COSC 461/561

cpass - store propagation pass in LLVM

Your assignment is to implement local and global versions of an an LLVM compiler pass, called *store propagation*, that aims to remove unnecessary load instructions in the LLVM IR. The pass should propagate values stored in a known location (say x) forward to instructions that use x, if no intervening instructions have changed the value stored in x.

Value propagation is related to the traditional compiler transformation known as *copy propagation*. Given an assignment $x \leftarrow y$ for some variables x and y, copy propagation replaces later uses of x with uses of y if no intervening instructions have changed the values of either x or y. For example, in the following C program:

```
int main(int a) {
  int i, j;

  i = 10;
  j = 0;

  j = i;
}
```

Copy propagation will propagate the copy $i \leftarrow 10$ by replacing the use of i in the instruction $j \leftarrow i$ with the constant value 10.

For the store propagation pass, you will implement a version of copy propagation that considers store instructions in the LLVM IR as copy instructions in the traditional pass. For each store instruction it encounters, store propagation creates a new <*dst*, *src*> pair, where *dst* is the location being stored into and *src* is the value being stored. The pass then propagates the effects of each store to subsequent instructions by replacing loads from the *dst* location with the *src* value, if no intervening instruction changes the value stored in *dst*.

Additionally, in cases where the value loaded by a load instruction is already known due to the presence of an earlier store instruction, store propagation will 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
}
```

Here, store propagation should identify the stores of the constants 10 and 0 to $*\%1$ and $*\%2$ as new <*dst*, *src*> pairs. 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 store 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 store propagation as an LLVM optimization pass. Your pass will take LLVM IR as input and produce LLVM IR with store 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 store 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 store propagation pass is available in the `store_prop` directory in `store_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 `store_prop.cpp`, you should use the `makefile`, as below:

```
> make
```

This command will build your pass and store the resulting `libstore_prop.so` file in your build directory. To run your pass with some example input, you need to use the `opt` target with the INPUT variable set to be the name of the one of the inputs in the inputs directory, as below:

```
> make opt_ll INPUT=input1
```

This will create an `input1.ll` file in the `ir/opt` directory in your project directory. You can compare the code generated by your pass to unoptimized LLVM IR using the `unopt_ll` target, as below:

```
> make unopt_ll INPUT=input1
```

The unoptimized LLVM IR is stored in `ir/unopt`. Similarly, you can compare the code generated by your pass to code generated by our solution using the `ref_opt_ll` target. Code compiled with the `ref_opt_ll` target is stored in `ir/ref_opt`.

Additionally, you can create executable files for each input without applying any optimization pass, applying only the `store_prop` optimization pass that you write for this assignment, and applying only the reference solution for the `store_prop` optimization pass, by using the `unopt_exe`, `opt_exe`, and `ref_opt_exe` targets, respectively.

We recommend you build and test local store propagation before moving on to global store 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.

For submission, you should upload your completed `store_prop.cpp` file to the Canvas course website before midnight on the project due date. Point breakdown is below:

- local store propagation (40)
- data flow analysis (50)
- global store propagation (10)