Dataflow Analysis

Division of work

As with previous assignments, we used code review to ensure that at least two members of the team understand each component. Liz wrote the local common subexpression visitor. Shaunak wrote the dead code elimination, with debugging assistance from Liz. David wrote the global common subexpression removal code. Maria and David wrote the lattice and genkill frameworks. Maria started an expression canonicalization. Liz performed a number of large refactors and ensured all tests passed following refactoring.

Clarification/addition/assumptions

We have changed the behavior of assignments to evaluate any expressions found in the left hand side first before evaluating the value in the right hand side and storing the RHS in the LHS. Previously, we had assumed the reverse was the case. This is not specified in the spec one way or another.

Overview of design, analysis of alternatives, and design decisions

We performed a number of dataflow optimizations, following the same basic pattern. First, we perform local transformations on each basic block, such as local CSE and local CP following the algorithm given in lecture.

Specifically, the local CSE walks the statements in each block in the forward direction, checking whether a statement's computation has already been cached, and substituting it for a MOV to retrieve the cached value if so. If a statement's computation is not cached, it adds an instruction to cache the value in a temporary and then populates the list mapping computed values to cached variables. A write to a variable causes expressions containing that value to be removed from the mappings. If a write to an array is detected, any computations using any values from the array are flushed from the mappings since two differently-appearing array indexes might overwrite the same index. If a local function is called, global values are flushed from the mappings since local functions can clobber global variables.

The local CP walks the statements in each basic block in the forward direction. As it encounters variables, it adds them to its mappings. If a known variable is assigned to another variable, an entry is created stating that the assignee is an alias for the previously seen variable. If an alias is encountered, it is replaced with its canonical previously seen variable. This allows dead code elimination to work more effectively since it allows for substitution of and removal of temporary variables or copies of variables whose reference variables are not changed before the temporary variable would be read.

We start by running CP to catch cases of the form a = foo(); b = a; c = a + 5; d = b + 5; and replacing them e.g. a = foo(); b = a; c = a + 5; d = a + 5;, allowing CSE to more efficiently detect identical computations. We then run local CSE, followed by a final round of CP to eliminate temporary variables where possible. After local CSE and CP have run, we use a workflow algorithm to compute the information we need for the global transformation, such as the available expression information. Finally, we pass this information to a class which traverses the control flow graph and performs the global transformation.

Our workflow algorithm is the generic fixed point algorithm from class. It takes the following parameters: a lattice, a set of items to traverse, and a starting item, and a starting input. The lattice is parametrized by a concrete type of items - usually the BasicBlockNodes which form the control flow graph - and an abstraction which contains the information the concrete item - such as a BitSet containing the expressions available at each node. The lattice also provides a least upper bound method for joining two abstractions, as well as methods to return bottom and top items. The items given to the workflow algorithm are WorkFlowItems, which are a wrapper around the abstractions that provides a transfer function which maps the information coming into a given node to output information. A subclass of these items, the GenKillItems, computes the transfer function based on GEN and KILL functions.

The global CSE is done by taking every OpStatement which is not merely a copy, enter, or return and treating it as an expression based on its arguments. A workflow algorithm to compute available expressions for every block. This information is stored in an AvailableExpressions class, which is also the lattice used to compute it. We run a second pass of copy propagation following the running of the global CSE.

Combining the lattice and the information storage is natural; because each type of lattice is closely tied to the type of information that is stored, having two separate classes would not add functionality or reduce the complexity of the code. Separating the dataflow information from the transformation is also useful, because it allows one kind of dataflow information to be used in multiple transformations.

The local and global dead code elimination are both done after the sets of variables which are live at each block are computed. The Liveness lattice computes this information by treating every use of a variable in an OpStatement, method call, or system call as a use and by running the worklist algorithm backwards. After the visitor finds the liveness information for every block, the WorklistItems for each block compute the list of dead statements using the local dead code elimination algorithm. Finally, the DeadCodeElimination visitor removes dead code.

