Lab 6: Dataflow Analysis

Synopsis

Building a "division-by-zero" static analysis for a subset of the C language that includes branches and loops.

Objective

In this lab, you will build a static analyzer that detects potential divide-by-zero erros in C programs at compile-time. You will accomplish this by writing an LLVM pass. Since developing a static analyzer for a full-fledged language like C is a non-trivial endeavor, this lab will be split up into two parts.

PART 1

1. Implement `DivZeroAnalysis::check` that checks if a given instruction could lead to an error.
2. Implement `DivZeroAnalysis::transfer` found in `src/Transfer.cpp`.
3. Implement the `eval` functions in `src/Transfer.cpp` by completing the provided function stubs.

PART 2

For the second part of this lab you will implement various functions in `src/ChaoticIteration`.

1. Implement `doAnalysis` function that performs the chaotic iteration algorithm for your analysis.
2. Implement `flowIn` function that joins the out memory of all incoming flows.
3. Implement `flowOut` function that updates out memory and queues all outgoing flows to `WorkSet` as necessary.
4. Implement `join` function that takes the union of two Memory objects, accounting for Domain values.
5. Implement `equal` function that checks if two Memory objects are equal, accounting for Domain values.

Setup

The skeleton code for Lab6 is located under `cis547vm/lab6/`. We will frequently refer to the top level directory for Lab 6 as `lab6` when describing file locations for the lab. Open the lab6 directory in VSCode following the Instructions from the Course VM document.

Step 1.

The following commands set up the lab, using the Cmake/Makefile pattern seen before.

One thing to note is the use of the `-DUSE_REFERENCE=ON` flag: this lab comprises two parts and this flag will allow you to focus on the features needed for Part 1 independently of Part 2.

```
/lab6$ mkdir build && cd build
/lab6/build$ cmake -DUSE_REFERENCE=ON ..
/lab6/build$ make
```

Among the files generated, you should now see `DivZeroPass.so` in the `lab6/build` directory.

We are now ready to run our bare-bones lab on a sample input C program.

Step 2

Before running the pass on a test program, we need to generate the LLVM IR code for it.

The `clang` command generates LLVM IR program from the input C program `test03.c`.

The `opt` command optimizes that LLVM IR program and generates an equivalent LLVM IR program that is simpler to process for the analyzer you will be building in this lab. In particular, the `-mem2reg` option promotes every AllocaInst to a register, allowing your analyzer to ignore handling pointers in this lab.

Later in Lab 7 you will extend this lab to handle pointers, and we will stop using `-mem2reg`.

```
/lab6/test$ clang -emit-llvm -S -fno-discard-value-names -Xclang -disable-O0-optnone -c -o test03.ll test03.c
/lab6/test$ opt -mem2reg -S test03.ll -o test03.opt.ll
```

Step 3

Similar to former labs, you will implement your analyzer as an LLVM pass, called `DivZeroPass`.

Then you will use the `opt` command to run this pass on the optimized LLVM IR program as follows:

```
/lab6/test$ opt -load ../build/DivZeroPass.so -DivZero -disable-output test03.opt.ll > test03.out 2> test03.err
```

Upon successful completion of this lab, the output in `test/test03.out` should be as follows:

```
Running DivZero on main
Potential Instructions by DivZero:
  %div1 = sdiv i32 %div, %div
```

The debug output of your program (printed using `errs()`) will be available in the `test/test03.err` file.

Format of Input Programs

To reduce the complexity of the lab we restrict the set of instructions that your analysis must handle. We assume that the input programs for this lab may only use the following subset of the C language:

- All values are integers (i.e. no floating points, pointers, structures, enums, arrays, etc). You can ignore other types of values.
- The program may have assignments, signed and unsigned arithmetic operations (+, -, *, /), and comparison operations (<, <=, >, >=, ==, !=). All the other instructions are considered to be nop.
- The program may have if-statements and loops.
- User inputs are only introduced via the set of functions where the provided `isInput` function returns `True`. You can ignore other call instructions to other functions.

Lab Instructions

A full-fledged static analyzer has three components:

1. An abstract domain
2. Transfer functions for individual instructions that evaluates the instruction using abstract domains.
3. Combining analysis results of individual instructions to obtain analysis results for entire functions or programs.

