Phase 5 Design Document

Phase Ordering

With -O all, we perform 5 rounds of optimizations on every program. For each round we performed, constant propagation, common subexpression, copy propagation elimination followed by dead code elimination. Performing constant propagation helps reveal common sub expressions. If found, common subexpression elimination produces copy instructions. The copy instructions are propagated. Constant propagation, common subexpression elimination, and copy propagation all produce dead code which is removed at the last step.

SSA

Our strategy for performing optimizations on our CFG was to first convert to SSA using the standard algorithm (described in our phase 4 document), optimize the code, and then convert out of SSA using the standard algorithm. The following sections discuss the optimizations (including optimizations from previous phases) in some detail.

Copy Propagation

In SSA, copy propagation was straightforward to implement. We iterate through the tree to find all the statements of the form y <- x, where y is not a global variable, and then iterate again to replace all occurrences of y with x. We handled chained copies like by replacing copies with their most fundamental values. For example, given the statements t1 <- t2 and t2 <- t3, we would replace t1 and t2 by t3 everywhere in the cfg.

Performing copy propagation across phi expressions makes the SSA non-conventional as defined in section 2.6 of the SSA book. Converting from conventional SSA to flat variables involves giving all versioned variables that are related via phi expressions a unique name. This transformation does not work in non-conventional SSA, and applications of it give rise to the lost copy problem.

To convert back to conventional SSA, we introduce parallel copy instructions into the CFG. Like phi instructions, parallel copy instructions can't be implemented directly in assembly, so we must cleverly convert them into a set of regular copy instructions. This process potentially introduces many temporaries. Using algorithm 3.6 in the SSA book we minimized the number of these temporaries created.

We did not propagate copies of global variables since we allow them to chance throughout the program while in SSA form.

Example:

```
1   ---------------------------------
2   Original Code
3   ---------------------------------
4   import printf;
5
6   void main() {
7     int a, b, c;
8
9     b = 1;
10    a = b;
11    c = a;
12
13    printf("%d %d %d", a, b, c);
14  }
15
16  ---------------------------------
17
18  method after ssa construction:
19  Method: main
20  -------------------------
21  BasicBlock 0
22    Parents: []
23  | t4_v1 <- $1 |
24  | t2_v1 <- t4_v1 |
25  | t1_v1 <- t2_v1 |
26  | t3_v1 <- t1_v1 |
27  | call printf(t"%d %d %d", t1_v1, t2_v1, t3_v1) |
28  | nowhere |
29  -------------------------
30
31  method after ssa destruction:
32  Method: main
33  -------------------------
34  BasicBlock 0
35    Parents: []
36  | t4 <- $1 |
37  | call printf(t"%d %d %d", t4, t4, t4) |
38  | nowhere |
39  -------------------------
```

In the examples, t4_v1 is copied into t2_v1 and implicitly copied into t1_v1 and t3_v1. After copy propagation the copy instructions are removed and the three copies of t4_v1 are replaced with the same value.

Dead Code Elimination

We performed dead code elimination in two steps. First, we compute the set of all variables that are sources for instructions. Then, we remove all instructions that store results in non-global variables that are never used. Then, we repeat these two steps until the CFG stops changing.

Example

```
Original Code
import printf;

void main() {
  int a, b, c;

  a = 5;
  b = 6;
  c = a + b;

  printf("%d %d", a, b);
}

method after ssa construction:
Method: main
-------------------------
BasicBlock 0
  Parents: []
| t4_v1 <- $5 |
| t1_v1 <- t4_v1 |
| t5_v1 <- $6 |
| t2_v1 <- t5_v1 |
| t6_v1 = t1_v1 + t2_v1 |
| t3_v1 <- t6_v1 |
| call printf(t"%d %d", t1_v1, t2_v1) |
| nowhere |
-------------------------

method after dead code elimination:
Method: main
-------------------------
BasicBlock 0
  Parents: []
| t4_v1 <- $5 |
| t1_v1 <- t4_v1 |
| t5_v1 <- $6 |
| t2_v1 <- t5_v1 |
| t6_v1 = t1_v1 + t2_v1 |
| call printf(t"%d %d", t1_v1, t2_v1) |
| nowhere |
-------------------------
```

The single round of dead code elimination accurately removed the definition of t3_v1 since it is never a source for another instruction.

Common Subexpression Elimination

We performed CSE in SSA form, where the value of subexpressions are never changed. In the program. We computed common subexpressions by assigning every subexpression in the cfg a hash.

```
5 implementations
pub enum CSEHash<T> {
    Add(ImmVar<T>, ImmVar<T>),
    Sub(ImmVar<T>, ImmVar<T>),
    Mul(ImmVar<T>, ImmVar<T>),
    Div(ImmVar<T>, ImmVar<T>),
    Mod(ImmVar<T>, ImmVar<T>),
    Lt(ImmVar<T>, ImmVar<T>),
    Lte(ImmVar<T>, ImmVar<T>),
    Eq(ImmVar<T>, ImmVar<T>),
    Neq(ImmVar<T>, ImmVar<T>),
    Neg(ImmVar<T>),
    Not(ImmVar<T>),
}
```

Before assigning a subexpression a hash, standardized them so that commutative expressions received the same hash. For commutative expressions involving immediates, the immediate was always on the right. For commutative expression involving variables, variables with lower indices were always on the left.