Implementation issues

The main issue that we ran into while implementing global CSE was representing expressions in a useful way. Our first approach was using the OpStatement class directly, but it was too inflexible. We eventually found that using a wrapper class around the OpStatements was an effective way of representing and comparing different expressions.

When implementing dead code elimination, we had a number of bugs which were due to incorrectly computing the list of variables defined and used by a statement. For example, we did not count the index of an array as a use at first, and we did not note that a method call can use any global variable. We were able to debug these problems by running a diff of the assembly produced with and without this optimization.

When implementing CSE/CP, we realized that we were always allocating a temporary for each subcomputation, even if a temporary would immediately be used to assign to a variable. This effectively duplicated temporary variables by writing t1 = x + y; z = t1; t2 = z; for the original expression z = x + y. We optimized to always assign the result of the final computation in an assignment directly to the permanent storage variable, removing the spurious t1. This was very buggy and took a long time to pass all regression tests due to reversing the execution order of destination and value from our original design.

Commented test cases

cse-01

Replace second (a + b) computation with load from temporary variable:

```
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{c / -24(%rbp)}
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{d / -32(%rbp)}
+    OpStatement(MOVE, INT{c / -24(%rbp)}): INT{136lcltmp / -136(%rbp)}
+    OpStatement(MOVE, INT{136lcltmp / -136(%rbp)}): INT{d / -32(%rbp)}
```

cse-02

Replace second (a + b) computation with load from temporary variable

```
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{48lcltmp / -48(%rbp)}
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{56lcltmp / -56(%rbp)}
+    OpStatement(MOVE, INT{48lcltmp / -48(%rbp)}): INT{56lcltmp / -56(%rbp)}
     OpStatement(MULTIPLY, INT{48lcltmp / -48(%rbp)}, INT{56lcltmp / -56(%rbp)}): INT{c / -
24(%rbp)}
     CallStatement(printf, .str38667, INT{c / -24(%rbp)}): INT{72lcltmp / -72(%rbp)}
```

cse-03

Replace all (a + b) computations after first with load from temporary variable

```
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{56lcltmp / -56(%rbp)}
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{64lcltmp / -64(%rbp)}
+    OpStatement(MOVE, INT{56lcltmp / -56(%rbp)}): INT{64lcltmp / -64(%rbp)}
     OpStatement(MULTIPLY, INT{56lcltmp / -56(%rbp)}, INT{64lcltmp / -64(%rbp)}): INT{c / -
24(%rbp)}
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{80lcltmp / -80(%rbp)}
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{88lcltmp / -88(%rbp)}
+    OpStatement(MOVE, INT{56lcltmp / -56(%rbp)}): INT{80lcltmp / -80(%rbp)}
+    OpStatement(MOVE, INT{56lcltmp / -56(%rbp)}): INT{88lcltmp / -88(%rbp)}
```

cse-04

Replace all (a + b) computations after first with load from temporary variable (say, called t)
We replace the second (a + b) * (a + b) computation with the temporary variable stored after the computation of c.

```
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{56lcltmp / -56(%rbp)}
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{64lcltmp / -64(%rbp)}
+    OpStatement(MOVE, INT{56lcltmp / -56(%rbp)}): INT{64lcltmp / -64(%rbp)}
     OpStatement(MULTIPLY, INT{56lcltmp / -56(%rbp)}, INT{64lcltmp / -64(%rbp)}): INT{c / -
24(%rbp)}
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{80lcltmp / -80(%rbp)}
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{88lcltmp / -88(%rbp)}
-    OpStatement(MULTIPLY, INT{80lcltmp / -80(%rbp)}, INT{88lcltmp / -88(%rbp)}): INT{d / -
32(%rbp)}
+    OpStatement(MOVE, INT{c / -24(%rbp)}): INT{168lcltmp / -168(%rbp)}
+    OpStatement(MOVE, INT{168lcltmp / -168(%rbp)}): INT{d / -32(%rbp)}
```

cse-05

No CSE can be performed here, due to intermediate assignment to a.

cse-06

No CSE can be performed here, since first (a + b) calculation comes on only one branch.