In part 1 of the lab, we will focus only on implementing item 2, and only for the limited subset of instructions as described above.

More concretely, your task is to implement how the analysis evaluates different LLVM IR instructions on abstract values from a provided abstract domain, defined in `Domain.h`.

In part 2 of the lab, we will focus on implementing item 3, to combine the results of individual transfer functions to get an intra-procedural, flow-sensitive, path-insensitive Divide-by-Zero analysis. Later on in Lab 7 you will further extend on item 3 to use the results of Pointer Analysis.

We have provided a framework to build your division-by-zero static analyzer. The framework is composed of files `Domain.cpp`, `Transfer.cpp`, `ChaoticIteration.cpp` and `DivZeroAnalysis.cpp` under `lab6/src/`.

Additionally, you have been provided with `src/Utils.cpp` which defines a few useful functions:

- `variable` takes a `Value` and returns string. This string is used as the key in the Memory maps stored in `InMap` and `OutMap`.
- `getOrExtract` takes a `Memory` and a `Value` and returns the `Domain` corresponding to `Value` in `Memory`, if not found then it tries to extract the `Domain` from the instruction itself.
- `printMemory`, `printInstructionTransfer` and `printMap` will print various debug information to `stderr`.

Part 1: The Check and Transfer Functions

Step 1

Refresh your understanding about program abstractions by reading the article on A Menagerie of Program Abstractions.

Once you have a good understanding of abstract domains, study the `Domain` class to understand the abstract domain that we have defined for you to use in this lab. The files `include/Domain.h` and `src/Domain.cpp` include the abstract values and operations on them. These operations will perform an abstract evaluation ***without running the program***. As described in the article, we have defined abstract operators for addition, subtraction, multiplication and division.

An important part of this analysis is realizing that you are never actually running the program. This means that when you go to evaluate an instrution such as:

```
%cmp = icmp slt i32 %x, %y
```

The Domain of `%cmp` is not determined by the runtime values of `%x` and `%y` but by the evaluation of their individual Domains with respect to the comparison instruction. So, more concretely, if the Domain of `%x` is `Domain::Zero` and the Domain of `%y` is `Domain::Zero`, since the less than comparison would be considered **False When Equal**, the resulting Domain would be `Domain::Zero`.

Step 2

Inspect `DivZeroAnalysis::runOnFunction` to understand how, at a high-level, the compiler pass performs the analysis:

```
bool DivZeroAnalysis::runOnFunction(Function &F) {
  outs() << "Running " << getAnalysisName() << " on " << F.getName() << "\n";

  // Initializing InMap and OutMap.
  for (inst_iterator Iter = inst_begin(F), E = inst_end(F); Iter != E; ++Iter) {
    auto Inst = &(*Iter);
    InMap[Inst] = new Memory;
    OutMap[Inst] = new Memory;
  }

  // The chaotic iteration algorithm is implemented inside doAnalysis().
  doAnalysis(F);

  // Check for Errors in the Function;
  ...
}
```

The procedure `runOnFunction` is called for each function in the input C program that the compiler encounters during a pass. Each instruction `I` is used as the key to initialize a new `Memory` object in the global `InMap` and `OutMap` hash maps. These maps are described in more detail in the next step, but for now you can think of them as storing the abstract values of each variable before and after an instruction. For example, the abstract state might store facts like "*at the point before instruction i, the variable x is positive*". Since `InMap` and `OutMap` are global, feel free to use them directly in your code.

Once the **In** and **Out** Maps are initialized, `runOnFunction` calls `doAnalysis`: a function that you will implement in Part 2 to perform the chaotic iteration algorithm. For Part 1, you can assume that it simply calls `transfer` using the appropriate `InMap` and `OutMap` maps.

So, at a high level, `runOnFunction` will:

1. Initialize the **In** and **Out** maps.
2. Fill them using a chaotic iteration algorithm.
3. Find potential divide by zero errors by using the `InMap` entries for each divide instruction to check whether the divisor may be zero.

