CS5218 Principal of Program Analysis

Assignment 3

Wang Jiadong (A0105703)

Interval analysis is a famous analysis developed in 1950s and 1960s. In this approach, an interval for a variable x is represented by x: [a,b] where a is the lower bound and b is the upper bound on the possible values that x can contain. In this assignment, we will be implementing an Interval analysis with some path sensitivity for all the variables in the input program.

Task1: Interval Analysis

- Flow (F).

Intuitively, interval 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, then it is very clearly the starting block is init(S*).

- Initial Value(ι).

Since there are no variables declared initially, we just define the initial value for each context (path) as an empty set (Ø). 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.

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”.

- Lattice and Partial Order

The lattice can be represented in this way:

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

where P is the complete context set.

It is partially ordered by subset inclusion: $\sqsubseteq=\subseteq$

- Monotonicity

As mentioned above, under the same context (constraints), 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:

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

where:

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

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

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.

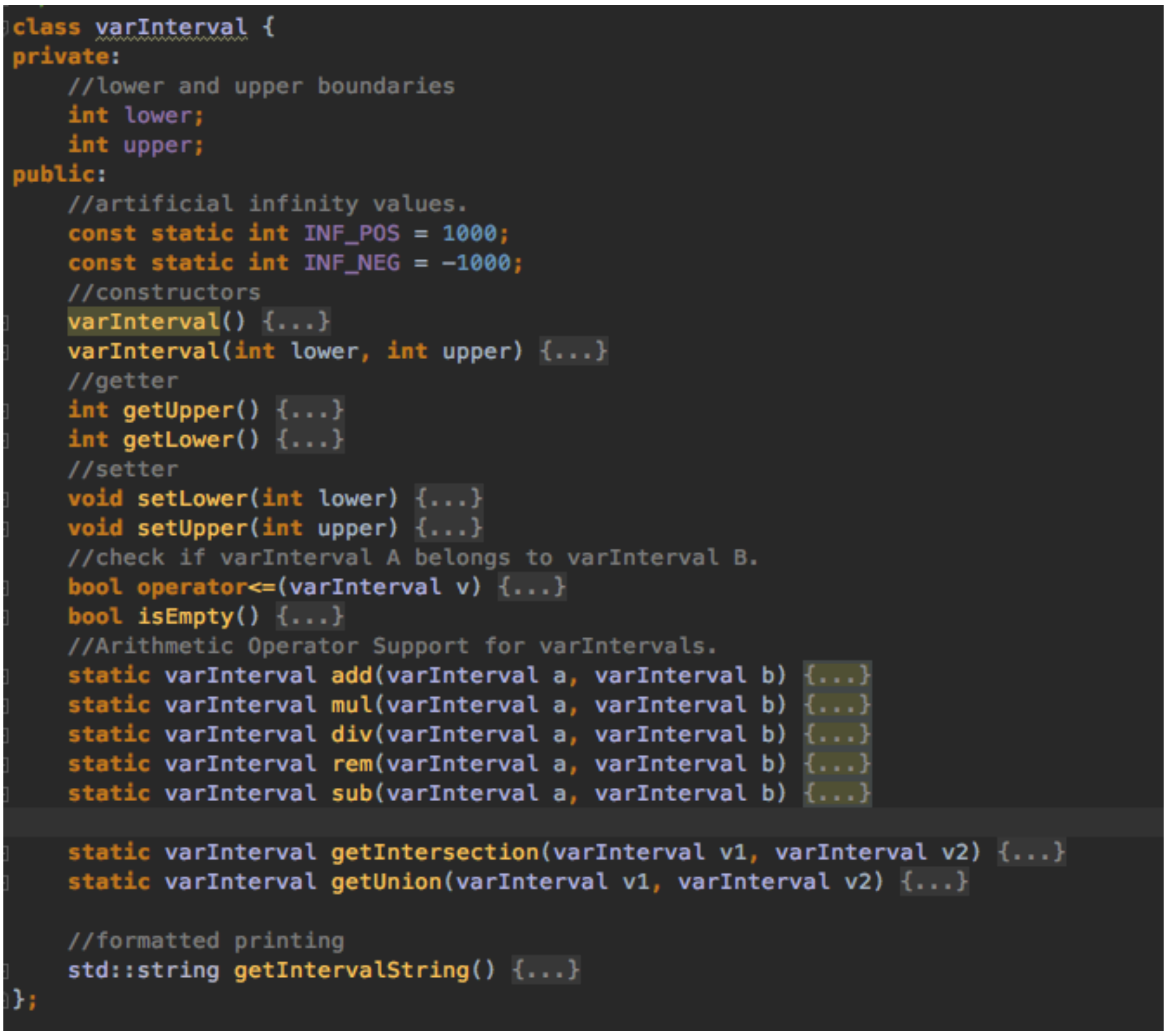
Task 2 & 3: Design and Implementation of path-sensitive interval analysis.

- 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. Besides, utility functions like subset checker, emptiness checker, intersection function, union function, and formatted string generation are also available.