cse-07

Replace all (a + b) computations after first with load from temporary variable

```
 main:
     [...]
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{c / -24(%rbp)}
+    OpStatement(MOVE, INT{c / -24(%rbp)}): INT{224lcltmp / -224(%rbp)}
     OpStatement(MOVE, BOOLEAN{x / -48(%rbp)}): %r11
   next: .block2, branch: .block0
 .block2:
     NOPStatement
   next: .block1
 .block1:
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{e / -40(%rbp)}
+    OpStatement(MOVE, INT{224lcltmp / -224(%rbp)}): INT{e / -40(%rbp)}
     CallStatement(printf, .str38667, INT{c / -24(%rbp)}): INT{104lcltmp / -104(%rbp)}
     CallStatement(printf, .str38667, INT{d / -32(%rbp)}): INT{112lcltmp / -112(%rbp)}
     CallStatement(printf, .str38667, INT{e / -40(%rbp)}): INT{120lcltmp / -120(%rbp)}
   [END]
 .block0:
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{d / -32(%rbp)}
+    OpStatement(MOVE, INT{224lcltmp / -224(%rbp)}): INT{d / -32(%rbp)}
```

cse-08

Do not replace (a + b) on branch with temp, due to intermediate assignment to a.
Replace last (a + b) computation with load from temporary variable, because expression is calculated on all paths.

```
 main:
     [...]
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{c / -24(%rbp)}
+    OpStatement(MOVE, INT{c / -24(%rbp)}): INT{232lcltmp / -232(%rbp)}
     OpStatement(MOVE, BOOLEAN{x / -48(%rbp)}): %r11
   next: .block2, branch: .block0
 .block2:
     NOPStatement
   next: .block1
 .block1:
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{e / -40(%rbp)}
+    OpStatement(MOVE, INT{232lcltmp / -232(%rbp)}): INT{e / -40(%rbp)}
     CallStatement(printf, .str38667, INT{c / -24(%rbp)}): INT{104lcltmp / -104(%rbp)}
     CallStatement(printf, .str38667, INT{d / -32(%rbp)}): INT{112lcltmp / -112(%rbp)}
     CallStatement(printf, .str38667, INT{e / -40(%rbp)}): INT{120lcltmp / -120(%rbp)}
 [...]
 .block0:
     OpStatement(MOVE, $1): INT{a / -8(%rbp)}
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{d / -32(%rbp)}
+    OpStatement(MOVE, INT{d / -32(%rbp)}): INT{232lcltmp / -232(%rbp)}
```

cse-09

Replace (a + b) on branch with load from temporary variable.
Do not replace (a + b) at end, due to assignment to a on branch.

```
 main:
     [...]
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{c / -24(%rbp)}
+    OpStatement(MOVE, INT{c / -24(%rbp)}): INT{232lcltmp / -232(%rbp)}
     OpStatement(MOVE, BOOLEAN{x / -48(%rbp)}): %r11
   next: .block2, branch: .block0
 [...]
 .block0:
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{d / -32(%rbp)}
+    OpStatement(MOVE, INT{232lcltmp / -232(%rbp)}): INT{d / -32(%rbp)}
     OpStatement(MOVE, $1): INT{a / -8(%rbp)}
```

cse-10

Replace (a + b) at end with load from temporary variable, because expression is calculated on all paths.

```
 .block2:
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{d / -32(%rbp)}
+    OpStatement(MOVE, INT{d / -32(%rbp)}): INT{208lcltmp / -208(%rbp)}
     NOPStatement
   next: .block1
 .block1:
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{e / -40(%rbp)}
+    OpStatement(MOVE, INT{208lcltmp / -208(%rbp)}): INT{e / -40(%rbp)}
     CallStatement(printf, .str38667, INT{c / -24(%rbp)}): INT{96lcltmp / -96(%rbp)}
     CallStatement(printf, .str38667, INT{d / -32(%rbp)}): INT{104lcltmp / -104(%rbp)}
     CallStatement(printf, .str38667, INT{e / -40(%rbp)}): INT{112lcltmp / -112(%rbp)}
   [END]
 .block0:
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{c / -24(%rbp)}
+    OpStatement(MOVE, INT{c / -24(%rbp)}): INT{208lcltmp / -208(%rbp)}
```

