CS5218 Assignment2

Wang Jiadong (A0105703U)

Task1: Define analysis model using a monotone framework.

Difference Analysis is to determine the maximum difference of any variable pair in any program point. In order to conduct the difference analysis, we actually need to analyze the possible value range for each variable in all program points. Therefore, Difference Analysis is simply a variable value range analysis. Hence, we have to collect all the variable ranges for each basic block, and then do a pair-wise comparison to get the variable maximum difference.

- Flow (F).

Intuitively, difference analysis is a top-down(forward) analysis. We analyze the range for each variable from init(S*) to final(S*), and we adjust the variable value range for each instruction. However, for branching instructions, we do have to do some extra backward analysis, to get the entry value set for its successors.

- Starting Block(E).

Since the analysis flow is top-down (forward), then it is very clearly the starting block is init(S*).

- Lattice and Partial Order

We use a lattice, with each node as an element the power set of variables, along with their intervals:

$$\mathcal{P}(\text{variable} \rightarrow \text{interval}),$$

where interval is defined as [lower boundary, upper boundary].

In this lattice, the bottom is an empty set ($\emptyset$), while the top is all variables with range [-infinity, +infinity]. Hence, the lattice is partially ordered by subset inclusion: $\sqsubseteq = \subseteq$

- Initial Value(ι).

Since there are no variables declared initially, we just define the initial value as an empty set ($\emptyset$). In LLVM, only after some instructions that declare new variables (e.g. ***Alloca***, ***Load***, etc.), shall we add the new variable, along with its interval, into the lattice. For instance, for "%1 = alloca i32, align 4", we add "%1 → [INFINITY_NEGATIVE, INFINITY_POSITIVE]" into the analysis result.

- May or Must.

As we are trying to get the maximum difference of each pair of variables, we are actually trying to find the "possible" range for each variable. So union function is used, instead of intersection. Hence, it is a "May".

- Monotonicity

As mentioned above, the top element of the lattice, is the complete set of all variables with range from [INFINITY_NEGATIVE, INFINITY_POSITIVE], while the bottom element is an empty set. Hence, for any pair of element "a" and element "b" in this lattice, if there is a path from a to b and a <= b (which is defined as that ***b*** contains all the variables that ***a*** has, and for each variable ***x*** that ***a*** contains, the corresponding interval in ***b*** is its super set), then ***a*** is a subset of ***b***.

- Transfer Functions.

Transfer function can be represented as:

$$f_\ell(l) = (l \backslash kill([B]^\ell)) \cup gen([B]^\ell) \ \textbf{where} \ [B]^\ell \in blocks(S_\star) \quad ,$$

where:

$$\text{Kill}[\ a = \exp\ ]^{\ell} = \{a \rightarrow \text{old\_interval}\}$$

$$\text{Gen}[\ a = \exp\ ]^{\ell} = \{a \rightarrow \text{new\_interval}\}.$$

If the expression causing a's range to be altered, then the exiting ***a*** value will be killed, and a new range will be created. Instructions, such as Add, Sub, Mul, Rem, load, and store, have this effect.

Task 2 & 3: Implementation of difference analysis.

- Variable Interval (***varInterval***) Object Declaration

First of All, in order to facilitate the analysis, I have defined a new class named as ***varInterval***. This class is an abstraction to describe the integer value range of a variable. Each variable is within the range of [lower, upper], while the two variables, namely lower and upper, are restricted within [INF_NEG, INF_POS].

INF_NEG and INF_POS denote the negative infinity and positive infinity. Intuitively, the lower and upper boundary of any variable is not allowed to exceed INF_NEG or INF_POS. Since in real life, it is impossible to compute and analyze infinity, I have manually defined INF_NEG as -1000, and INF_POS as 1000. Hence, if a variable's upper bound is more than 1000, e.g. 1001, its upper bound will be overwritten by 1000, representing INF_POS. Similar technique applies to the lower boundary. In other word, this class does not allow value exceeding its [INF_NEG, INF_POS] boundary.

