Assignment 4: Inter-procedural Constant Propagation

Introduction to Program Analysis and Compiler Optimization

March 26, 2024

1 Inter-Procedural Constant Propagation

Constant propagation is an optimization technique used in compilers and interpreters to simplify expressions by substituting values of constants for their corresponding variables at compile-time. You have already implemented the LLVM pass for constant propagation; however, it was an intra-procedural analysis and without any transformation pass. In this assignment, you will implement an **inter**-procedural context **insensitive** and flow-sensitive analysis pass along with a **transformation** pass. The final output of your pass should be an optimized code with proper substitution of computation with correct constant values.

2 Example

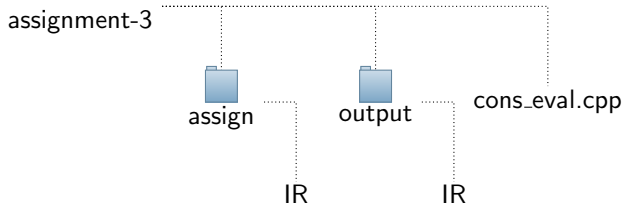
Consider the example depicted in Figure 1a. The function `main` invokes three distinct functions: `fun`, `foo`, and `bar`. Within `fun`, argument values are utilized to determine the value of the variable `a` (Line 13). As `fun` is invoked exclusively from a single callsite and the argument values are available at compile time, the redundant computations within `fun` (as illustrated in Figure 1b) can be easily eliminated.

Further, the function `foo` is invoked from `fun`, where an argument of `fun` and the computed variable `a` are passed as parameters to `foo`. Although `foo` is invoked from two distinct callsites with different argument values, the values read inside the function remain the same across various contexts; the argument `k`, which holds different values, is overwritten within `foo`. Consequently, optimization of the function `foo` is also feasible (as depicted in Figure 1b).

However, in contrast, the function `bar` is called from multiple callsites with differing argument values. Therefore, the computations within `bar` cannot be optimized during compile time as the arguments have no constants.

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 that are public test cases. On the given LLVM IR, you will run your optimization pass. The folder `output` will contain `ll` files, each containing the transformed IR.

```
void main()
{
    int l=40;
    fun(10, 100, 1000);
    bar(1000, 400, 300);
    foo(100, 400, 1000, l);
}
void fun(int i, int j, int k)
{
    bar(2000, i, 1000);
    int o = i *2;
    int q=2;
    int a = q * o * i;
    foo(100, a, k, 800);
}
void foo(int i, int j, int k, int x)
{
    x = i+j+k;
    printf("%d", x);
}
void bar(int i, int j, int k)
{
    j = (k*i)/2;
    printf("%d", j);
}
```

(a) Example

```
void main()
{
    fun(10, 100, 1000);
    bar(1000, 400, 300);
    foo(100, 400, 1000, 40);
}
void fun(int i, int j, int k)
{
    bar(2000, 10, 1000);
    foo(100, 400, 1000, 800);
}
void foo(int i, int j, int k, int x)
{
    printf("%d", 1500);
}
void bar(int i, int j, int k)
{
    j = (k*i)/2;
    printf("%d", j);
}
```

(b) Output to the given example

4 Hints

- You may need to study the cpp STL containers (std::set, std::map, std::vector).
- You may need to reuse your codes from assignment-II (constant propagation).

5 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)
- The test cases may contain (1) Data type: boolean, integer, floats, (2) Operation: arithmetic operation, logical operation and (3) miscellaneous: cast instruction, load instruction, store instructions.
- You need not handle global variables and macros.
- You must change the variable value to a constant wherever required; such transformation may also be needed for arguments passed to a function at call sites. It is not straightforward, so we recommend that you start early.
- The test case files have debug information in them. If you want to utilize that, please build your LLVM in debug mode. This may take time. So plan early.

Here are some DOs and DONTs for the assignment.
DOs

- Use git commit to upload the assignment.
- Clone the assignment repository inside the llvm-project folder.
- Write your pass only in the appropriate section of `cons_eval.cpp` file.
- Your output IR file should have the same name as the input IR files. For example, the output file corresponding to `file1.ll` should be named as `file1.ll`.

DONTs

- Do not change the name of any files or folders.
- Do not edit any other things (e.g., name of the pass) in the `cons_eval.cpp` file.
- Do not use `mem2reg` pass; you must not also use the constant propagation pass from `LLVM`. Doing the same will come under `plagiarism`.