Step 3

Understand the memory abstraction in the provided framework. For each `Instruction`, `DivZeroAnalysis::InMap` and `DivZeroAnalysis::OutMap` store the *abstract state* before and after the instruction, respectively. An abstract state is a mapping from LLVM variables to abstract values; in particular, we have defined `Memory` as a `std::map<std::string, Domain *>`. Since we refer to variables as `std::string`, we have provided an auxiliary function named `variable` that encodes an LLVM `Value` into our internal string representation for variables. Note that an `Instruction` is also a `Value`. For example, consider the following LLVM program. We have shown the abstract state, denoted **M**, before and after each instruction:

ID	Instruction	Before Instruction	After Instruction
`I1`	`%x = call i32 ( ... ) @input()`	`{ }`	`{ %x: T }`
`I2`	`%y = add i32 %x, 1`	`{ %x: T }`	`{ %x: T, %y: T }`

In the first instruction `I1`, we assign an input integer to variable `%x`. In the abstract state, we use an abstract value **T** (also known as "top" or `MaybeZero`) since the value is unknown at compile time. Instruction `I2` updates the abstract value of `%y` that is computed using the abstract add operation (denoted +) on the abstract value of `%x`. Note that, in the LLVM framework, the object for an assignment instruction (e.g., call, binary operator, icmp, etc.) also represents the variable it defines (i.e. its left-hand side). Therefore you will use the objects for instructions `I1` and `I2` to refer to variables `%x` and `%y`, respectively, in your implementation. For example, `variable(I1)` will refer to `%x`.

Step 4

Now that we understand how the pass performs the analysis and how we will store each abstract state, we can begin implementation.

First, you will implement a function `DivZeroAnalysis::transfer`, found in `src/Transfer.cpp`, to populate the `OutMap` for each instruction. In particular, given an instruction and its incoming abstract state (`const Memory *In`), `transfer` should populate the outgoing abstract state (`Memory *NOut`) which is derived from the appropriate implementation of `eval`.

The `Instruction` class represents the parent class of all *types* of instructions. There are many subclasses of `Instruction`. In order to populate the `OutMap`, each type of instruction should be handled differently.

Recall for this lab you should handle:

1. Binary Operators (add, mul, sub, etc)
2. CastInst
3. CmpInst (icmp, eq, ne, slt, sgt, sge, etc)
4. user input via `getchar()` - recall from above that this is handled using `isInput()` from `src/Transfer.cpp`.

LLVM provides several template functions to check the type of an instruction. We will focus on `dyn_cast<>` for now. In this example, we check if the `Instruction I` is a BinaryOperator.

```
if (BinaryOperator *BO = dyn_cast<BinaryOperator>(I)) {
  // I is a BinaryOperator, do something
}
```


At runtime, `dyn_cast` will return `I` *casted* to a `BinaryOperator` if possible, and null otherwise.

At this point, your `eval( ... )` implementation will take the instruction and determine how this instructions Domain is affected by the operation. For example,


```
%add = add nsw i32 %x, %y
```

Assuming `%x` has a domain of `Domain::Zero` and `%y` has a domain of `Domain::NonZero`, Since `%y` can take any value that is not zero (positive or negative) the resulting domain for `%add` will be determined by the addition of `Zero` to a `NonZero` value. Consequently, the domain for `%add` is determined to be `Domain::NonZero`. In this way, the `DivZeroAnalysis::transfer` function updates the `OutMap` for the associated action of a given `Instruction`.

The `eval` function for `PhiNode` has been implemented for you and offers an example of how to use the utility function `getOrExtract` as well as `Domain::join`.

Working with LLVM PHI Nodes. For optimization purposes, compilers often implement their intermediate representation in *static single assignment*(SSA) form and LLVM IR is no different. In SSA form, a variable is assigned and updated at exactly one code point. If a variable in the source code has multiple assignments, these assignments are split into seperate variables in the LLVM IR and then *merged* back together. We call this merge point a **phi node**.

To illustrate phi nodes, consider the following code:

```
int f() {
  int y = input();
  int x = 0;
  if (y < 1) {
  # then
    x++;

  } else {
    x--;
  }
  # end
  return x;
}
```

```
entry:
  %call = call i32 ( ... ) @input()
  %cmp = icmp slt i32 %call, 1
  br i1 %cmp, label %then, label %else
then:                         ; preds = %entry
  %inc = add nsw i32 0, 1  ; equates to x++ to the left
  br label %if.end
else:                         ; preds = % entry
  %dec = add nsw i32 0, -1 ; equates to x-- to the left
  br label %end
end:                          ; preds = %else, %then
  %x = phi i32 [%inc, %then ], [%dec, %else ]
  ret i32 %x
```

Depending on the value of `y`, we either take the left branch and execute `x++`, or the right branch and execute `x--`. In the corresponding LLVM IR, this update on `x` is split into two variables `%inc` and `%dec`. `%x` is assigned after the branch executes with the `phi` instruction; abstractly, `phi i32 [ %inc, %then ], [ %dec, %else ]` says assign `%inc` to `%x` if the then branch is taken, or `%dec` to `%x` if the else branch was taken.