During this interval analysis, we also require the concept as "Empty Set". Hence I have manually defined [INF_POS, INF_NEG] as Empty Set. If there is any variable's range [lower, upper], with lower > upper, we will overwrite it as [INF_POS, INF_NEG];

In order for better support, arithmetic operator functions such as adding, subtracting, multiplication, dividing are defined for ***varInterval***, and encapsulated inside the class.

```cpp
/**
 * This is a data structure for variable interval. variable is in [lower, upper];
 * Here we assume 1000 as INF_POS (positive infinity), and -1000 as INF_NEG(negative infinity)
 * No value shall be set outside this range.
 */
class varInterval {
private:
    //lower and upper boundaries
    int lower;
    int upper;
public:
    //artificial infinity values.
    const static int INF_POS = 1000;
    const static int INF_NEG = -1000;
    //constructors
    varInterval() {...}
    varInterval(int lower, int upper) {...}
    //getter
    int getUpper() {...}
    int getLower() {...}
    //setter
    void setLower(int lower) {...}
    void setUpper(int upper) {...}
    //check if varInterval A belongs to varInterval B.
    bool operator<=(varInterval v) {...}
    bool isEmpty() {...}
    //Arithmetic Operator Support for varIntervals.
    static varInterval add(varInterval a, varInterval b) {...}
    static varInterval mul(varInterval a, varInterval b) {...}
    static varInterval div(varInterval a, varInterval b) {...}
    static varInterval rem(varInterval a, varInterval b) {...}
    static varInterval sub(varInterval a, varInterval b) {...}
    //formatted printing
    void printIntervals() {...}
    std::string getIntervalString() {...}
};
```

Figure 1 Class varInterval cpp code.

- Variables:

Globally, we define two variables, as below:

```cpp
std::stack<std::pair<BasicBlock*, std::map<Instruction*, varInterval>>> traversalStack
std::map<BasicBlock*, std::map<instruction*, varInterval>> analysisMap
```

Figure 2 Global variables defined.

TraversalStack is the work-list, containing the basic blocks that we will have to process and analyze. For each basic block, the entry value set are also provided inside ***traversalStack***.

AnalysisMap is the output collector. It contains the range for each variable in each of the basic blocks, and it is updated for each iteration. Once the iteration is completed, we can conduct the pairwise difference analysis easily for each basic block using this ***analysisMap***.

- Algorithm Pseudo-Code.

Initialization

In order to initialize, not only we need to declare the variables, but also set the initial state for the traversal stack, which is the entry block with empty entry variable set.

```
std::map<BasicBlock *, std::map<Instruction *, varInterval>> analysisMap;
std::stack<std::pair<BasicBlock *, std::map<Instruction *, varInterval>>> traversalStack;

std::map<Instruction *, varInterval> emptySet;
traversalStack.push(std::make_pair(entryBB, emptySet));
```

Figure 3 Initialization Pseudo Code.

Iteration

As long as the traversal stack is not empty, we will keep iterating this loop.

Inside the loop, we remove the first basic block from the stack, and analyze it accordingly. The analysis will update its analysis map, provide us with its successors, as well as the update status. If the block's exit value is altered, then the successors will be push into the stack, otherwise, they are ignored.

```
while(traversalStack is not empty){
    [currentBlockPtr,  currentBlockEntryVarSet] = traversalStack.pop();
    bool changed = analyzeBlock(currentBlockPtr, currentBlockEntryVarSet, currentBlockExitVarSet, SuccessorsMap);
    if(changed){
        //push all successors into traversalStack
        traversalStack.push(SuccessorsMap);
    }

}
```

Figure 4 Iteration Pseudo Code.

Block Analysis.

