Assignment 2: Intra-Procedural Constant Propagation

Introduction to Program Analysis and Optimization

January 28, 2024

1 Constant Propagation

Constant propagation analysis tracks the values of variables at compile-time by traversing the program's control flow graph to determine which variables hold constant values at different points in the program. Constant propagation optimization is a compiler optimization technique that replaces variables with their constant values identified during the analysis phase.

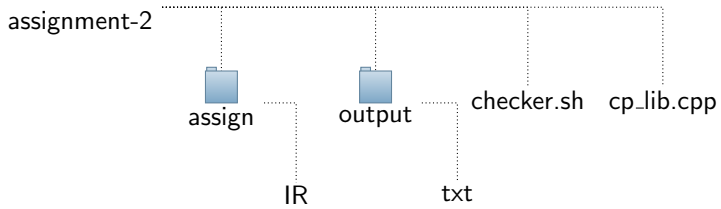
In this assignment, you will be implementing the **analysis** phase of constant propagation, using LLVM compilation framework.

2 Example

Figure 1 shows a naive C code doing addition and multiplication operations and its corresponding LLVM IR. Lines 3 to 6 in the IR code are `alloca` instructions, indicating the allocation of variables. Line 7 and 8 are `store` instructions that are used to store constant values to variables `a` and `b`. Line 9 and 10 are the load instructions that are loading values of the variable `a` and `b` in the register `%0` and `%1`, respectively. Single Static Assignment (SSA) is a crucial feature of LLVM IR that simplifies code analysis and transformation. Every variable in SSA form is assigned exactly once, and every use of the variable refers to that single definition. Hence, to use `a` and `b` in `add` instruction (Line 11), `a` and `b` must be redefined (`load` instructions). The result of addition was then stored in variable `c`. Since `a` and `b` contained constant values, your analysis should be able to statically identify the values of variable `c` and `d` without actually executing the program. In this case, the analysis should be able to identify that the value contained by variable `c` is 6.

3 Deliverables

The structure of the given template directory is mentioned above.



There are two folders, named `assign` and `output`. The former contains the IR codes on which you will run your constant propagation pass. The folder `output` will contain `text` files, with each `text` file containing constant values of each variable corresponding to each instruction in LLVM IR. See Figure 2 for a desired output of the given LLVM IR code in Figure 1. Each variable's domain is {TOP, $\mathbb{Z}$, BOTTOM}. The concepts of "TOP" and "BOTTOM" are the same as those explained in the class.

```c
int main()
{
    int a = 2;
    int b = 4;
    int c = a + b;
    int d = c * 3;
}
```

(a) Naive C code

```
define dso_local i32 @main() #0 {
entry:
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  store i32 2, i32* %a, align 4
  store i32 4, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, i32* %c, align 4
  %2 = load i32, i32* %c, align 4
  %mul = mul nsw i32 %2, 3
  store i32 %mul, i32* %d, align 4
  ret i32 0
}
```

(b) Corresponding LLVM IR

Figure 1: Example showing constant propagation on LLVM IR

```

  %a = alloca i32, align 4 --> %a=TOP
  %b = alloca i32, align 4 --> %b=TOP
  %c = alloca i32, align 4 --> %c=TOP
  %d = alloca i32, align 4 --> %d=TOP
  store i32 2, i32* %a, align 4 --> %a=2
  store i32 4, i32* %b, align 4 --> %b=4
  %0 = load i32, i32* %a, align 4 --> %0=2, %a=2
  %1 = load i32, i32* %b, align 4 --> %1=4, %b=4
  %add = add nsw i32 %0, %1 --> %add=6, %0=2, %1=4
  store i32 %add, i32* %c, align 4 --> %add=6, %c=6
  %2 = load i32, i32* %c, align 4 --> %2=6, %c=6
  %mul = mul nsw i32 %2, 3 --> %mul=18, %2=6
  store i32 %mul, i32* %d, align 4 --> %mul=18, %d=18
  ret i32 0
```

Figure 2: Example output

4 Additional Details

The marks distribution for the constant propagation assignment is as follows:
1. Correct Output on Public Test Cases. (30 pts)
2. Correct Output on Private Test Cases. (70 pts)

Here are some DOs and DONTs for the assignment.
DOs

- Use git commit to upload the assignment.
- Run the script in `checker.sh` file and submit the assignment only after receiving an "Accept" output from the script. It checks the naming conventions and folder structure. Note: `checker.sh` cannot validate the correctness of your assignment results.
- Clone the assignment repository inside the llvm-project folder.
- Write your pass only in the appropriate section of `cp_lib.cpp` file.
- Your output text file should have the same name as input IR files. For example, the output file corresponding to `file1.ll` should be named as `file1.txt`.

DONTs

- Do not submit assignment if the `checker.sh` gives "Rejected". Your assignment will not be evaluated if your directory structure or naming conventions do not match.
- Do not change the name of any files or folders.
- Do not edit any other things (e.g., name of the pass) in the `cp_lib.cpp` file.