Here is a piece of sample code to help you address phi nodes, as the specifics are beyond this course; however, feel free to read up more on SSA if these kinds of compiler details pique your interest.

```
Domain *eval(PHINode *Phi, const Memory *InMem) {
  if (auto ConstantVal = Phi->hasConstantValue()) {
    return new Domain(extractFromValue(ConstantVal));
  }

  Domain *Joined = new Domain(Domain::Uninit);

  for (unsigned int i = 0; i < Phi->getNumIncomingValues(); i++) {
    auto Dom = getOrExtract(InMem, Phi->getIncomingValue(i));
    Joined = Domain::join(Joined, Dom);
  }
  return Joined;
}
```

Step 5

Implement the `DivZeroAnalysis::check` function found in `src/DivZeroAnalysis.cpp`. This function checks an `Instruction` to determine if a division-by-zero is **possible**. Any Instruction that is a **signed** or **unsigned** division instruction with a divisor whose `Domain` is either `Domain::Zero` or `Domain::MaybeZero` would be considered a potential divide-by-zero. You should use `DivZeroAnalysis::InMap` to decide if there is an error or not.

To test your `check` and `transfer` functions, we have provided a reference `doAnalysis` binary. In part 2, you will need to implement the `doAnalysis` function yourself, but for now you may test with our binary solution in order to make sure the functions you have implemented thus far are working correctly. Follow these steps to compile using the reference binary:

```
/lab6/build$ rm CMakeCache.txt
/lab6/build$ cmake -DUSE_REFERENCE=ON ..
/lab6/build$ make
```

As we demonstrated in the Setup section, run your analyzer on the test files using `opt`:

```
/lab6/test$ opt -load ../build/DivZeroPass.so -DivZero -disable-output test03.opt.ll
```

If there is a divide-by-zero error in the program, your output should be as follows:

```
Running DivZero on Main
Instructions that potentially divide by zero:
  %div = sdiv i32 1, 0
```

Part 2 : Putting it all together - dataflow analysis

Now that you have code to populate in and out maps and use them to check for divide-by-zero errors, your next step is to implement the chaotic iteration algorithm in function `doAnalysis` found in `src/ChaoticIteration.cpp`.

First, review the dataflow analysis lecture content. In particular, study the reaching definition analysis and the chaotic iteration algorithm. Informally, a dataflow analysis creates and populates an **IN** set and an **OUT** set for each node in the program's control flow graph. The *flowIn* and *flowOut* operations are repeated until the algorithm has reached a fixed point.

More formally, the `doAnalysis` function should maintain a `WorkSet` that holds nodes that "need more work." When the `WorkSet` is empty, the algorithm has reached a fixed point. For each instruction in the `WorkSet` your function do the following:

1. Perform the *flowIn* operation by joining all **OUT** sets of incoming flows and saving the result in the **IN** set for the current instruction. Here, you will use the entries from the `InMap` and `OutMap` that you populated in Part 1 as the **IN** and **OUT** sets.
2. Apply the `transfer` function that you implemented in Part 1 to populate the **OUT** set for the current instruction.
3. Perform the *flowOut* operation by updating the `WorkSet` accordingly. The current instruction's successors should be added only if the **OUT** set was changed by the `transfer` function.

Here is an example of how the `WorkSet` needs to be loaded with instructions as well as introducing the llvm::SetVector container, feel free to use this code as part of your implementation:

```
void DivZeroAnalysis::doAnalysis(Function &F) {
  SetVector<Instruction *> WorkSet;
  for(inst_iterator I = inst_begin(F), E = inst_end(F); I != E; ++I) {
    WorkSet.insert(&(*I));
  }
  // ...
}
```

For this lab, we do not need to maintain an explicit control flow graph; LLVM already maintains one in its internals. In order for you to focus on the dataflow portion of this assignment, we have provided two auxiliary functions `getSuccessors` and `getPredecessors` (defined in `include/DivZeroAnalysis.h`) that lookup and return the successors and predecessors for a given LLVM `Instruction`.