Once found, common subexpressions are assigned a temporary variable that is assigned the value of the subexpression. And the instructions involving the common subexpressions are replaced by move operations. The definition of the subexpression is placed so that they dominate all of the move instructions.

```
method after ssa construction:
Method: main
Fields: [4, 5, 6, 3, 2, 1]
block 0:
------------------------
|  |
| t5_v1 = t1_v0 + t2_v0 |
| t3_v1 <- t5_v1 |
| t6_v1 = t2_v0 + t1_v0 |
| t4_v1 <- t6_v1 |
| call printf(t"%d %d", t3_v1, t4_v1) |
| nowhere |
------------------------
```

```
method after CSE:
Method: main
Fields: [4, 5, 6, 3, 2, 1, 7]
block 0:
-------------------------
|  |
| t7_v1 = t1_v0 + t2_v0 |
| t5_v1 <- t7_v1 |
| t6_v1 <- t7_v1 |
| call printf(t"%d %d", t5_v1, t6_v1) |
| nowhere |
-------------------------
```

We see that after common subexpression elimination, the additions of a and b are replaced by moves.

Constant Propagation

We perform constant propagation in SSA form. If we find a constant load t <- $x, then we replace all uses of t with $x. This potentially turns unary and binary operations into constant loads if all of their sources are immediates.

doc / phase5-code / = cop_example.txt

```
Method: main
Fields: [4, 5, 3, 1, 6, 2]
block 0:
-------------------------
| t1_v1 <- $1 |
| t6_v1 = t1_v1 + t2_v0 |
| t3_v1 <- t6_v1 |
| call printf(t"%d %d", t3_v1, t4_v0) |
| nowhere |
-------------------------
```

```
method after constant propagation:
Method: main
Fields: [4, 5, 3, 1, 6, 2]
block 0:
-------------------------
|  |
| t6_v1 = $1 + t2_v0 |
| t3_v1 <- t6_v1 |
| call printf(t"%d %d", t3_v1, t4_v0) |
| nowhere |
-------------------------
```

We see that the $1 has propagated into the binary operation that defines t6_v1. Here's an example where the constant propagation cascades into further simplifications.

```
Example with Reduction of Binary Operations
After SSA Construction
Method: main
-------------------------
|  |
| t1_v1 <- $1 |
| t2_v1 <- $1|
| t6_v1 = t1_v1 + t2_v1 |
| t3_v1 <- t6_v1 |
| call printf(t"%d %d", t3_v1) |
| nowhere |
-------------------------

method after constant propagation:
Method: main
Fields: [3, 5, 6, 1, 2, 4]
block 0:
-------------------------
|  |
| call printf(t"%d %d", $2) |
| nowhere |
-------------------------
```

Here constant propagation was able to remove all variable definitions in the program.

Register Allocation

We did a non-SSA graph coloring register allocation phase. Immediately after transformation out of SSA form, we analyze the CFG to find def-use “webs”. Then we find the convex hull of each web (i.e., the set of instructions lying on a path from a def in the web to a use in the web), and we construct the interference graph by seeing which webs have intersecting convex hulls. Finally, we color the interference graph, assigning a register to each web, and use these assignments to construct a modified CFG in which the variables are registers (e.g. %rax) rather than integer labels (e.g., var1, var2, …) as it had been prior to register allocation.

Our first implementation of spilling worked as follows. After constructing the interference graph, we would attempt to color it using a simple heuristic: repeatedly remove all nodes with degree smaller than the number of registers; if you manage to remove all nodes, you can easily construct a coloring, and if not, we say that coloring failed, in which case we use some heuristic to pick a node to spill, modify the CFG by inserting spill and reload instructions, and then recompute the interference graph and try coloring again.

We tried a few heuristics to choose good webs to spill. The first thing we tried was spilling the web with the largest degree in the interference graph (after removing the colorable nodes). We also tried spilling the web with the most instructions in its convex closure. We also tried looking at all the program points where there were too many live variables, and for each program point picking the web that had the greatest distance to its next use and then spilling that web. We don’t have a great idea how any of these heuristics compare to each other in efficiency of generated code; we were mainly hoping to find a heuristic that would make the register allocation phase stop taking so long, and none of the ones we tried accomplished that.

We found that our implementation of regalloc/spilling was very slow on large test cases (especially 12-huge.dcf), due to having to recompute the interference graph after spilling each web (we had made little attempt at making the interference graph computation efficient, just focusing on correctness). So we also tried another version of spilling, in which coloring would never fail—instead, if the coloring heuristic found that every node had degree at least equal to the number of registers, it would just pick some node to mark as "spilled", remove it from the interference graph, and keep coloring (instead of quitting and inserting spill/reload instructions in the CFG as before). This made the register allocation phase significantly faster. It also led to generated code being faster, for reasons we don't completely understand. Likely, this was partly due to the fact that explicitly inserting spill and reload definitions into our CFG increased the number of webs and made coloring harder.

We did not have time to implement any sort of coalescing or web splitting.

Extras

No extras.

Peephole optimizations

We implemented peephole optimizations, but did not get to incorporate them into our compiler. After constant propagation, we could check for reducible operations, like addition,subtraction or multiplication by 0 and division or multiplication by a power of 2. We also check for relational operations that could be simplified, including comparing a variable to itself.

Difficulties

We had difficulties with our register allocation phase being slow. This led to our second attempt at implementing it, described above. After this second attempt, we did not observe any slowness issues on our laptops, but our compiler still consistently timed out on durp on the autograder.

We never managed to get the best (i.e., fastest generated code) version of our compiler to avoid timing out on all the autograder tests (just had difficulties with durp). I think the fastest version of our compiler that managed to pass all the derby test cases (i.e., managed to avoid timing out) did not have register allocation implemented.

Contribution

Marshall did common subexpression elimination and constant propagation. He also tried out some peephole optimizations, but we never managed to submit a compiler with these optimizations to the derby (too busy trying to fix regalloc). Owen mostly wrote an implementation of global code motion, but this too never managed to be incorporated into a

version that we submitted to the derby. We worked together to write, debug, and speed up the register allocation phase.

Where is our code?

The master branch of our repository has our latest work, along with docs.