6.035 Dataflow Analysis Project

Team: Premature Optimization

David Stingley, Jason Paller-Rzepka, Manuel Cabral

Introduction

The purpose of this document is to describe our work for Project 4: Dataflow Analysis. The document is organized into four sections. First, we describe the new behavior that we introduced. Second, we explain how to exercise the new functionality. Third, we give an overview of the modifications we made to the codebase. Finally, fourth, we discuss some design decisions we made.

Expected Behavior

We added support for one dataflow-analysis-based optimization: Common Subexpression Elimination. When this CSE optimization is enabled, the compiler causes expressions to be stored in temporary variables. It identifies each statement where that expression is re-computed, and replaces the re-computation with a load from the corresponding temporary variable.

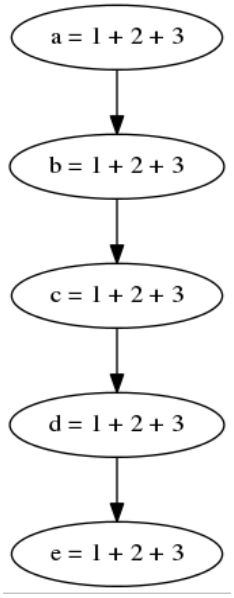
In the rest of this section, we will provide examples of correct operation, and also describe features that we left out in the interest of time.

Examples of Correct Operation

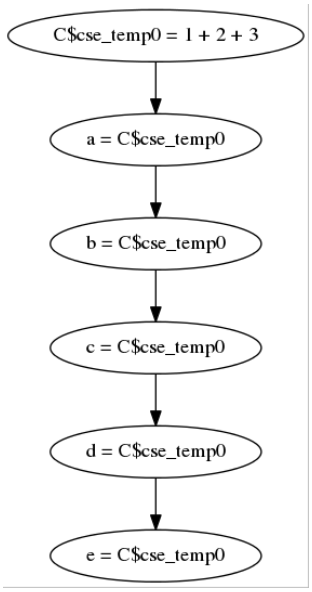
This section contains examples of correct operation of the Common Subexpression eliminator:

Very Basic Use of a Temporary Variable

Before:

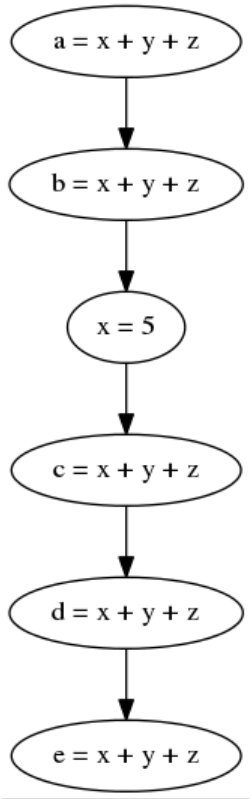


After:

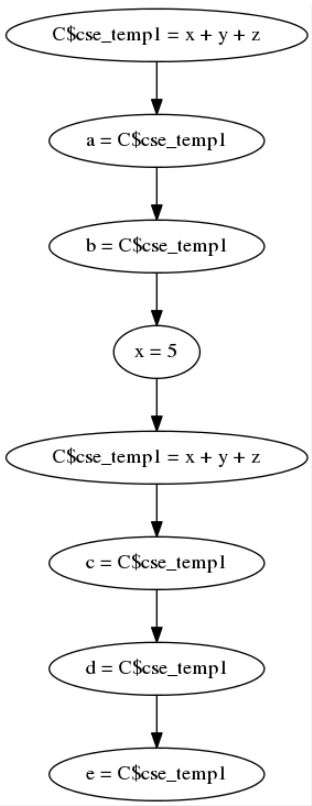


“Re-Filling” a Temporary Variable After Its Expression Becomes Unavailable

Before:

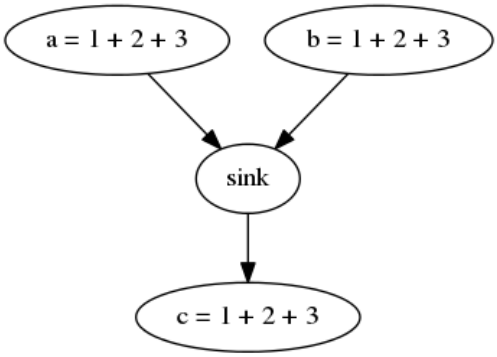


After:

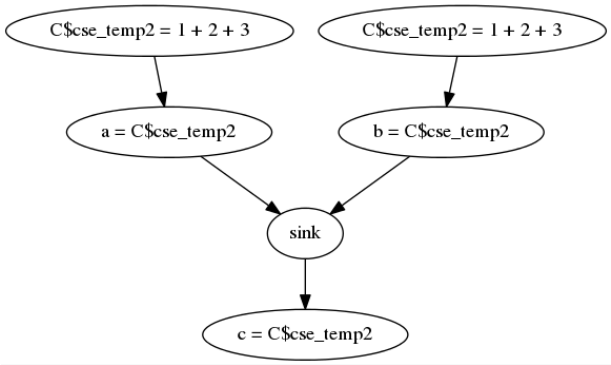


Judging an Expression to be Reachable Only if it is Reachable Along All Paths

Before:



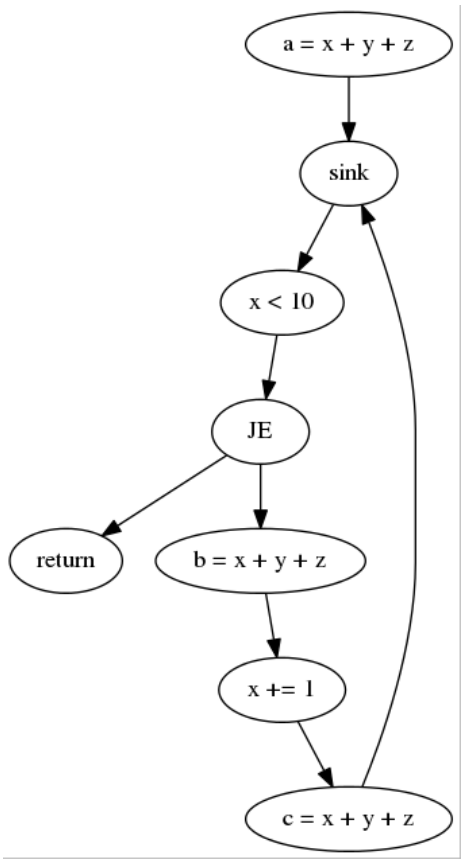
After:



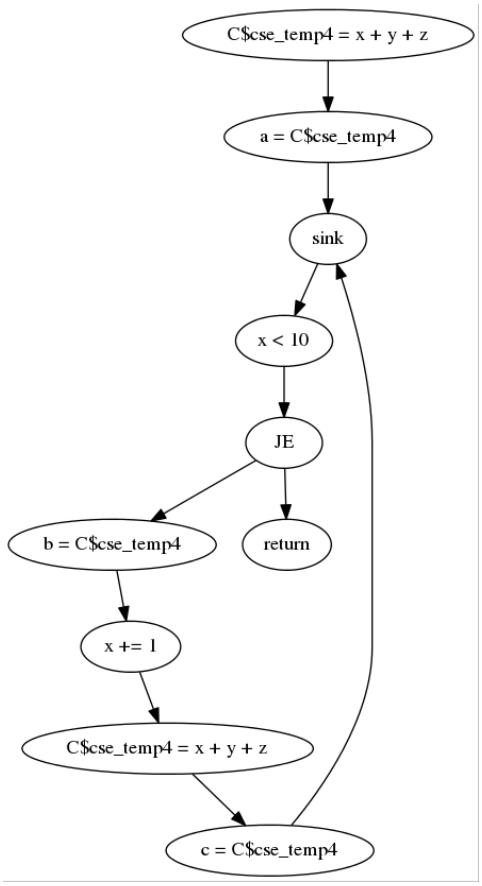
In a Loop: Judging Expressions to be Available When Appropriate

Note: For this data flow representation, when a node has multiple direct successors, the order of those direct successors is meaningless.