Block analysis is basically to analyze each instruction sequentially, and update the current block exit variable set. However, if the instruction is conditional branch, we have to find the successors, analyze the comparison instruction, and analyze backward accordingly. The backward analysis results are pushed into successor Map and returned into the main function.

```
analyzeBlock(currentBlockPtr, currentBlockEntryVarSet, &currentBlockExitVarSet, &SuccessorsMap){
    for (all instructions inside currentBlock){
        if( sub || add || mul || rem ){
            /*calculate and update variables*/
        }else if(load || restore || alloca ){
            /*create new variable */
        }else if(ret || cmp){
            /*nothing to do, just ignore*/
        }else if(br && currentBlockExitVarSet Updated){
            /*if unconditional
            SuccessorsMap.push([brInstruction.getSuccessor, currentBlockExitVarSet]);
            //if conditional
            sucessors = analyzeComp(CompIns, BranchInstruction);
            for(each successor in successors){
                analyzeBlockBackward(currentBlockPtr, currentBlockExitSet, successorEntryVarSet);
                SuccessorsMap.push([successor, successorEntryVarSet]);
            }
        }else{
            //ignore
        }
    }
}
```

Figure 5 Iteration Pseudo Code.

Backward Analysis.

Backward analysis is very similar to forward analysis, while the only difference is that we will update the operands' value ranges, instead of the output value range of the instruction.

```
analyzeBlockBackward(currentBlockPtr, currentBlockExitVarSet, &successorEntryVarSet){
    for(all instruction inside currentBlock **REVERSELY**){
        if(sub || add || mul || rem){
            //update successorEntryVarSet
        }else{
            //ignore
        }
    }
}
```

Figure 6 Backward Analysis Pseudo Code.

Below are instruction analysis function declarations. Each of them supports both forward and backward analysis.

```cpp
/** analysis for alloca instruction, supports both forward and backward.*/
void analyzeAlloca(Instruction &I, std::map<Instruction *, varInterval> &blockMap,
                   std::map<std::string, Instruction *> &instructionMap, bool backward);
/** analysis for add instruction, supports both forward and backward.*/
void analyzeAdd(Instruction &I, std::map<Instruction *, varInterval> &blockMap,
                std::map<std::string, Instruction *> &instructionMap, bool backward);
/* analysis for sub instruction, supports both forward and backward.*/
void analyzeSub(Instruction &I, std::map<Instruction *, varInterval> &blockMap,
                std::map<std::string, Instruction *> &instructionMap, bool backward);
/** analysis for mul instruction, supports both forward and backward.*/
void analyzeMul(Instruction &I, std::map<Instruction *, varInterval> &blockMap,
                std::map<std::string, Instruction *> &instructionMap, bool backward);
/** analysis for srem instruction, supports both forward and backward.*/
void analyzeSrem(Instruction &I, std::map<Instruction *, varInterval> &blockMap,
                 std::map<std::string, Instruction *> &instructionMap, bool backward);
/** analysis for store instruction, supports both forward and backward.*/
void analyzeStore(Instruction &I, std::map<Instruction *, varInterval> &blockMap,
                  std::map<std::string, Instruction *> &instructionMap, bool backward);
/** analysis for load instruction, supports both forward and backward.*/
void analyzeLoad(Instruction &I, std::map<Instruction *, varInterval> &blockMap,
                 std::map<std::string, Instruction *> &instructionMap, bool backward);
/** analysis for br instruction, supports both forward and backward.*/
void analyzeBr(BasicBlock *BB, Instruction &I, std::map<Instruction *, varInterval> &blockMap,
               std::map<std::string, Instruction *> &instructionMap,
               std::map<BasicBlock *, std::map<Instruction *, varInterval>> &result);
```

Figure 7 Instruction Analysis Function Declaration.

- Build and Run:

Build:

```
jiadong-2:Assignment_new wangjiadong$ clang++  -o DiffAnalysisNew DiffAnalysisNew.cpp `llvm-c
onfig --cxxflags` `llvm-config --ldflags` `llvm-config --libs` -lpthread -lncurses -ldl
```

Figure 8 Difference Analyzer Build Command.

Run:

```
jiadong-2:Assignment_new wangjiadong$ clang -emit-llvm -S -o example4.ll example4.c
jiadong-2:Assignment_new wangjiadong$
```

Figure 9 LL Code Generation Command.

```
jiadong-2:Assignment_new wangjiadong$ ./DiffAnalysisNew example4.ll
```

Figure 10 Difference Analyzer Execution Command.

