COSC 461/561

cpass - copy propagation pass in LLVM

Your assignment is to implement local and global copy propagation as an LLVM compiler pass. Copy propagation is a transformation that, given an assignment $x \leftarrow y$ for some variables x and y, replaces later uses of x with uses of y if no intervening instructions have changed the values of either x or y. For example, consider the following C program:

```
int main() {
  int i = 10;
  int j = 0;

  j = i;
}
```

Copy propagation aims to propagate the copy $i \leftarrow 10$ to the later use of i in $j \leftarrow i$. Traditionally, this pass will replace uses of the operand that is stored into (called the destination) with the operand that is being stored (called the source) in subsequent instructions if no intervening instructions change the values of either the source or destination operands.

In the LLVM IR, copy instructions are translated as a pair of instructions: a load instruction that loads the source value from a location, and a store instruction that stores this value to the destination location. For this assignment, you should only consider store instructions as new copy instructions, each of which defines a new <*dst*, *src*> pair. Load instructions may use a location from a previous store, but are not to be considered new copy instructions. However, in cases where the value of a load is known due to the presence of an earlier store instruction, your pass should eliminate the load instruction and associate the known value with the destination of the load. In this way, the known value can be propagated to subsequent instructions that use the destination of the load instruction.

As an example, consider the unoptimized LLVM IR for the program above:

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 10, i32* %1, align 4
  store i32 0, i32* %2, align 4
  %3 = load i32, i32* %1, align 4
  store i32 %3, i32* %2, align 4
  ret i32 0
}
```

Your copy propagation pass should identify the stores of the constants 10 and 0 to $*\%1$ and $*\%2$ as new copy instructions. Since $*\%1$ is used in a subsequent load instruction, the pass will associate the value 10 with the destination of the load (%3) and remove the load instruction. Additionally, the pass will then replace subsequent uses of the register %3 with the value 10. The code after the copy propagation pass is shown below:

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 10, i32* %1, align 4
  store i32 0, i32* %2, align 4
  store i32 10, i32* %2, align 4
  ret i32 0
}
```

Your task for this assignment is to implement both local and global propagation as an LLVM optimization pass. Your pass will take LLVM IR as input and produce LLVM IR with copy propagation applied as output.

A detailed description of the traditional copy propagation algorithm, along with a description of how to collect the data flow analysis that is necessary to implement global copy propagation, are available in the book *Advanced Compiler Design and Implementation* by Steven S. Muchnick. The relevant pages are available in `copy_prop_muchnick.pdf` in your project directory and on the Canvas site.

The *cpass* starter directory includes several important files that you will need to complete your assignment. First, some starter code for the copy propagation pass is available in `copy_prop/copy_prop.cpp`. You will only need to edit this file to implement your pass. The file contains classes and data structures that we used in our solution as well as function prototypes and comments to help get you started.

To build the pass in `copy_prop.cpp`, you should use the `build.sh` script, as below:

```
> ./build.sh
```

This command will build your pass and store the resulting `libcopy_prop.so` file in your build directory. To run your pass with some example input, you need to use the `run_opt.sh` script, as below:

```
> ./run_opt.sh TEST --opt [INPUT]
```

with INPUT set to be the name of the one of the inputs in the inputs directory. For example, you can run your pass with `ex2.c` as below:

```
> ./run_opt.sh TEST --opt ex2
```

This will create an `ex2.ll` file in the `llvm_ir/test` directory in your project directory. The unoptimized LLVM IR for the ex2 input is in `llvm_ir/unoptimized`. If you want to compare your solution with our solution, you can use the REF option with `run_opt.sh`, as below:

```
> ./run_opt.sh REF --opt ex2
```

The output of the solution pass will be stored in `llvm_ir/ref`.

We recommend you build and test local copy propagation before moving on to global copy propagation. Additionally, follow the text and make sure you understand the algorithm before attempting to implement this pass. There are printing routines implemented for you in the starter code. Use these routines to debug your code and see examples of how to use the data structures in the starter code.

Finally, you will also submit a project report, similar to the reports you submitted for previous projects (one to two pages describing what you set out to do, your approach, and any issues you had). You should upload a gzipped tar file (created with `tar cvzf ...`) with your source files and a pdf of your report to the Canvas course website before midnight on the project due date. Point breakdown is below:

- local copy propagation (40)
- data flow analysis (40)
- global copy propagation (10)
- project report (10)