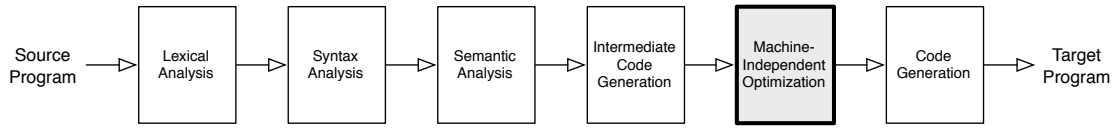
CS 301 Spring 2019
Tutorial Assignment
9 April

Optimization, Data-Flow Analysis

1 Plan



While you are implementing intermediate code generation and x86 machine code generation in your Roost compilers, we will begin exploring machine-independent optimization of code. Automatically improving the efficiency of code is a tall order, especially considering it must not break the code (change *what* it computes) along the way! We will spend 2+ weeks on a survey of optimization topics. We start with small, ad hoc optimizations. Next week, we will show how a fairly general theoretical model can support clean and principled implementations of many sophisticated analyses and optimizations necessary to generate efficient code.

The readings and exercises are in parts: control-flow graphs and an introduction to optimization; local optimizations; and data-flow analysis. This set of reading and exercises is a little larger than usual, since we did not have meetings this past week. We will preview some parts during class on Friday ahead of tutorial meetings.

- The first set of readings/exercises gives an overview of control-flow graphs and optimization in general.
- The second set of readings and exercises explores simple, *local* optimizations for eliminating redundant expressions with a technique called *value numbering*. This optimization is *local* in that it considers only single *basic blocks* – small stretches of code that always execute together in all executions of the program.
- The third set of readings and exercises introduces the idea of *data-flow analysis* to support a few specific optimizations that reason about the flow of data through more complicated control flow structures like conditionals and loops. Data-flow analyses in general compute *facts* or *invariants* that must be true at the beginning and end of each basic block in a Control Flow Graph for a single method/function/procedure body.

Next week we will continue with more depth on data-flow analysis and other techniques.

2 Readings

Control-flow graphs, basic blocks, and optimization:

- Dragon 8.4
  Alternative: EC 5.2, 5.3.2
- EC 8.1 - 8.3 (Skip or skim 8.2.1)

Local optimizations:

- EC 8.4 (Skip or skim 8.4.2)
- Dragon 8.5

Intro to data-flow analysis:

- Dragon 9 – 9.2

3 Exercises

1. Consider the following TAC code:

```
 x = 2
 y = 3
 z = 11
L0:
 T0 = x < 10
 fjump T0 L1
 T1 = x < y
 fjump T1 L3
 T2 = x + 1
 x = T2
 jump L2
L3:
 T3 = y < 100
 fjump T3 L2
 T4 = y + 1
 y = T4
 jump L3
L2:
 T5 = z + 3
 z = T5
 jump L0
L1:
```

   (a) Build the basic blocks and control flow graph for this code.

   (b) Identify the loops in the CFG. (We will see some formal definitions for a couple loop notions later. For now, use intuition.)

2. Consider the following two basic blocks:

```
a = b + c        a = b + c
d = c            e = c + c
e = c + d        f = a + c
f = a + d        g = b + e
g = b + e        h = b + c
h = b + d
```

   (a) Build a DAG for each block to show the dependences between the operations it performs. (Dragon and EC use different DAG notation. The Dragon book form is more flexible.)

   (b) Perform *local value numbering* separately on each of the two basic blocks.

   (c) Explain any differences in the redundancies found by these two techniques.

   (d) At the end of each block, `f` and `g` have the same value. Why do the algorithms have difficulty discovering this fact?

3. (Dragon 8.5.6) Consider this basic block of intermediate code that uses C-style pointers. Recall that the expression `*p` uses the value of `p` as an address, referring to the contents of the memory location given by that address (not to the contents of variable `p` itself).

```
a[i] = b
*p = c
d = a[j]
e = *p
*p = a[i]
```

(a) Assume `p`'s value is unrestricted. In other words, `p` may hold the address of (*i.e.*, point to) any location in memory. Construct the DAG for the basic block.

(b) Assume `p`'s value is restricted to hold the address of (i.e., point to) only the storage for `b` or `d`. Construct the DAG for the basic block.

4. For the Control Flow Graph (CFG) in Dragon Figure 9.10 (attached):

   (a) Identify the loops.

   (b) Statements (1) and (2) in B_1 are both copy statements, in which a and b are given constant values. For which uses of a and b can we perform copy propagation and replace these uses of variables by uses of a constant. Do so, wherever possible, and show the resulting CFG. Do any statements become Dead Code?

   (c) Identify any global common subexpressions for each loop, and eliminate them wherever possible. Show the resulting CFG.

   (d) Using the CFG from (b), Identify any induction variables for each loop. Be sure to take into account any constants introduced in (b). Can strength reduction and/or induction variable elimination be applied? If so, show the resulting CFG. If not, describe one or two instructions that, if added the flow graph, would result in an opportunity for strength reduction.

   (e) Does the CFG from part (c) contain any loop-invariant computations to which code motion can be applied? If not, describe one or two instructions that, if added to the flow graph, would result in an opportunity for code motion.

5. For the CFG in Dragon Figure 9.10 (attached), compute the following:

   (a) Reaching Definitions:

   - The *gen* and *kill* sets for each block. I usually represent this information in a table like the one started below:

Block	*gen*	*kill*
B_1	(1), (2)	(8), (10), (11)
B_2		
B_3		
B_4		
B_5		
B_6		

   - The IN and OUT sets for each block. Write the IN and OUT sets on the CFG (attached) while working through the algorithm. Print a few copies to use for the other parts of this problem.