Results:

- Test Case 1: (Loop Free)

C Code:

```c
int main() {
    int a,b,c,d = 0;
    if (a > 0)
        c = 5;
    else
        c = 10;
    if (b > 0)
        d = -11;

}
```

Figure 11 Test Case 1 C Source Code.

LL Code:

```llvm
; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  store i32 0, i32* %1
  store i32 0, i32* %d, align 4
  %2 = load i32* %a, align 4
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %5

; <label>:4                                       ; preds = %0
  store i32 5, i32* %c, align 4
  br label %6

; <label>:5                                       ; preds = %0
  store i32 10, i32* %c, align 4
  br label %6

; <label>:6                                       ; preds = %5, %4
  %7 = load i32* %b, align 4
  %8 = icmp sgt i32 %7, 0
  br i1 %8, label %9, label %10

; <label>:9                                       ; preds = %6
  store i32 -11, i32* %d, align 4
  br label %10

; <label>:10                                      ; preds = %9, %6
  %11 = load i32* %1
  ret i32 %11
}
```

Figure 12 Test Case 1 LL Code.

Test Output:

```
jiadong-2:Assignment_new wangjiadong$ ./DiffAnalysisNew example1.ll
==============Analysis Report==============
>>>>Basic Block: %0
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  INF_NEG-INF_POS
  %b = alloca i32, align 4  >>  INF_NEG-INF_POS
  %c = alloca i32, align 4  >>  INF_NEG-INF_POS
  %d = alloca i32, align 4  >>  0-0
  %2 = load i32* %a, align 4  >>  INF_NEG-INF_POS
a <--> b : INF
a <--> c : INF
a <--> d : INF
b <--> c : INF
b <--> d : INF
c <--> d : INF
===========================================
>>>>Basic Block: %4
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  1-INF_POS
  %b = alloca i32, align 4  >>  INF_NEG-INF_POS
  %c = alloca i32, align 4  >>  5-5
  %d = alloca i32, align 4  >>  0-0
  %2 = load i32* %a, align 4  >>  1-INF_POS
a <--> b : INF
a <--> c : INF
a <--> d : INF
b <--> c : INF
b <--> d : INF
c <--> d : 5
```

Figure 13 Test Case 1 Output Basic Block 0,4.

```
===========================================
>>>>Basic Block: %5
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  INF_NEG-0
  %b = alloca i32, align 4  >>  INF_NEG-INF_POS
  %c = alloca i32, align 4  >>  10-10
  %d = alloca i32, align 4  >>  0-0
  %2 = load i32* %a, align 4  >>  INF_NEG-0
a <--> b : INF
a <--> c : INF
a <--> d : INF
b <--> c : INF
b <--> d : INF
c <--> d : 10
===========================================
>>>>Basic Block: %6
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  INF_NEG-INF_POS
  %b = alloca i32, align 4  >>  INF_NEG-INF_POS
  %c = alloca i32, align 4  >>  5-10
  %d = alloca i32, align 4  >>  0-0
  %2 = load i32* %a, align 4  >>  INF_NEG-INF_POS
  %7 = load i32* %b, align 4  >>  INF_NEG-INF_POS
a <--> b : INF
a <--> c : INF
a <--> d : INF
b <--> c : INF
b <--> d : INF
c <--> d : 10
```

Figure 14 Test Case 1 Output Basic Block 5,6.

```
============================================
>>>>Basic Block: %9
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  INF_NEG-INF_POS
  %b = alloca i32, align 4  >>  1-INF_POS
  %c = alloca i32, align 4  >>  5-10
  %d = alloca i32, align 4  >>  -11--11
  %2 = load i32* %a, align 4  >>  INF_NEG-INF_POS
  %7 = load i32* %b, align 4  >>  1-INF_POS
a <--> b : INF
a <--> c : INF
a <--> d : INF
b <--> c : INF
b <--> d : INF
c <--> d : 21
============================================
>>>>Basic Block: %10
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  INF_NEG-INF_POS
  %b = alloca i32, align 4  >>  INF_NEG-INF_POS
  %c = alloca i32, align 4  >>  5-10
  %d = alloca i32, align 4  >>  -11-0
  %2 = load i32* %a, align 4  >>  INF_NEG-INF_POS
  %7 = load i32* %b, align 4  >>  INF_NEG-INF_POS
  %11 = load i32* %1  >>  0-0
a <--> b : INF
a <--> c : INF
a <--> d : INF
b <--> c : INF
b <--> d : INF
c <--> d : 21
============================================
jiadong-2:Assignment_new wangjiadong$
```

