6.035 Project 4 Writeup (le03)

Division of Labor
As with previous projects, we tended to work as a group and often resorted to pair coding. Saif took the lead on algebraic simplification and Josh on GCSE. Saji worked on local CSE and David handled the worklist implementation. We all participated in testing and documentation.

Clarifications and Assumptions
We did not make any major assumptions over the course of this project. In fact, the majority of our design was based off the lecture material.

Design

Introduction
The primary goal of this assignment was to implement global common subexpression elimination. In order to accomplish this, we built a framework for dataflow analysis that enables the identification and transformation of basic code blocks. Based on local and global states, optimizations are integrated by stitching code back together. Assembly is then generated.

The common subexpression optimization is available through the compiler with `-opt cse`.

Dataflow Analysis Framework
Overview
The fundamental component of our framework is a basic block. Each block is defined as a set of statements leading up to, but not including, a jump or label. Statements within a block are guaranteed to execute together. We briefly considered the idea of each statement composing a basic block but ultimately decided it may be too problematic to debug. The underlying mid-level representation, a linked-list of nodes closely resembling the final assembly, does *not* change. Blocks are defined by the start and end nodes in the list and the analysis framework queries the block for the node to stop at (not necessarily the end of the mid-level list).

A doubly-linked data structure of basic blocks is used to maintain a graph-like representation of a program. Transfer functions are then applied using the worklist algorithm described in lecture. We use Java generics to write the generic framework, using state and transfer function interfaces that are implemented by specific optimizations.

Common Subexpression Elimination
Global vs. Local
We perform forward analysis using our dataflow framework and reason about the state of each basic block through maintained sets of mappings between values, local expressions, and nodes in our IR. Each transfer function updates the global state in terms of available expressions. Simultaneously, it transforms the block for local CSE optimizations.

Nodes added through transformations use a new subclass to differentiate them from others. When a block is processed multiple times (because the out state of its predecessor has changed), these subclasses are used to prevent global CSE from transforming previous local

CSE optimizations.

It is difficult to track what field declarations are modified by a function call. As such, we made a design decision to forgo some potential optimizations and remove all saved references to field declarations when a function is called. It is possible to track all fields that a function can potentially modify and only clear references to those fields, but due to time constraints we refrained from doing so.

Example
To demonstrate global CSE, we present the following simple assembly example:

```
class Program {
        void main() {
                int a, b, c, d, e;
                boolean z;
                z = true;
                if (z) {
                        c = a + b;
                } else {
                        d = a + b;
                }
                e = a + b;
        }
}
```

We would expect CSE to create a temporary variable that both branches of the if/else statement share. Then, the variable `e` would just reference it. We can see this is the case from the assembly (portions omitted for brevity):

```
   ...
fi1:
   MOV        R11, qword [ RBP - 16 ]        ; <MidLoadNode: reg29,<MidLocalVarDeclNode: b > >
   MOV        qword [ RBP - 128 ], R11       ; <MidSaveNode: reg29 >
   MOV        R11, qword [ RBP - 8 ]         ; <MidLoadNode: reg27,<MidLocalVarDeclNode: a > >
   MOV        qword [ RBP - 136 ], R11       ; <MidSaveNode: reg27 >
   MOV        R11, qword [ RBP - 80 ]        ; <MidLoadNode: reg39,<MidTempDeclNode: t16 > >
   MOV        qword [ RBP - 144 ], R11       ; [OPT] <MidSaveNode: reg39 >
   MOV        R11, qword [ RBP - 144 ]       ; <MidLoadNode: reg35,<MidTempDeclNode: t34 > >
   MOV        qword [ RBP - 40 ], R11        ; <MidSaveNode: reg35 >
   ...
```

The assembly above corresponds to the “fi1” block, or the block that the two branches of the if statement merge into. The expression `e = a + b` is included in this block. The bolded lines show where the temp variable `[ RBP - 80 ]` (saved to by the two if branches) was reused to save into the temp variable `[ RBP - 144 ]` (an artifact of our flattening process). `[ RBP - 144 ]` is then saved into `[ RBP - 40 ]`, where e is stored. Notice that there is no `ADD` command, and the extraneous temporary variables before the bolded lines will get cleaned in dead code removal.

Example
For local CSE, we use the example:

```
class Program {
  void main() {
    int a,b,c,d;
    c = a+b;
    d = a+b;
  }
}
```