cse-11

Replace (a + b) at end with load from temporary variable, because expression is calculated on all paths. The assignment to b does not change the validity of this transform, because we use the same temp on all paths, though the value may be different.

```
 .block2:
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{d / -32(%rbp)}
+    OpStatement(MOVE, INT{d / -32(%rbp)}): INT{216lcltmp / -216(%rbp)}
     NOPStatement
   next: .block1
 .block1:
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{e / -40(%rbp)}
+    OpStatement(MOVE, INT{216lcltmp / -216(%rbp)}): INT{e / -40(%rbp)}
     CallStatement(printf, .str38667, INT{c / -24(%rbp)}): INT{96lcltmp / -96(%rbp)}
     CallStatement(printf, .str38667, INT{d / -32(%rbp)}): INT{104lcltmp / -104(%rbp)}
     CallStatement(printf, .str38667, INT{e / -40(%rbp)}): INT{112lcltmp / -112(%rbp)}
 [...]
 .block0:
     OpStatement(MOVE, $1): INT{b / -16(%rbp)}
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{c / -24(%rbp)}
+    OpStatement(MOVE, INT{c / -24(%rbp)}): INT{216lcltmp / -216(%rbp)}
```

cse-12

No CSE can be performed here, due to intermediate assignment to b on one of the branches.

cse-13

No CSE can be performed here, because (a + b) is not calculated on all branches.

cse-14

Replace (a + b) in loop body with load from temporary variable generated at top of main.

```
 main:
     [...]
     OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{c / -24(%rbp)}
+    OpStatement(MOVE, INT{c / -24(%rbp)}): INT{224lcltmp / -224(%rbp)}
     OpStatement(MOVE, INT{x / -40(%rbp)}): BOOLEAN{96lcltmp / -96(%rbp)}
     OpStatement(MOVE, $5): INT{i / -56(%rbp)}
     OpStatement(LESS_THAN, INT{i / -56(%rbp)}, BOOLEAN{96lcltmp / -96(%rbp)})
 [...]
 .block0:
-    OpStatement(ADD, INT{a / -8(%rbp)}, INT{b / -16(%rbp)}): INT{d / -32(%rbp)}
+    OpStatement(MOVE, INT{224lcltmp / -224(%rbp)}): INT{d / -32(%rbp)}
     OpStatement(ADD, INT{i / -56(%rbp)}, $1): INT{i / -56(%rbp)}
     OpStatement(LESS_THAN, INT{i / -56(%rbp)}, BOOLEAN{96lcltmp / -96(%rbp)})
   next: .block2, branch: .block0
```

cse-15

No CSE can be performed here, due the the assignment to a at the end of the loop. This assignment means that (a + b) may differ from the previously calculated value, when we are at the beginning of the loop.

cse-16

Replace second (a + b) with load from temporary variable.

```
     OpStatement(ADD, INT{a / a}, INT{b / b}): INT{c / c}
-    OpStatement(ADD, INT{a / a}, INT{b / b}): INT{d / d}
+    OpStatement(MOVE, INT{c / c}): INT{72lcltmp / -72(%rbp)}
+    OpStatement(MOVE, INT{72lcltmp / -72(%rbp)}): INT{d / d}
```

cse-17

No CSE can be performed, because the function call may (and in fact does) modify any global, including a and/or b.

cse-18

Do not perform CSE because our bounds checking is generated as we translate our low-level IR into assembly, but CSE is performed on the low-level IR only. Later, after we have raised the array bounds checking into the low-level IR, this CSE should be performed.

Commented source code

See the README file for an index of what components are located in which packages, and the individual source files for commented code.

Known problems

Please see https://github.com/lizthegrey/6035-compiler/issues for an up-to-date list of known issues.