   (b) Available Expressions:

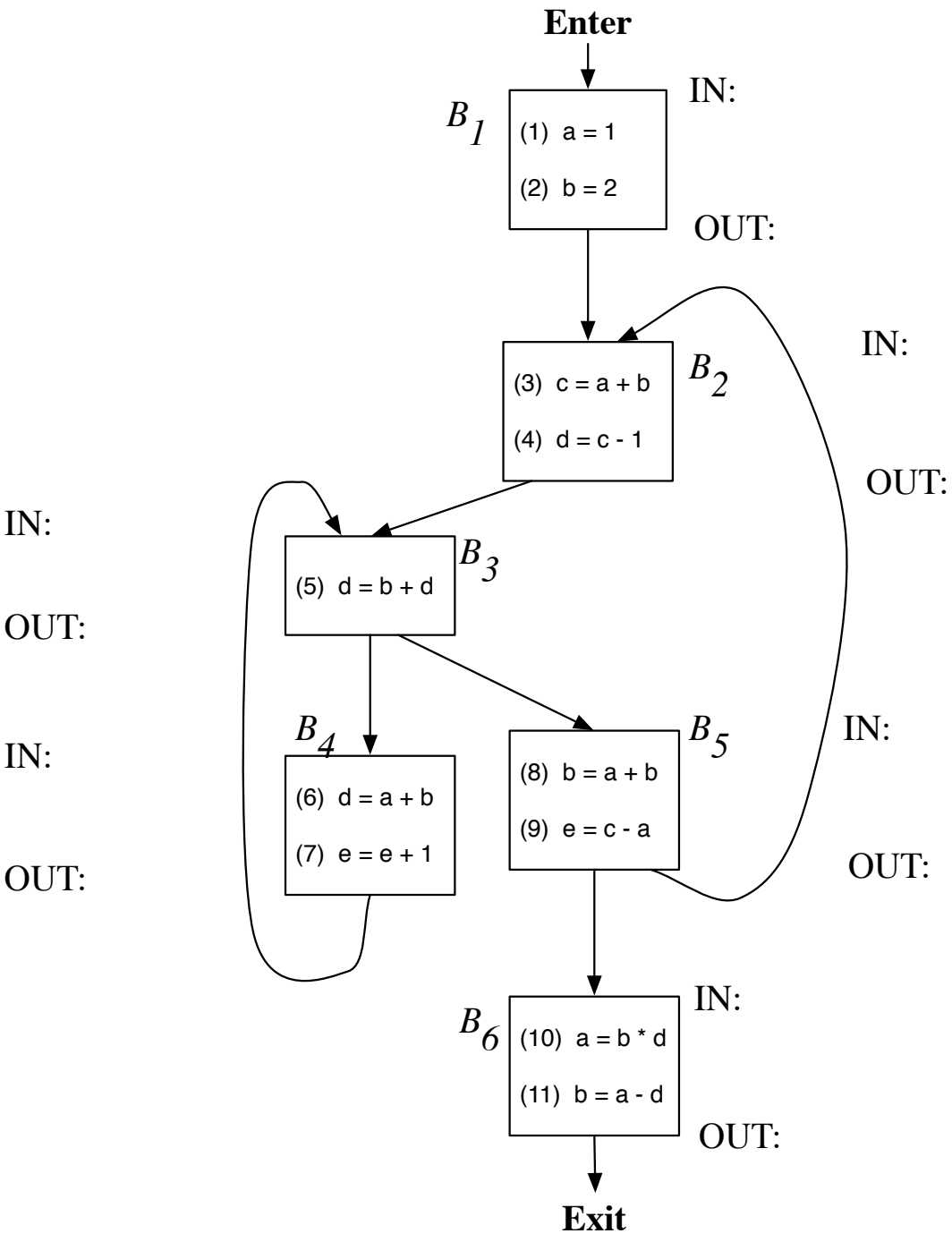
   - The *e_gen* and *e_kill* sets for each block. For the *e_kill* sets, you may use a description like "all expressions using a or b as an operand."
   - The IN and OUT sets for each block.

   (c) Live Variables:

   - The *def* and *use* sets for each block.
   - The IN and OUT sets for each block.

6. (Dragon 9.2.6) Prove by induction that the IN and OUT sets for a block never shrink while computing Reaching Definitions with Dragon Algorithm 9.11 (page 607). In other words, show that, once a definition has been added to one of these sets on an iteration, all future iterations also include it in that set. Use induction over the number of iterations of the innermost loop. Consider what Reaching Definitions means and explain intuitively why we would want the proven property to hold. A succinct, careful explanation here can supplant a formal proof if needed.

7. Consider Dragon 9.2.7 or 9.2.8 and try to develop intuition for how to relate data-flow facts back to program behavior. What do they *really mean?* You do not need to build full solutions.


Enter
B_1
(1) a = 1
(2) b = 2
IN:
OUT:
B_2
(3) c = a + b
(4) d = c - 1
IN:
OUT:
B_3
(5) d = b + d
IN:
OUT:
B_4
(6) d = a + b
(7) e = e + 1
IN:
OUT:
B_5
(8) b = a + b
(9) e = c - a
IN:
OUT:
B_6
(10) a = b * d
(11) b = a - d
IN:
OUT:
Exit