Assignment 2: Intra-procedural Constant Propagation

Introduction to Program Analysis and Optimization

Deadline: 15th February, 2025

January 30, 2025

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 demonstrates the process of constant propagation in LLVM IR. Figure 1a shows the original LLVM IR code, where the program performs the addition of two integer variables, `a` and `b`, followed by a multiplication operation. Initially, the values of `a` and `b` are stored in memory (lines 5 and 6), and these values are then loaded into registers `%0` and `%1` (lines 9 and 10). The addition operation is performed on these registers, and the result is stored in `c` (line 11). The value of `c` is then loaded, multiplied by 3, and stored in d (lines 13–14).

Figure 1b) illustrates the outcome of constant propagation applied to the IR code. In this output, constant values for variables `a` and `b` are propagated through the instructions, allowing the analysis to directly compute the values of `c` and `d`. At each step, the constant values are tracked and updated in the output:

- `a` is set to 2 (Line 7, Figure 1b)
- `b` is set to 4 (Line 8, Figure 1b)
- The sum of `a` and `b` (`%add`) is computed as 6 and stored in `c` (Line 12, Figure 1b)
- The value of `c` is loaded as 6 (Line 13, Figure 1b)
- The multiplication of `c` by 3 (`%mul`) results in 18 (Line 14, Figure 1b)
- This result of multiplication is stored in `d` (Line 15, Figure 1b)

3 Deliverables

The structure of the given template directory is mentioned in Figure 2. There are two folders, named `assign` and `output`. The former contains the IR codes on which you will run your constant propagation

```
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
}
```

(a) Corresponding LLVM IR

```
define dso_local i32 @main() #0 {
entry:
  %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
}
```

(b) Example output

Figure 1: Example showing constant propagation on LLVM IR

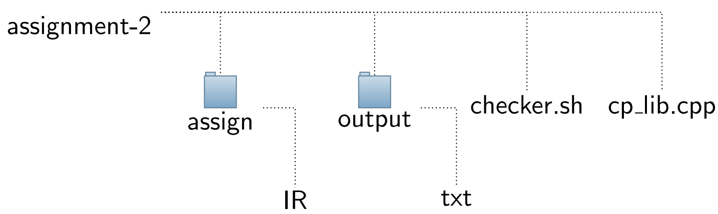

Figure 2: Directory Structure

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 1b for a desired output of the given LLVM IR code in Figure 1a. Each variable's domain is {TOP, $\mathbb{Z}$, BOTTOM}. The concepts of "TOP" and "BOTTOM" are the same as those explained in the class.

4 Additional Details

The marks distribution for the constant propagation assignment is as follows:

1. Correct Output on Public Test Cases $\longrightarrow$ 30 pts (Each Test Case 5 pts)
2. Correct Output on Private Test Cases $\longrightarrow$ 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`.

- Try to submit within the deadline. There is a late penalty (**20**%) for each day after the deadline.

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.
- Do not use the GPT tool to write the theory answers. There will be a heavy penalty for such behaviour.
- Do not try to copy from any online resource or from another student's assignment. In case of plagiarism, both the students (sink and source) will be seen as **equally** guilty.