Figure 15 Test Case 1 Output Basic Block 9, 10

- Test Case 2 (With While Loop)

C Source Code:

```c
int main() {
    int a,b,x,y,N, z = 0 ;
    // assume N is an input value
    int i = 0;
    while (i < N) {
        x = -((x + 2*y * 3*z) % 3);
        y = (3*x + 2*y + z) % 11;
        z++;
    }
}
```

Figure 16 Test Case 2 C Source Code.

LL Code:

```
; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %x = alloca i32, align 4
  %y = alloca i32, align 4
  %N = alloca i32, align 4
  %z = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 0, i32* %1
  store i32 0, i32* %z, align 4
  store i32 0, i32* %i, align 4
  br label %2

; <label>:2
  %3 = load i32* %i, align 4
  %4 = load i32* %N, align 4
  %5 = icmp slt i32 %3, %4
  br i1 %5, label %6, label %26
```

Figure 17 Test Case 2 LL Code Part 1.

```

; <label>:6
  %7 = load i32* %x, align 4
  %8 = load i32* %y, align 4
  %9 = mul nsw i32 2, %8
  %10 = mul nsw i32 %9, 3
  %11 = load i32* %z, align 4
  %12 = mul nsw i32 %10, %11
  %13 = add nsw i32 %7, %12
  %14 = srem i32 %13, 3
  %15 = sub nsw i32 0, %14
  store i32 %15, i32* %x, align 4
  %16 = load i32* %x, align 4
  %17 = mul nsw i32 3, %16
  %18 = load i32* %y, align 4
  %19 = mul nsw i32 2, %18
  %20 = add nsw i32 %17, %19
  %21 = load i32* %z, align 4
  %22 = add nsw i32 %20, %21
  %23 = srem i32 %22, 11
  store i32 %23, i32* %y, align 4
  %24 = load i32* %z, align 4
  %25 = add nsw i32 %24, 1
  store i32 %25, i32* %z, align 4
  br label %2

; <label>:26
  %27 = load i32* %1
  ret i32 %27
}
```

Figure 18 Test Case 2 LL Code Part 2.

Analysis Output:

```
==============Analysis Report==============
>>>>Basic Block: %0
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  INF_NEG-INF_POS
  %b = alloca i32, align 4  >>  INF_NEG-INF_POS
  %x = alloca i32, align 4  >>  INF_NEG-INF_POS
  %y = alloca i32, align 4  >>  INF_NEG-INF_POS
  %N = alloca i32, align 4  >>  INF_NEG-INF_POS
  %z = alloca i32, align 4  >>  0-0
  %i = alloca i32, align 4  >>  0-0
a <--> b : INF
a <--> x : INF
a <--> y : INF
a <--> N : INF
a <--> z : INF
a <--> i : INF
b <--> x : INF
b <--> y : INF
b <--> N : INF
b <--> z : INF
b <--> i : INF
x <--> y : INF
x <--> N : INF
x <--> z : INF
x <--> i : INF
y <--> N : INF
y <--> z : INF
y <--> i : INF
N <--> z : INF
N <--> i : INF
z <--> i : 0
```