Before:



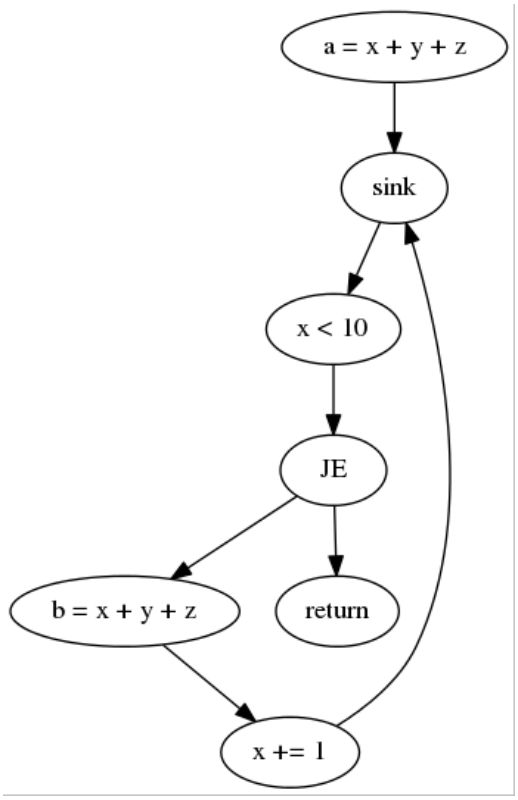
After:



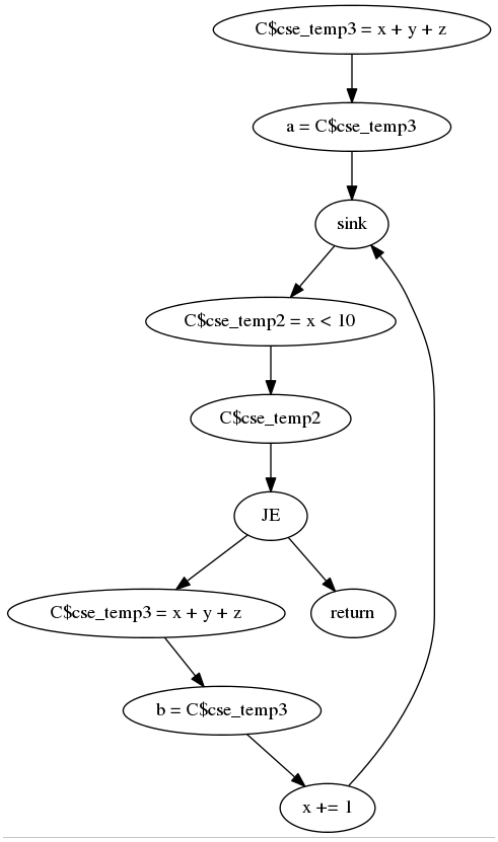
In a Loop: Judging Expressions to be Unavailable When Appropriate

Note: A side effect of the optimizer is to add some unnecessary temp variables. We do not do copy propagation (yet!). So, there are some unnecessary temp variables left in this graph.

Before:



After:



Not-Yet-Implemented Functionality:

There are a number of features we could add to improve our optimizer:

- Track which assignments are actually used as available expressions in the future. Only store expressions in temp variables if they're needed. (This could be addressed by implementing dead-code elimination and copy-propagation, instead.)
- Replace expressions on a more fine grained level. For example, we should be able to replace "`x = 1 + a + b +c; y = 2 + a + b + c;`" with "`temp = a + b + c; x = 1 + temp; y = 2 + temp`".
- Relax our policy of killing all expressions that contain globals whenever we do a method call. For now, we assume that all method calls mutate all global variables!

