. Each set will contains the pointer which has the exact same info. In another word, there are definitely the same. The python does regular expression on source file and replaces the source. It will loop on it again and again until it cannot do modification anymore. This algorithm has a problem. It will remove the same statement without any further checking. For example, we have

```
int * a = y;
int *b = y;
cout<<*a;
cout<<*b;
```

[4]C++ joke, http://www.nerdware.org/doc/coding.html

if we a and b only points to y, then the first loop we wil get

```
int *a = y;
int *a = y;
cout<<*a;
cout<<*a;
```

In second round, we will get

```
int * a = y;
cout<<*a;
```

We indeed remove redundant info about b. However, we also lost the second print. Miner is too hard working sometimes. This can be solved by adding some additional criteria on replacement. The work is trivial, we can work on it in further implementation.

7. CONCLUSION

In this project, we implemented Constant Propagation Analysis, Available Expressions Analysis, Range Analysis and Intra-procedural Pointer Analysis in this project. In addtion, we applied them on several benchmark examples and completed visualization for some analysis. We also notice that there are still some limitations in current version and we still can improve it if we have more time.