Figure 19 Test Case 2 Basic Block 0 Analysis Output.

```
=============================================
>>>>Basic Block: %2
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  INF_NEG-INF_POS
  %b = alloca i32, align 4  >>  INF_NEG-INF_POS
  %x = alloca i32, align 4  >>  INF_NEG-INF_POS
  %y = alloca i32, align 4  >>  INF_NEG-INF_POS
  %N = alloca i32, align 4  >>  INF_NEG-INF_POS
  %z = alloca i32, align 4  >>  0-INF_POS
  %i = alloca i32, align 4  >>  0-0
  %3 = load i32* %i, align 4  >>  0-0
  %4 = load i32* %N, align 4  >>  INF_NEG-INF_POS
  %7 = load i32* %x, align 4  >>  INF_NEG-INF_POS
  %8 = load i32* %y, align 4  >>  INF_NEG-INF_POS
  %17 = mul nsw i32 3, %16  >>  -6-0
  %9 = mul nsw i32 2, %8  >>  INF_NEG-INF_POS
  %10 = mul nsw i32 %9, 3  >>  INF_NEG-INF_POS
  %11 = load i32* %z, align 4  >>  0-INF_POS
  %12 = mul nsw i32 %10, %11  >>  0-INF_POS
  %13 = add nsw i32 %7, %12  >>  INF_NEG-INF_POS
  %14 = srem i32 %13, 3  >>  0-2
  %15 = sub nsw i32 0, %14  >>  -2-0
  %16 = load i32* %x, align 4  >>  -2-0
  %18 = load i32* %y, align 4  >>  INF_NEG-INF_POS
  %19 = mul nsw i32 2, %18  >>  INF_NEG-INF_POS
  %20 = add nsw i32 %17, %19  >>  INF_NEG-INF_POS
  %21 = load i32* %z, align 4  >>  0-INF_POS
  %22 = add nsw i32 %20, %21  >>  INF_NEG-INF_POS
  %23 = srem i32 %22, 11  >>  0-10
  %24 = load i32* %z, align 4  >>  0-INF_POS
  %25 = add nsw i32 %24, 1  >>  1-INF_POS
```

Figure 20 Test Case 2 Basic Block 2 Variable Ranges.

```
a <--> b : INF
a <--> x : INF
a <--> y : INF
a <--> N : INF
a <--> z : INF
a <--> i : INF
b <--> x : INF
b <--> y : INF
b <--> N : INF
b <--> z : INF
b <--> i : INF
x <--> y : INF
x <--> N : INF
x <--> z : INF
x <--> i : INF
y <--> N : INF
y <--> z : INF
y <--> i : INF
N <--> z : INF
N <--> i : INF
z <--> i : INF
```

Figure 21 Test Case 2 Basic Block 2 Variable Pair Maximum Difference.

```
============================================
>>>>Basic Block: %6
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  INF_NEG-INF_POS
  %b = alloca i32, align 4  >>  INF_NEG-INF_POS
  %x = alloca i32, align 4  >>  -2-0
  %y = alloca i32, align 4  >>  0-10
  %N = alloca i32, align 4  >>  1-INF_POS
  %z = alloca i32, align 4  >>  1-INF_POS
  %i = alloca i32, align 4  >>  0-0
  %3 = load i32* %i, align 4  >>  0-0
  %4 = load i32* %N, align 4  >>  1-INF_POS
  %7 = load i32* %x, align 4  >>  INF_NEG-INF_POS
  %8 = load i32* %y, align 4  >>  INF_NEG-INF_POS
  %17 = mul nsw i32 3, %16  >>  -6-0
  %9 = mul nsw i32 2, %8  >>  INF_NEG-INF_POS
  %10 = mul nsw i32 %9, 3  >>  INF_NEG-INF_POS
  %11 = load i32* %z, align 4  >>  0-INF_POS
  %12 = mul nsw i32 %10, %11  >>  0-INF_POS
  %13 = add nsw i32 %7, %12  >>  INF_NEG-INF_POS
  %14 = srem i32 %13, 3  >>  0-2
  %15 = sub nsw i32 0, %14  >>  -2-0
  %16 = load i32* %x, align 4  >>  -2-0
  %18 = load i32* %y, align 4  >>  INF_NEG-INF_POS
  %19 = mul nsw i32 2, %18  >>  INF_NEG-INF_POS
  %20 = add nsw i32 %17, %19  >>  INF_NEG-INF_POS
  %21 = load i32* %z, align 4  >>  0-INF_POS
  %22 = add nsw i32 %20, %21  >>  INF_NEG-INF_POS
  %23 = srem i32 %22, 11  >>  0-10
  %24 = load i32* %z, align 4  >>  0-INF_POS
  %25 = add nsw i32 %24, 1  >>  1-INF_POS
```