HOWTO

To run the compiler *without* optimizations, run the following command:

```
$ ./run.sh -t assembly source.dcf
```

To run the compiler *with* Common Subexpression Elimination optimizations, run the following command:

```
$ ./run.sh -t assembly --opt=cse source.dcf
```

If you wish to see the Data Flow Graph (with or without optimizations), run the following command:

```
$ ./run.sh -t DFG [--opt=cse] source.dcf | dot -Tpng > dfg.png
```

(Note that the `dot` utility can be installed with `apt-get install graphviz`.)

Overview of Modifications

New Intermediary Representation

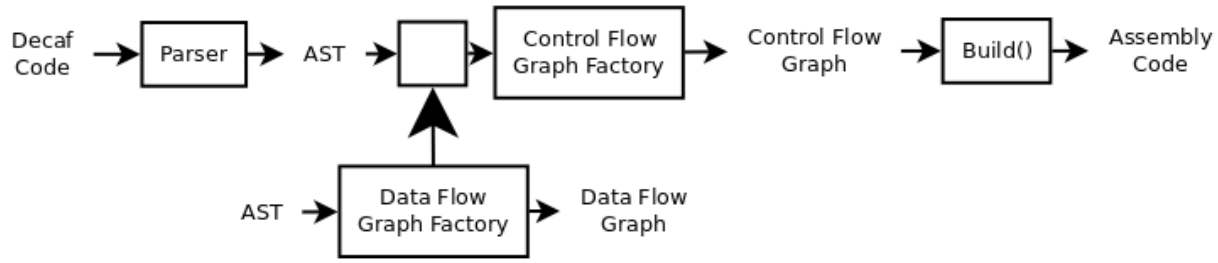


Figure: Visual representation of the new components in the graph.

We agreed that in order to apply optimizations to our compiler in an easy to debug, disable and reorder fashion we first needed to craft another intermediate representation for input code. This is due to the fact that an AST makes it easy to define hierarchical relationships between lines of code, for instance available variables or expressions, but difficult to apply changes due to a lack of structured control flow between lines of code. On the other hand, our instruction level representation of individual machine instructions is good for expressing the control flow in a program's execution, but has at that point discarded most information relating any set of instructions to any other set of instructions.

Thus our first major design decision for implementing data flow analysis and enabling optimizations was to create an intermediate representation between ASTs and pseudo-machine instruction control flow graphs. At this level, we wanted to turn an AST into a linearized representation of the code, but keep information about the expressions and assignments themselves. Therefore we agreed to make **D**ata **F**low **G**raphs, which would be structurally similar to the **C**ontrol **F**low **G**raphs of machine instructions below them but still contain all the information from the AST representation.

Available Expressions Analysis

The compiler now performs Common Subexpression Elimination. The available expressions are calculated according to the algorithm found in lecture. See below for link. There are two minor changes from that algorithm. In order to simplify the calculation of the GEN and KILL sets, the transfer function is **(GEN U IN[n]) - KILL** instead of what was presented in lecture. This handles statements such as "x = x+c better than the original transfer function.

Calculating E, GEN, and KILL

The compiler does a pass through the DFG to calculate the list of all expressions E, what expressions each node generates (GEN), and what expressions each node kills (KILL). We kept it at the top level of what an expression could be for simplicity of implementation. There is also a notion of how complex an expression is. We only save unary, binary, and ternary operations. Here are the rules for their calculation:

E: Any valid expression encountered during the pass through the DFG, unless it contains a method call.

Gen Sets: The top level expression of each node is saved, unless that expression contains a method call.

Kill Sets: Any node that contains a method call kills all expressions that contain global variables. This relies on the assumption that method calls may edit any of the global variables. Also, assignments kill all subexpressions that rely on the variable being assigned.