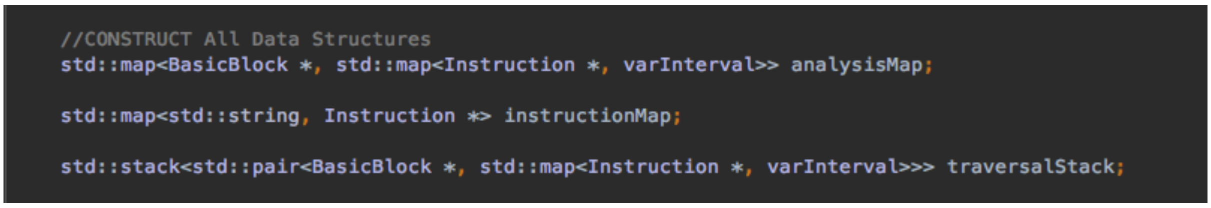
```cpp
class varInterval {
private:
    //lower and upper boundaries
    int lower;
    int upper;
public:
    //artificial infinity values.
    const static int INF_POS = 1000;
    const static int INF_NEG = -1000;
    //constructors
    varInterval() {...}
    varInterval(int lower, int upper) {...}
    //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) {...}

    static varInterval getIntersection(varInterval v1, varInterval v2) {...}
    static varInterval getUnion(varInterval v1, varInterval v2) {...}

    //formatted printing
    std::string getIntervalString() {...}
};
```

- Variables.

Pre-Context Generation:

In order to get the context list for the program, my approach is to run the non-path sensitive interval analysis first, which is very similar to assignment 2.

Hence there are a few global variables used for pre-context generation:



```cpp
    //CONSTRUCT All Data Structures
    std::map<BasicBlock *, std::map<Instruction *, varInterval>> analysisMap;

    std::map<std::string, Instruction *> instructionMap;

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

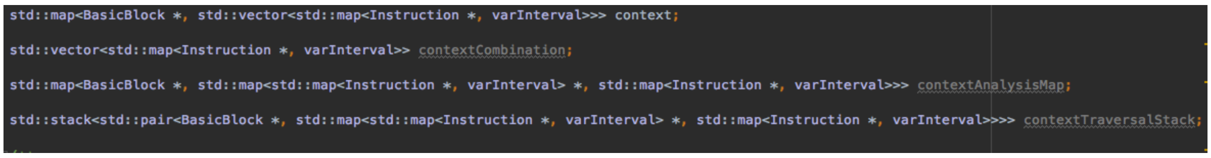
AnalysisMap is the output collector. It contains the range for each variable in each of the basic blocks, and it is updated for each iteration. Once the iteration is completed.

InstructionMap is a global dictionary for name to instruction pointer mapping.

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.

Post-Context Generation:

After we get the context list for the program, we will start the path-sensitive analysis, during which a few global variables are declared.



```
std::map<BasicBlock *, std::vector<std::map<Instruction *, varInterval>>> context;

std::vector<std::map<Instruction *, varInterval>> contextCombination;

std::map<BasicBlock *, std::map<std::map<Instruction *, varInterval> *, std::map<Instruction *, varInterval>>> contextAnalysisMap;

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

Virtually, they are just counterparts of the pre-context generation global variables in path-sensitive context.

Context is the context generated by single conditional branch instruction. For example,

$$If\ (\ a\ >\ 0\ )\ \{\ \ldots\ \}\ then\ \{\ \ldots\ \}$$

This will generate single context pair:

$$a\ \rightarrow [1, +\infty), and\ a\ \rightarrow (-\infty, 0]$$

Context organize these context pairs in the map form:

$$\{\text{Basic Block \#0}\ \rightarrow [\{\text{a} \rightarrow [1, +\infty)\}, \{\text{a} \rightarrow (-\infty, 0]\ \}]\}$$

ContextCombination is the complete set of all combinations of different context pairs, forming the all possible contexts for the program. Its data structure is comparatively simpler:

$$[\{a \rightarrow [1, +\infty), b \rightarrow [1, +\infty)\},$$
$$\{a \rightarrow [1, +\infty), b \rightarrow (-\infty, 0]\},$$
$$\{a \rightarrow (-\infty, 0], b \rightarrow [1, +\infty)\},$$
$$\{a \rightarrow (-\infty, 0], b \rightarrow (-\infty, 0]\}]$$

ContextAnalysisMap is the counterpart of **AnalysisMap** in path-sensitive context. Similarly, **ContextTraversalStack** is the counterpart of **TraversalStack** in path-sensitive context.

