CSE 501
Programming Assignment 2

Haichen Shen

Building

An ant build file is included in the tar. The default target will build the project and generate the jar file (opt.jar).

How to use

A script run.sh is provided. The options supported are listed below:

```
run.sh <filename> [-opt=<optimize>] [-backend=<backend>]

Optimization supported options:
ssa     SSA optimization
cp      Constant propagation optimization (depends on SSA)
vn      Value numbering optimization (depends on SSA)

Backend supported options:
asm     Assembly code (default)
cfg     Control flow graph
ir      Intermediate representation
ssa     SSA code
report  Report
```

The input file should be .start file. I implemented three different optimization: SSA, constant propagation and value numbering (global common subexpression elimination). And the format of backend has the following options:

Assembly code	The code could run in the start interpreter
Intermediate representation (IR)	Slightly different from assembly code, more human-readable
Control flow graph (CFG)	Basically same as IR; include the predecessors, successors, immediate dominator and children in the dominator tree

SSA	SSA format
Report	Generate the report of optimization

Implementation

Intermediate Representation (IR)

I use a different type of IR which is more human-readable. It looks quite similar to the assembly code, but without offset for each variable. And I move the lhs of all statement to the real left hand side, for example:

```
instr 15: move (10) i#-4      ⇒      i := (10)
instr 16: add i#-4 10 :int    ⇒      (6) := add i 10
instr 17: load (9) :int       ⇒      (17) := load (9)
```

Translate to SSA

I implemented the method taught in the class. Before I wrote the program, I firstly add two data structure for phi node and entry statement. The phi node is a combination of a phi statement and a move statement. For instance, it contains:

```
instr 15: (15) := phi i$1 i$2
instr 16: i$3 := (15)
```

But it looks the same as i$3 := phi i$1 i$2 from outside.

Besides, the entry statement is a declaration for all the parameters and local variables. With it, it's easier to rename the variables and analyze the program. If the method has two parameter x and y, and one local variable i, then the entry statement will be:

```
instr 20: entry x$0 y$0 i$0
```

The entry statement is also included in the IR format.

The work flow of program is:

1. Generate the dominance frontier for each basic block.
2. Place a phi node at the dominance frontier.
3. Rename the variables.
4. Dead code elimination.

I add an additional step to SSA translation, which is dead code elimination. By using the Def-Use analysis, I recursively eliminate the variables and phi nodes which are never used.

Translate out of SSA

Since I don't use the copy folding optimization and loop invariant code motion, the brute force way to remove the phi node works. (Actually, I also read the paper how to translate out of SSA when brute force way doesn't work. When the copy folding is committed to the program, we need to split the critical edges and add temporary variables, referring to [1]. And after using loop invariant code motion optimization, we need to calculate the liveness of each variables and

allocate by coloring [2].) However, I found it complicated to implement them. So I didn't implement it and both two optimization mentioned before.
Here what I did is replacing a k-input phi node by k ordinary assignments, one at the end of each control flow predecessor of X. After removed all phi nodes, I renamed the variable and adjust the stack allocation. I use their SSA names as their new names, but assign the stack offset to each variable in the SSA. At last, I reordered the statement index.

Constant Propagation Optimization

I implemented the sparse conditional constant algorithm [3], which is capable of propagating constants, eliminating the block and statement which will never be executed. So the side effect of this algorithm is dead code elimination.

Value Numbering

I implemented the global value numbering [4], which could replace a redundant computation in any block dominated by the first occurrence as well as replace the redundant phi node. I didn't simply the expression which is part of the constant propagation job.

Result

First, I verify the correctness of my translation into and out of SSA with both optimization on. For all the examples the optimized program outputs the same result as the original one.

Here are part of report I get from my optimization program:
For mmm.start:
    Function: alloc
    Number of constants propagated: 1
    Number of expressions eliminated: 1
    Function: main
    Number of constants propagated: 13
    Number of expressions eliminated: 30
For prime.start:
    Function: main
    Number of constants propagated: 5
    Number of expressions eliminated: 8
For regslarge.start:
    Function: main
    Number of constants propagated: 15456
    Number of expressions eliminated: 0
For struct.start:
    Function: main
    Number of constants propagated: 4
    Number of expressions eliminated: 23

Generally the optimization could propagate several constants and eliminate several redundant

expressions. For regslarge.start, I check the source code and find it have a lot of redundant expressions all of which are constants. Therefore, the optimization could propagate such a large number of constants.

Summary

- Use ./run.sh <filename> [-opt=<optimize>] [-backend=<backend>] to dump the optimized code or report.
- The program could translate the assembly code into SSA form and get back out of it.
- The program use the sparse conditional constant algorithm and global value numbering to optimize the code.

Reference

[1] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. Softw., Pract. Exper. 28(8): 859-881 (1998)
[2] Chaitin, G. J. Register allocation and spilling via graph coloring. In proceedings of the SIGPLAN 82 Symposium on Compiler Construction. SIGPLAN Not. (ACM) 17, 6 (June 1982), 98-105.
[3] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 2 (April 1991), 181-210
[4] P. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. Software—Practice and Experience, 27(6), June 1996.