Link to Available Expressions fixed-point algorithm:
http://6.035.scripts.mit.edu/sp14/slides/F14-lecture-10.pdf

Common Subexpression Elimination

The compiler does an initial pass over the program to generate available expressions for the entire program. We then feed this information into a method that loops over the entire set of blocks reachable from the start of the program's execution. For each block, we check the computed set of available expressions for that block, if an expression in that block is in the set of available expressions at that block we attempt to replace it with a reference to a compiler generated variable. This means that initially all expressions will be replaced with a compiler generated variable assignment as an intermediate step. However, we know that future passes of copy propagation will remove this redundancy, so we employ this strategy to simplify the process as a whole.

When a block contains an expression that is already represented by some earlier existing compiler variable, we instead just replace it with that variable as we do not need to re-define compiler variable, due to the method of their implementation they are ensured to be

available to all expressions that they replace. This is because we currently place all compiler variable at the method level for instantiation. In the future we hope to replace this with intelligent identification of the least upper bound of scope depth necessary to a given compiler variable, up to the global level, which should improve the accuracy of our optimization.

Design Decisions

Mutability vs. Immutability

Initially, it was our goal to make as much of each intermediate representation immutable as possible, mainly to avoid complex debugging of intermediate states. With this in mind both our AST and CFG representations are largely immutable, we maintain this through extensive use of immutable wrapper classes provided by Guava, and explicit builders and factories that recursively construct each representation.

However, when approaching DFGs we acknowledged that the nature of optimizations made mutability a desirable trait. We decided that the trade-off in ease of design and operation, and the reduction in explicit factories and data transfer that having a mutable DFG that we edit with successive optimization passes was was worth the potential for bugs that a mutable structure would bring.

Dropping Bit Vectors

The proposed implementation of most optimization procedures discussed in class involved the use of an explicit bit vector that mapped each unique statement to a corresponding bit and then operated on the resultant sets with binary union and intersection operators (OR and AND). However, Java provides us with the luxury of being able to define explicit sets of elements and perform set operations on them when provided with proper equality implementations. So, we chose to eliminate the bit vector representation altogether in favor of directly representing expressions in a set of expressions, and providing methods of evaluating equality to enable set operations as expected. This reduced the complexity of our implementation by eliminating the need to store explicit mappings between bit vectors and expressions.

This choice to directly maintain sets of expressions also has the benefit of allowing us to directly manipulate those expression sets in future passes for other forms of optimization. For instance, we can apply algebraic simplifications directly to the set of all expressions or determine later techniques for memory arrangement and register use by looking directly at the sets of expressions instead of attempting to consult mappings.

Basic Block Size

Since proper transfer functions are composable, we decided to make the basic units of our DFG single statements of Decaf code. This is correct as larger blocks would simply be compositions of the singleton as permitted. However, by reducing the size of basic blocks to single element we are able to greatly simplify how to handle evaluating GEN and KILL for a block as it only needs to interact with a single expression. It also has the benefit of making

error analysis for the DFG optimization passes easier to digest and interpret, as the process of running an optimization can be stepped through on a statement-to-statement level for the entirety of a Decaf program to identify offending code segments and potential problems.

Available Expressions, or how we learned that Premature Optimization didn't refer to our code, rather it referred to us.

Notes:
how to use our code
overview of mods
overview of design decisions

DataFlowNode
Availability Calculator
Optimizer

blocks of size one
transfer function now (gen u in) - kill
bit vector v sets of subexpressions

Things we missed:
-benefactor, beneficiary
-recursing on method calls for subexpressions
-knowing which globals each method edits

The following is an example of assembly code generated without and with the optimization for the following program:

```
void main() {
  int x, y, z;
  int a, b;

  x = 1;
  y = 2;
  z = 3;

  a = x + y + z;

  while (x < 10) {
    // The expression is NOT available, because it's not available in the
    // branch that loops back to the top of the loop.
    b = x + y + z;
    x += 1;
  }
}
```