- Algorithm Pseudo-Code

Initialization

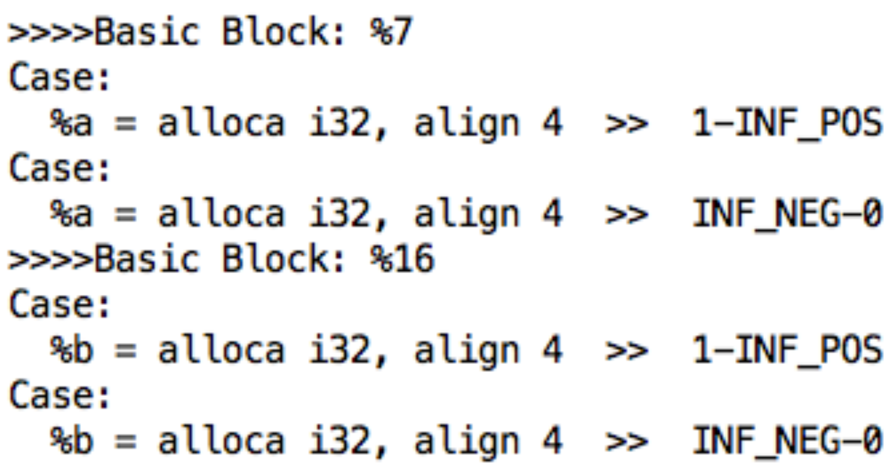
In order to initialize, not only we need to declare the variables, but also set the initial state for the traversal stack, which is the entry block with empty entry variable set.

Pre-Context Generation

Context are generated due to conditional branch instructions. When there is a IF…ELSE… structure, it will create two different paths. In LLVM, when we consider about conditional branching, we will conduct backward analysis to generate two sets of input data for the two successors. These two sets of input data can be considered as a context pair, while we also need to do some processing:

- Remove the variables that have identical interval in both contexts.
- Remove the local register variables from the context pair.
- Remove the context pairs that have no variables inside.

After complete these processing steps, we will have a few context pairs. An example is shown below:



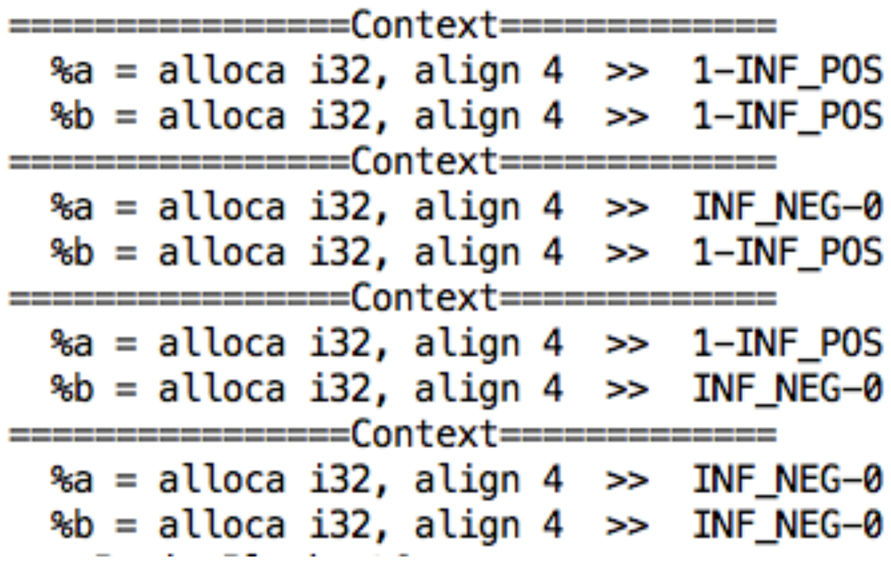
```
>>>>Basic Block: %7
Case:
  %a = alloca i32, align 4  >>  1-INF_POS
Case:
  %a = alloca i32, align 4  >>  INF_NEG-0
>>>>Basic Block: %16
Case:
  %b = alloca i32, align 4  >>  1-INF_POS
Case:
  %b = alloca i32, align 4  >>  INF_NEG-0
```

This output means that:

In Basic Block 7, one context pair is created for variable **a,** which is $(-\infty, 0]$ *and* $[1, +\infty)$;

In Basic Block 16, one context pair is created for variable **b**, which is also $(-\infty, 0]$ *and* $[1, +\infty)$.