Figure 22 Test Case 2 Basic Block 6 Variable Value Range.

```
a <--> b : INF
a <--> x : INF
a <--> y : INF
a <--> N : INF
a <--> z : INF
a <--> i : INF
b <--> x : INF
b <--> y : INF
b <--> N : INF
b <--> z : INF
b <--> i : INF
x <--> y : 12
x <--> N : INF
x <--> z : INF
x <--> i : 2
y <--> N : INF
y <--> z : INF
y <--> i : 10
N <--> z : INF
N <--> i : INF
z <--> i : INF
```

Figure 23 Test Case 2 Basic Block 6 Variable Pair Maximum Difference.

```
===========================================
>>>>Basic Block: %26
  %1 = alloca i32, align 4  >>  0-0
  %a = alloca i32, align 4  >>  INF_NEG-INF_POS
  %b = alloca i32, align 4  >>  INF_NEG-INF_POS
  %x = alloca i32, align 4  >>  INF_NEG-INF_POS
  %y = alloca i32, align 4  >>  INF_NEG-INF_POS
  %N = alloca i32, align 4  >>  INF_NEG-0
  %z = alloca i32, align 4  >>  0-INF_POS
  %i = alloca i32, align 4  >>  0-0
  %3 = load i32* %i, align 4  >>  0-0
  %4 = load i32* %N, align 4  >>  INF_NEG-0
  %7 = load i32* %x, align 4  >>  INF_NEG-INF_POS
  %8 = load i32* %y, align 4  >>  INF_NEG-INF_POS
  %17 = mul nsw i32 3, %16  >>  -6-0
  %9 = mul nsw i32 2, %8  >>  INF_NEG-INF_POS
  %10 = mul nsw i32 %9, 3  >>  INF_NEG-INF_POS
  %11 = load i32* %z, align 4  >>  0-INF_POS
  %12 = mul nsw i32 %10, %11  >>  0-INF_POS
  %13 = add nsw i32 %7, %12  >>  INF_NEG-INF_POS
  %14 = srem i32 %13, 3  >>  0-2
  %15 = sub nsw i32 0, %14  >>  -2-0
  %16 = load i32* %x, align 4  >>  -2-0
  %18 = load i32* %y, align 4  >>  INF_NEG-INF_POS
  %19 = mul nsw i32 2, %18  >>  INF_NEG-INF_POS
  %20 = add nsw i32 %17, %19  >>  INF_NEG-INF_POS
  %21 = load i32* %z, align 4  >>  0-INF_POS
  %22 = add nsw i32 %20, %21  >>  INF_NEG-INF_POS
  %23 = srem i32 %22, 11  >>  0-10
  %24 = load i32* %z, align 4  >>  0-INF_POS
  %25 = add nsw i32 %24, 1  >>  1-INF_POS
  %27 = load i32* %1  >>  0-0
```

Figure 24 Test Case 2 Basic Block 26 Variable Value Range.

```
a <--> b : INF
a <--> x : INF
a <--> y : INF
a <--> N : INF
a <--> z : INF
a <--> i : INF
b <--> x : INF
b <--> y : INF
b <--> N : INF
b <--> z : INF
b <--> i : INF
x <--> y : INF
x <--> N : INF
x <--> z : INF
x <--> i : INF
y <--> N : INF
y <--> z : INF
y <--> i : INF
N <--> z : INF
N <--> i : INF
z <--> i : INF
===========================================
jiadong-2:Assignment_new wangjiadong$
```

Figure 25 Test Case 2 Basic Block 26 Variable Pair Maximum Difference.