Without Optimizations:

```
// All methods.
    .globl main
main:
enter $40, $0
movq %rbp, .e_COMPILER$mainbaseptr
push $0
pop %r11
movq %r11, -8(%rbp)
push $0
pop %r11
movq %r11, -16(%rbp)
push $0
pop %r11
movq %r11, -24(%rbp)
push $0
pop %r11
movq %r11, -32(%rbp)
push $0
pop %r11
movq %r11, -40(%rbp)
push $1
pop %r11
movq %r11, -8(%rbp)
push $2
pop %r11
movq %r11, -16(%rbp)
push $3
pop %r11
movq %r11, -24(%rbp)
movq -8(%rbp), %r11
push %r11
movq -16(%rbp), %r11
push %r11
pop %r10
pop %r11
addq %r10, %r11
push %r11
movq -24(%rbp), %r11
push %r11
pop %r10
pop %r11
addq %r10, %r11
push %r11
pop %r11
movq %r11, -32(%rbp)
```

With Common Subexpression Elimination:

```
// All methods.
    .globl main
main:
enter $56, $0
movq %rbp, .e_COMPILER$mainbaseptr
push $0
pop %r11
movq %r11, -8(%rbp)
push $0
pop %r11
movq %r11, -16(%rbp)
push $0
pop %r11
movq %r11, -24(%rbp)
push $0
pop %r11
movq %r11, -32(%rbp)
push $0
pop %r11
movq %r11, -40(%rbp)
push $1
pop %r11
movq %r11, -8(%rbp)
push $2
pop %r11
movq %r11, -16(%rbp)
push $3
pop %r11
movq %r11, -24(%rbp)
movq -8(%rbp), %r11
push %r11
movq -16(%rbp), %r11
push %r11
pop %r10
pop %r11
addq %r10, %r11
push %r11
movq -24(%rbp), %r11
push %r11
pop %r10
pop %r11
addq %r10, %r11
push %r11
pop %r11
movq %r11, -56(%rbp)
```

```
.cf_93_multi_source:
movq -8(%rbp), %r11
push %r11
push $10
pop %r11
pop %r10
cmp %r11, %r10
movq $0, %r11
movq $1, %r10
cmovl %r10, %r11
movq %r11, %r11
push %r11
pop %r10
cmp $1, %r10
jne .cf_113_false
movq -8(%rbp), %r11
push %r11
movq -16(%rbp), %r11
push %r11
pop %r10
pop %r11
addq %r10, %r11
push %r11
movq -24(%rbp), %r11
push %r11
pop %r10
pop %r11
addq %r10, %r11
push %r11
pop %r11
movq %r11, -40(%rbp)
movq -8(%rbp), %r11
push %r11
push $1
pop %r10
pop %r11
addq %r10, %r11
push %r11
pop %r11
movq %r11, -8(%rbp)
jmp .cf_93_multi_source
.cf_113_false:
leave
ret


// All strings.
```

```
movq -56(%rbp), %r11
push %r11
pop %r11
movq %r11, -32(%rbp)
.cf_101_multi_source:
movq -8(%rbp), %r11
push %r11
push $10
pop %r11
pop %r10
cmp %r11, %r10
movq $0, %r11
movq $1, %r10
cmovl %r10, %r11
movq %r11, %r11
push %r11
pop %r11
movq %r11, -48(%rbp)
movq -48(%rbp), %r11
push %r11
pop %r10
cmp $1, %r10
jne .cf_129_false
movq -8(%rbp), %r11
push %r11
movq -16(%rbp), %r11
push %r11
pop %r10
pop %r11
addq %r10, %r11
push %r11
movq -24(%rbp), %r11
push %r11
pop %r10
pop %r11
addq %r10, %r11
push %r11
pop %r11
movq %r11, -56(%rbp)
movq -56(%rbp), %r11
push %r11
pop %r11
movq %r11, -40(%rbp)
movq -8(%rbp), %r11
push %r11
push $1
pop %r10
```

```
.data

// All globals.

.bss
    .e_COMPILER$mainbaseptr: .quad 1
```

```
pop %r11
addq %r10, %r11
push %r11
pop %r11
movq %r11, -8(%rbp)
jmp .cf_101_multi_source
.cf_129_false:
leave
ret


// All strings.

.data

// All globals.

.bss
    .e_COMPILER$mainbaseptr: .quad 1
```