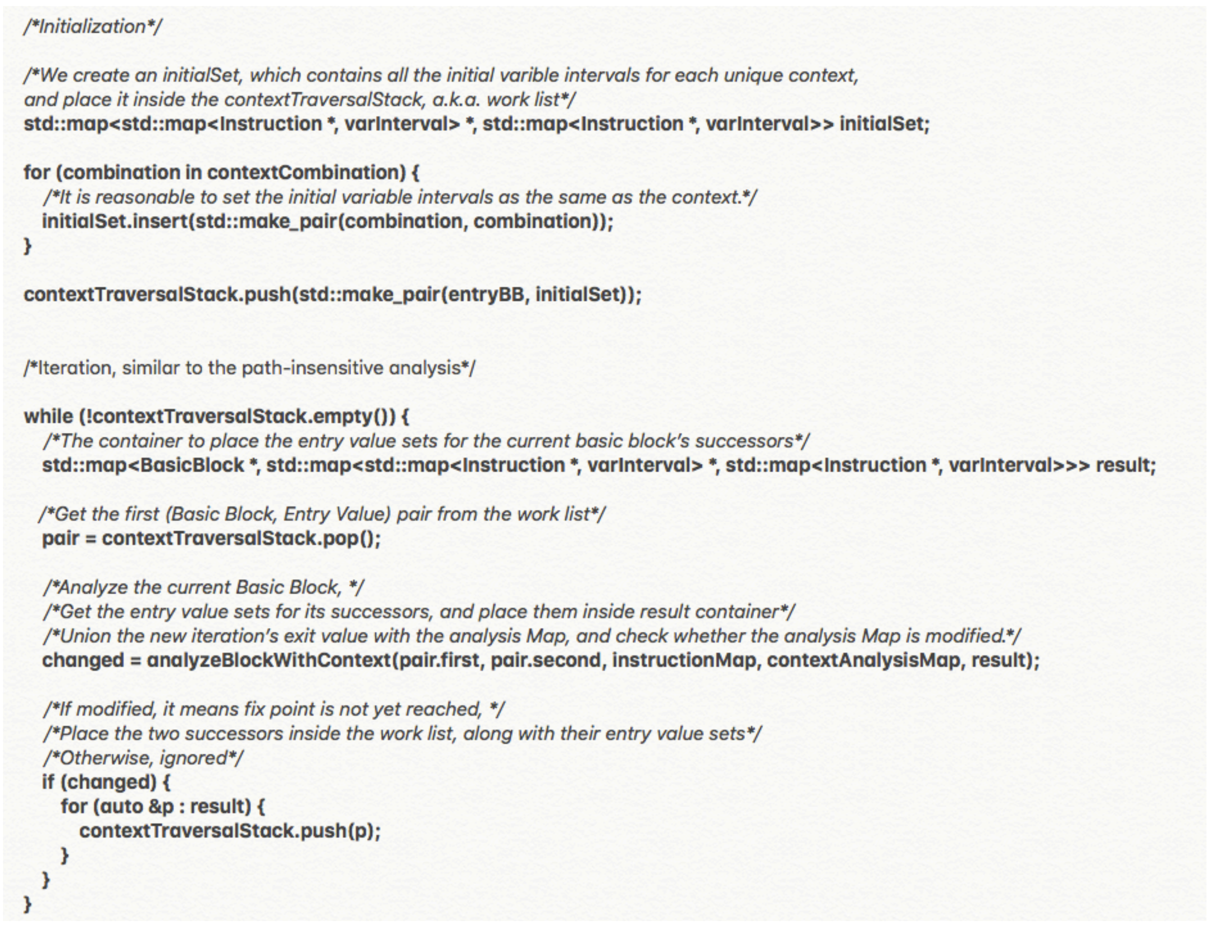
Once context pairs are generated, we need to generate the combination of different context pairs for the program. An example is also provided below:



```
================Context============
  %a = alloca i32, align 4  >>  1-INF_POS
  %b = alloca i32, align 4  >>  1-INF_POS
================Context============
  %a = alloca i32, align 4  >>  INF_NEG-0
  %b = alloca i32, align 4  >>  1-INF_POS
================Context============
  %a = alloca i32, align 4  >>  1-INF_POS
  %b = alloca i32, align 4  >>  INF_NEG-0
================Context============
  %a = alloca i32, align 4  >>  INF_NEG-0
  %b = alloca i32, align 4  >>  INF_NEG-0
```

Hence, as we can see, four different context sets are generated for this program.

Post-Context Generation



```
/*Initialization*/

/*We create an initialSet, which contains all the initial varible intervals for each unique context,
and place it inside the contextTraversalStack, a.k.a. work list*/
std::map<std::map<Instruction *, varInterval> *, std::map<Instruction *, varInterval>> initialSet;

for (combination in contextCombination) {
  /*It is reasonable to set the initial variable intervals as the same as the context.*/
  initialSet.insert(std::make_pair(combination, combination));
}

contextTraversalStack.push(std::make_pair(entryBB, initialSet));


/*Iteration, similar to the path-insensitive analysis*/

while (!contextTraversalStack.empty()) {
  /*The container to place the entry value sets for the current basic block's successors*/
  std::map<BasicBlock *, std::map<std::map<Instruction *, varInterval> *, std::map<Instruction *, varInterval>>> result;

  /*Get the first (Basic Block, Entry Value) pair from the work list*/
  pair = contextTraversalStack.pop();

  /*Analyze the current Basic Block, */
  /*Get the entry value sets for its successors, and place them inside result container*/
  /*Union the new iteration's exit value with the analysis Map, and check whether the analysis Map is modified.*/
  changed = analyzeBlockWithContext(pair.first, pair.second, instructionMap, contextAnalysisMap, result);

  /*If modified, it means fix point is not yet reached, */
  /*Place the two successors inside the work list, along with their entry value sets*/
  /*Otherwise, ignored*/
  if (changed) {
    for (auto &p : result) {
      contextTraversalStack.push(p);
    }
  }
}
```

The post-context generation analysis is pretty similar to the path-sensitive analysis:

- Initialize the contextTraversalStack, with corresponding initial values.
- Iterate until the contextTraversalStack is drained.
- During each iteration, get the first Basic-Block-Entry-Value pair from contextTraversalStack, analyzing it, and collecting the new exit value for current basic block, as well as the entry value for the successors.
- If the current iteration has modified contextAnalysisMap, then the Basic-Block-Entry-Value pairs for successors are placed into contextTraversalStack; otherwise, the successors are ignored.