You will next implement the various parts of the chaotic iteration algorithm.

Step 1

In `flowIn`, you will perform the first step of the reaching definitions analysis by taking the union of all **OUT** variables from all predecessors of `I`. You may find the `getPredecessors` method in `src/ChaoticIteration.cpp` to be helpful here. This should be done in the following function that is templated for you below:

- `void DivZeroAnalysis:flowIn(Instruction *I, Memory *In)`

Given an `Instruction I` and its **IN** set of variables, Memory `In`, you will need to union the **IN** with the **OUT** of every predecessor of `I`. In order to take the union of two memory states, you will need to implement the join function templated below:

- `Memory* join (Memory *M1, Memory *M2)`

Within this function, you will also need to consider the `Domain` values when merging these `Memory` objects. Refer to the abstract domain on why this is necessary. Recall that a `join` operation for combining two abstract values is defined in the `Domain` class.

Step 2

Call the `transfer` function that you implemented in Part 1 to populate the **OUT** set for the current instruction.

Step 3

In `flowOut`, you will determine whether or not a given instruction needs to be analyzed again. This should be done in the following function that is templated for you below:

- `void DivZeroAnalysis::flowOut(Instruction *I, Memory *Pre, Memory *Post, SetVector<Instruction *> &WorkSet)`

Given an `Instruction I`, you will analyze the *pre-transfer* Memory `Pre` and the *post-transfer* Memory `Post`. If there exists a change between the memory values after the `transfer` is applied, you will need to submit the instruction `I` for additional analysis. To determine if the memory has changed during the `transfer` function, you will implement the function `equal`:

- `bool equal(Memory *M1, Memory * M2)`

In this function, you will again consider the `Domain` values when determining whether two `Memory` objects are equal. Recall that an `equal` operation to evaluate equality between two abstract values is defined in the `Domain` class.

Lastly, in `flowOut` be sure that you update the `OutMap` for instruction `I` to include values in `Post`.

Step 4

Recall in Part 1, a reference `doAnalysis` could be used to verify your `check` and `transfer` implementations. Now that you're writing your own version of `doAnalysis`, you may need to rebuild the pass without the reference. Follow these steps to compile using your implementation:

```
/lab6/build$ rm CMakeCache.txt
/lab6/build$ cmake ..
/lab6/build$ make
```

Upon completing the above steps, your analysis should produce 2 output files.

1. `test.out`, where test is the program you are testing, is a condensed version of the results with just the instruction that has a potential divide-by-zero operation.
2. `test.err` is a complete report including any instructions with potential divide-by-zero operations as well as the final state of the `InMap` and `OutMap` for each instruction being reviewed.

Your output will be formatted like this:

```
Dataflow Analysis Results:
Instruction:   %cmp = icmp ne i32 0, 0
In set:

Out set:
    [ %cmp     ↦ Zero      ]

Instruction:   br i1 %cmp, label %if.then, label %if.end
In set:
    [ %cmp     ↦ Zero      ]
Out set:
    [ %cmp     ↦ Zero      ]

Instruction:   %div = sdiv i32 1, 0
In set:
    [ %cmp     ↦ Zero      ]
Out set:
    [ %cmp     ↦ Zero      ]
    [ %div     ↦ Uninit    ]

Instruction:   br label %if.end
In set:
    [ %cmp     ↦ Zero      ]
    [ %div     ↦ Uninit    ]
Out set:
    [ %cmp     ↦ Zero      ]
    [ %div     ↦ Uninit    ]

Instruction:   ret i32 0
In set:
    [ %cmp     ↦ Zero      ]
    [ %div     ↦ Uninit    ]
Out set:
    [ %cmp     ↦ Zero      ]
    [ %div     ↦ Uninit    ]
```

Submission

Once you are done with the lab, you can create a `submission.zip` file by using the following command:

```
lab6$ make submit
...
submission.zip created successfully.
```

Then upload the submission file to Gradescope.