The follow excerpt of assembly avoids calculating a+b twice:

```
   ...
   ADD        R11, R10                      ; <MidPlusNode>
   MOV        qword [ RBP - 56 ], R11       ; <MidSaveNode: reg10 >
   MOV        qword [ RBP - 64 ], R11       ; <OptSaveNode: reg10 >
   MOV        R11, qword [ RBP - 56 ]       ; <MidLoadNode: reg12,<MidTempDeclNode: t11 > >
   MOV        qword [ RBP - 24 ], R11       ; <MidSaveNode: reg12 >
   MOV        R11, qword [ RBP - 16 ]       ; <MidLoadNode: reg15,<MidLocalVarDeclNode: b > >
   MOV        qword [ RBP - 72 ], R11       ; <MidSaveNode: reg15 >
   MOV        R11, qword [ RBP - 8 ]        ; <MidLoadNode: reg13,<MidLocalVarDeclNode: a > >
   MOV        qword [ RBP - 80 ], R11       ; <MidSaveNode: reg13 >
   MOV        R11, qword [ RBP - 64 ]       ; <MidLoadNode: reg23,<MidTempDeclNode: t22 > >
   MOV        qword [ RBP - 88 ], R11       ; [OPT] <MidSaveNode: reg23 >
   MOV        R11, qword [ RBP - 88 ]       ; <MidLoadNode: reg21,<MidTempDeclNode: t20 > >
   MOV        qword [ RBP - 32 ], R11       ; <MidSaveNode: reg21 >
   ...
```

The first group of bolded lines is saving the results of an add operation (with results in R10 and R11). Note that the add result gets saved once into c ([RBP - 56]) and then into a temp ([RBP - 64]). In the second bolded group, [RBP - 64] is loaded into memory and saved into a temp variable instead of performing the operation again. (This temp is then saved into d at [RBP - 32]; the two stages is once again another artifact from our flattening process and can be cleaned in later optimizations.)

Algebraic Simplification

We chose to implement algebraic simplification in order to most effectively take advantage of GCSE. However, the algebraic simplification is a implemented as a visitor that is run after semantic checking but before low-level IR. It was decided that it should be done at this stage to catch complicated expressions before they became even more complicated at a flattened representation.

The following reductions were implemented:

Arithmetic Reductions

- Literal expressions, e.g. 1*3/2 + 5 → 6
- Literal comparisons, e.g. 1 <= 3 → true
- Additive identity: x + 0, 0 + x → x
- Multiplicative identity: x * 1, 1 * x → x
- Additive identity: expr(x) - expr(x) → 0 (through canonicalization)
- Zero: x*0, 0*x → 0
- Unary minus: --x → x
- Modulo 1: x % 1 → 0
- Divide by 1, x / 1 → x
- 0 Modulo x: 0 % x → 0 (so long as x != 0)
- 0 divide by x: 0/x → 0 (so long as x != 0)

Boolean Reductions

- Literal expressions, e.g. !(false || (true && false)) → true
- true && g, g && true → g
- false && g, g && false → false
- g == true, true == g, g != false, false !=g → g
- g != true, true != g, g == false, false ==g → !g
- !!g → g

Canonicalization
As expressions are simplified, they are canonicalized, allowing expressions to be symbolically compared. It can even identify complex and highly nested expressions that are equivalent. This is made possible by our canonicalization process. Internally, a canonicalization is either "1", a product of variables, a product/division/remainder of two canonicalizations, or a linear combination of canonicalizations; by properly defining the arithmetic operations on canonicalizations, we can generate consistent representations.

As of now, this is only used to simplify equality operations (e.g. expr(a) == expr(a) → true), and expressions that simplify to 0, e.g. a+(b+c)-(a+b)-c*1→ 0. In further iterations, canonicalization can be applied to boolean operations such as && and ||. Also, it can be applied to CSE, e.g. realizing that a+(b+c) has the same value as (a+b)+c.

Careful Elimination
Note that any expression involving function calls will never get eliminated because of potential side effects. For example, f()*0 does not become 0, because f() might print something or change a field variable.

However, we do remove potential division by zero. For example, 0/b, a/b*0 →0, with a check to make sure b != 0. If at compile time, we realize that b = 0 through algebraic simplification, we have a compile-time divide-by-zero error. We also handle array index out of bounds at compile-time if we can compute the array index in a similar manner.

Implementation Issues
Global versus Local CSE
We encountered difficulties in designing a framework so that global and local optimizations did not conflict. For example, local optimizations would add new temporary values in the process of optimizing a block. Some blocks would be processed more than once, and in those cases global CSE would attempt to transform those new temporary values, destroying any promises the temporary values had with respect to storing values for local CSE.

Working with the Past
We discovered various areas where our existing design made it less than ideal for implementing optimizations. Some earlier design decisions were impossible to undo - we made the decision to jump from an ANTLR representation to a mid-level representation complete with register writes and saves. An expression like `a = b + c` is transformed to `t1 = a, t2 = b, t3 = t1 + t2, c = t3`. In the case of CSE, it is much preferable to be able to work with the former rather than the latter.

However, working backwards to create a “high-level” IR before the mid-level would have required first writing code to transform ANTLR code to the high-level IR and then code to

transform high-level IR into mid-level IR. We decided to work with the existing implementation. For the most part, the same concepts applied - we just had unnecessary assignments to temporary variables before moving them to field variables. (These assignments can later be cleaned through copy propagation.)

Known Problems

We pass the provided tests and know of no problems with our code. However, it is very apparent that only global CSE has been implemented. Debugging consumed a large amount of our time, and we ultimately did not have the time to implement other optimizations before the deadline. We intend to implement constant propagation and dead code elimination at the very least for the next project.