Once termination is reached, the exit value set for each basic block is recorded inside the contextAnalysisMap.

Path-Sensitive Block Analysis

Similar to the path-insensitive block analysis, we analyze each of the instructions from beginning to end for each basic block. However, the algorithm for each instruction is totally different from path-insensitive analysis.

For arithmetic operation instructions, the algorithm can be generalized as below procedure:

1. Check if this operation is exerted on context variable.
2. If (1) is false, we simply calculate the new interval for the output variable according to the operands, and replace its old value.
3. If (1) is true,

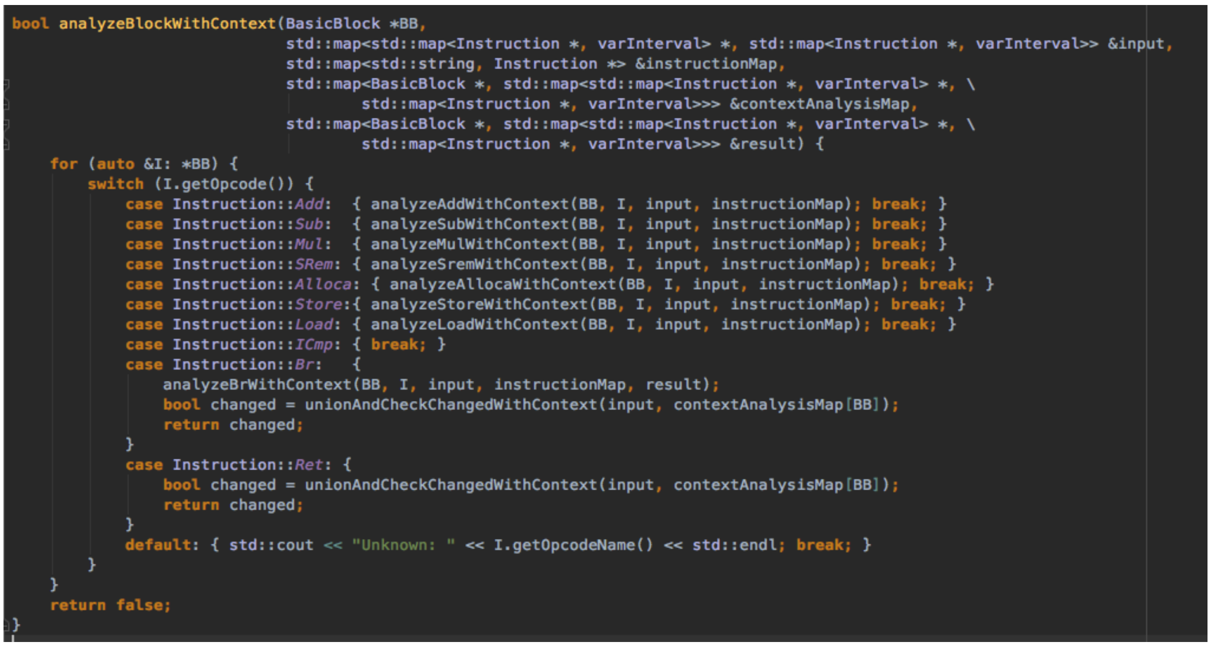
3.1. Join all the possible intervals under all contexts for all variables, as a joined set (JS).

3.2. Calculate the new interval for the context variable with the JS, and update it.

3.3. For each context, check if it has intersection with the JS.

3.4. If (3.3) is true, then overwrite the variable intervals inside this context with JS.

3.5. If (3.3) is false, then overwrite the variable intervals inside this context with Ø.



```
bool analyzeBlockWithContext(BasicBlock *BB,
                             std::map<std::map<Instruction *, varInterval> *, std::map<Instruction *, varInterval>> &input,
                             std::map<std::string, Instruction *> &instructionMap,
                             std::map<BasicBlock *, std::map<std::map<Instruction *, varInterval> *, \
                                     std::map<Instruction *, varInterval>>> &contextAnalysisMap,
                             std::map<BasicBlock *, std::map<std::map<Instruction *, varInterval> *, \
                                     std::map<Instruction *, varInterval>>> &result) {
    for (auto &I: *BB) {
        switch (I.getOpcode()) {
            case Instruction::Add:  { analyzeAddWithContext(BB, I, input, instructionMap); break; }
            case Instruction::Sub:  { analyzeSubWithContext(BB, I, input, instructionMap); break; }
            case Instruction::Mul:  { analyzeMulWithContext(BB, I, input, instructionMap); break; }
            case Instruction::SRem: { analyzeSremWithContext(BB, I, input, instructionMap); break; }
            case Instruction::Alloca: { analyzeAllocaWithContext(BB, I, input, instructionMap); break; }
            case Instruction::Store:{ analyzeStoreWithContext(BB, I, input, instructionMap); break; }
            case Instruction::Load: { analyzeLoadWithContext(BB, I, input, instructionMap); break; }
            case Instruction::ICmp: { break; }
            case Instruction::Br:   {
                analyzeBrWithContext(BB, I, input, instructionMap, result);
                bool changed = unionAndCheckChangedWithContext(input, contextAnalysisMap[BB]);
                return changed;
            }
            case Instruction::Ret: {
                bool changed = unionAndCheckChangedWithContext(input, contextAnalysisMap[BB]);
                return changed;
            }
            default: { std::cout << "Unknown: " << I.getOpcodeName() << std::endl; break; }
        }
    }
    return false;
}
```

This technique applies to all but not limit to:

$$ADD, SUB, MUL, SREM, ALLOCA, STORE, LOAD$$

BR Instruction

Branch instruction is actually easier for path-sensitive analysis, because we do not need to conduct backward analysis. The algorithm is rather straightforward:

1. Check if the current Branching instruction is conditional or unconditional.
2. If (1) is unconditional, directly set the entry value set for its sole successor using its exit value set.
3. If (1) is conditional,

   3.1. Initially set the entry value set for both successors using the current basic block's exit value set.

   3.2. For each side of the branch, calculate the intersection with the corresponding side of the context pair, and overwrite the initial entry value set.

Here is an example to elaborate 3.2. condition above. Suppose, we have a BR instruction as follow:

```
%18 = icmp sgt i32 %17, 0
br i1 %18, label %19, label %20
```

If the exit value set for current basic block is:

$$\%17 \ \rightarrow [-5, 5]$$

Hence the initial entry value for both $\%19$ and $\%20$ is:

$$\%17 \ \rightarrow [-5, 5]$$

However, since we have a branching here:

$$IF\ \%17 > 0, THEN\ \%19, ELSE\ \%20$$

Therefore, after intersect with the branching instruction, we have the respective value intervals for $\%19$ and $\%20$ as:

$$\%19\colon\ \%17 \ \rightarrow [1, 5]$$

$$\%20\colon\ \%17 \ \rightarrow [-5, 0]$$

After we get the entry value set for both successors. We need to check whether contextAnalysisMap is modified for current basic block during this iteration.

RET Instruction

This is comparatively easier than BR Instruction. We simply have to check whether contextAnalysisMap is modified and return a Boolean value.

- Build and Run.

Emit LLVM File

```
${llvm_bin_path}clang -emit-llvm -S -o  example2.ll example2.c
```

Build

```
${llvm_bin_path}clang++ -o IntervalAnalysis IntervalAnalysis.cpp \
`${llvm_bin_path}llvm-config --cxxflags` \
`${llvm_bin_path}llvm-config --ldflags` \
`${llvm_bin_path}llvm-config --libs` \
-lpthread -lncurses -ldl
```

Run

```
./IntervalAnalysis  example2.ll
```

- Output

Test Case 1: example 1 (Loop-Free)

Source Code (C):

```
int main() {
    int x, a;
    a = 10;
    int b = 5;
    if (a > 0)
        x = 3 + b;
    else
        x = 3 - b;
    assert (x >= 0);
    return x;
}
```

LLVM Code (LL):

```
; ModuleID = 'example1.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %x = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %1
  store i32 10, i32* %a, align 4
  store i32 5, i32* %b, align 4
  %2 = load i32* %a, align 4
  %3 = icmp sgt i32 %2, 0
  br i1 %3, label %4, label %7

; <label>:4                                       ; preds = %0
  %5 = load i32* %b, align 4
  %6 = add nsw i32 3, %5
  store i32 %6, i32* %x, align 4
  br label %10

; <label>:7                                       ; preds = %0
  %8 = load i32* %b, align 4
  %9 = sub nsw i32 3, %8
  store i32 %9, i32* %x, align 4
  br label %10

; <label>:10                                      ; preds = %7, %4
  %11 = load i32* %x, align 4
  %12 = icmp sge i32 %11, 0
  %13 = zext i1 %12 to i32
  %14 = call i32 (i32, ...)* bitcast (i32 (...)* @assert to i32 (i32,
...)*)(i32 %13)
  %15 = load i32* %x, align 4
  ret i32 %15
}

declare i32 @assert(...) #1
```

Test Output

```
>>>>Basic Block: %0==============================
  %x = alloca i32, align 4  -> INF_NEG-INF_POS
  %a = alloca i32, align 4  -> 10-10
  %b = alloca i32, align 4  -> 5-5
>>>>Basic Block: %4==============================
  %x = alloca i32, align 4  -> 8-8
  %a = alloca i32, align 4  -> 10-10
  %b = alloca i32, align 4  -> 5-5
>>>>Basic Block: %7==============================
>>>>Basic Block: %10==============================
  %x = alloca i32, align 4  -> 8-8
  %a = alloca i32, align 4  -> 10-10
  %b = alloca i32, align 4  -> 5-5
```

Test Case 2: example 2 (With-Loop)

Source Code (C)

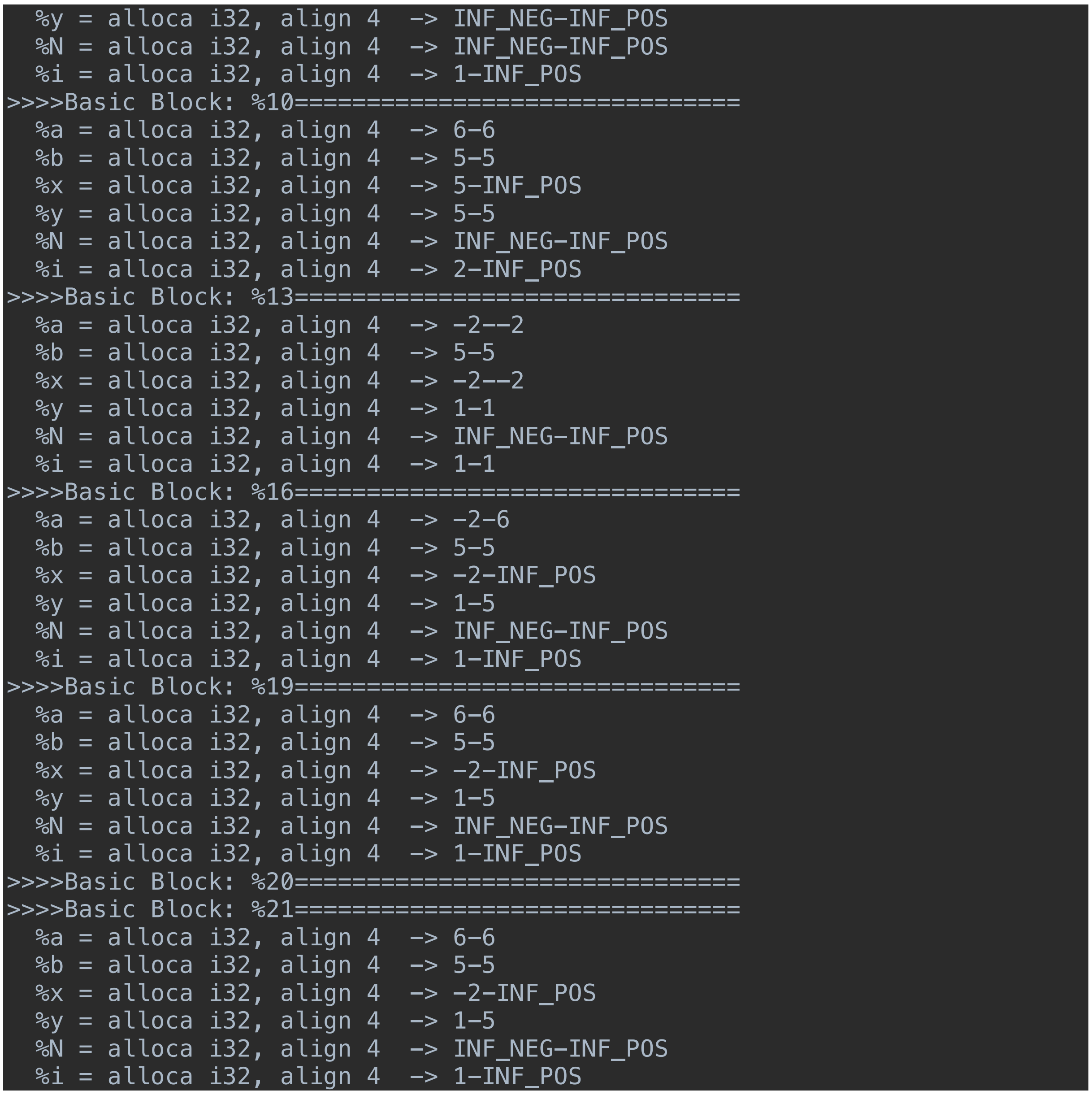
```
int main() {
    int a = -2, b = 5, x = 0, y, N;
    // assume N is an input value
    int i = 0;
    while (i++ < N) {
        if (a > 0) {
            x = x + 7;
            y = 5;
        } else {
            x = x - 2;
            y = 1;
        }
        if (b > 0) {
            a = 6;
        } else {
            a = -5;
        }
    }
}
```

LLVM Code (LL)

```
; ModuleID = 'example2.c'
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

; 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
  %i = alloca i32, align 4
  store i32 0, i32* %1
  store i32 -2, i32* %a, align 4
  store i32 5, i32* %b, align 4
  store i32 0, i32* %x, align 4
  store i32 0, i32* %i, align 4
  br label %2

; <label>:2                                         ; preds = %21, %0
  %3 = load i32* %i, align 4
  %4 = add nsw i32 %3, 1
  store i32 %4, i32* %i, align 4
  %5 = load i32* %N, align 4
  %6 = icmp slt i32 %3, %5
  br i1 %6, label %7, label %22

; <label>:7                                         ; preds = %2
  %8 = load i32* %a, align 4
  %9 = icmp sgt i32 %8, 0
  br i1 %9, label %10, label %13

; <label>:10                                        ; preds = %7
  %11 = load i32* %x, align 4
  %12 = add nsw i32 %11, 7
  store i32 %12, i32* %x, align 4
  store i32 5, i32* %y, align 4
  br label %16

; <label>:13                                        ; preds = %7
  %14 = load i32* %x, align 4
  %15 = sub nsw i32 %14, 2
  store i32 %15, i32* %x, align 4
  store i32 1, i32* %y, align 4
  br label %16

; <label>:16                                      ; preds = %13, %10
  %17 = load i32* %b, align 4
  %18 = icmp sgt i32 %17, 0
  br i1 %18, label %19, label %20

; <label>:19                                      ; preds = %16
  store i32 6, i32* %a, align 4
  br label %21

; <label>:20                                      ; preds = %16
  store i32 -5, i32* %a, align 4
  br label %21

; <label>:21                                      ; preds = %20, %19
  br label %2

; <label>:22                                      ; preds = %2
  %23 = load i32* %1
  ret i32 %23
}

attributes #0 = { nounwind ssp uwtable "less-precise-fpmad"="false" "no-
frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-
math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8"
"unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.5.2 (tags/RELEASE_352/final)"}
```

Output

```
>>>>Basic Block: %0================================
  %a = alloca i32, align 4  -> -2--2
  %b = alloca i32, align 4  -> 5-5
  %x = alloca i32, align 4  -> 0-0
  %y = alloca i32, align 4  -> INF_NEG-INF_POS
  %N = alloca i32, align 4  -> INF_NEG-INF_POS
  %i = alloca i32, align 4  -> 0-0
>>>>Basic Block: %2================================
  %a = alloca i32, align 4  -> -2-6
  %b = alloca i32, align 4  -> 5-5
  %x = alloca i32, align 4  -> -2-INF_POS
  %y = alloca i32, align 4  -> INF_NEG-INF_POS
  %N = alloca i32, align 4  -> INF_NEG-INF_POS
  %i = alloca i32, align 4  -> 1-INF_POS
>>>>Basic Block: %7================================
  %a = alloca i32, align 4  -> -2-6
  %b = alloca i32, align 4  -> 5-5
  %x = alloca i32, align 4  -> -2-INF_POS
  %y = alloca i32, align 4  -> INF_NEG-INF_POS
  %N = alloca i32, align 4  -> INF_NEG-INF_POS
  %i = alloca i32, align 4  -> 1-INF_POS
>>>>Basic Block: %22================================
  %a = alloca i32, align 4  -> -2-6
  %b = alloca i32, align 4  -> 5-5
  %x = alloca i32, align 4  -> -2-INF_POS
```



```
  %y = alloca i32, align 4  -> INF_NEG-INF_POS
  %N = alloca i32, align 4  -> INF_NEG-INF_POS
  %i = alloca i32, align 4  -> 1-INF_POS
>>>>Basic Block: %10================================
  %a = alloca i32, align 4  -> 6-6
  %b = alloca i32, align 4  -> 5-5
  %x = alloca i32, align 4  -> 5-INF_POS
  %y = alloca i32, align 4  -> 5-5
  %N = alloca i32, align 4  -> INF_NEG-INF_POS
  %i = alloca i32, align 4  -> 2-INF_POS
>>>>Basic Block: %13================================
  %a = alloca i32, align 4  -> -2--2
  %b = alloca i32, align 4  -> 5-5
  %x = alloca i32, align 4  -> -2--2
  %y = alloca i32, align 4  -> 1-1
  %N = alloca i32, align 4  -> INF_NEG-INF_POS
  %i = alloca i32, align 4  -> 1-1
>>>>Basic Block: %16================================
  %a = alloca i32, align 4  -> -2-6
  %b = alloca i32, align 4  -> 5-5
  %x = alloca i32, align 4  -> -2-INF_POS
  %y = alloca i32, align 4  -> 1-5
  %N = alloca i32, align 4  -> INF_NEG-INF_POS
  %i = alloca i32, align 4  -> 1-INF_POS
>>>>Basic Block: %19================================
  %a = alloca i32, align 4  -> 6-6
  %b = alloca i32, align 4  -> 5-5
  %x = alloca i32, align 4  -> -2-INF_POS
  %y = alloca i32, align 4  -> 1-5
  %N = alloca i32, align 4  -> INF_NEG-INF_POS
  %i = alloca i32, align 4  -> 1-INF_POS
>>>>Basic Block: %20================================
>>>>Basic Block: %21================================
  %a = alloca i32, align 4  -> 6-6
  %b = alloca i32, align 4  -> 5-5
  %x = alloca i32, align 4  -> -2-INF_POS
  %y = alloca i32, align 4  -> 1-5
  %N = alloca i32, align 4  -> INF_NEG-INF_POS
  %i = alloca i32, align 4  -> 1-INF_POS
```