I. Local Optimizations
   A. What they are
      1. Code modifications done solely within a block
         a. No use of data flow information
         b. Just use the properties of a block
      2. Constant propagation and folding
      3. Algebraic transformations
      4. Redundant expression elimination
         a. This involves a concept called value numbering
      5. Dead code elimination
   B. Note that most of these are done again later
      1. Done globally using flow information
      2. Done after other optimizations which might create code that needs to
         be further cleaned up
   C. Note that most of these can be done earlier
      1. Can be done when building the AST
         a. Constant propagation is necessary in some languages (C)
            i. Contexts that only allow a constant can take const expr
      2. Can be done as you are generating code for a block
         a. We already do some in reusing temporaries with same op/operands
         b. We also saw how you could do constant propagation at this time
      3. If you aren’t doing optimization this is often the case
         a. It just involves a fancier code generator
         b. Example: tracking constants
         c. Example: tracking known temporaries in the block
      4. However, if you are going to do optimization
         a. They will be needed later on as well
         b. Easier to code a routine that can be called repeatedly during
            optimization
   D. We need some basic tools before we proceed
1. Constant propagation
2. Value numbering

II. Constant Propagation

A. Task
   1. Replace constant arithmetic with constants
   2. Temporaries that hold constants can be replaced with that constant effectively

B. Method
   1. Maintain a value for each temporary
      a. Actually a Map from temporary to value
      b. Value consists of three parts
         i. The actual constant value
         ii. The type of the constant
         iii. A flag indicating TOP, BOTTOM, or CONSTANT
      c. TOP: the temporary is undefined (initial value)
      d. BOTTOM: the temporary is defined, not-constant
      e. CONSTANT: the temporary has a constant value
   2. Go thru the instructions in execution order
      a. Look at each operand in turn
      b. Check if it is constant
         i. Either if it is a constant directly
         ii. Or if it is a temporary which is a CONSTANT
            ➢ Replace the temporary with the constant
      c. If all constant
         i. Build an array of values
         ii. Do the appropriate constant arithmetic
         iii. Save the new value with the temporary as CONSTANT
      d. Otherwise
         i. Set temporary as BOTTOM
   3. Possible problems
      a. Avoid computations that might generate an error
      b. Avoid computations that might be different on the target machine
         i. E.g. different floating point arithmetics
         ii. Different floating point modes
         iii. Handling of overflow and underflow
         iv. Different word sizes
      c. Avoid computations that might have side effects
d. Still generating constant instructions (not removing code)

III. Value Numbering

A. We want to record what values are stored in temporaries
   1. In particular we want to know when two values are the same
   2. Note that the values can be computed differently and still be the same
      a. Constant propagation
      b. Arithmetic Identities \((A + B = B + A)\)
         i. Are these always true?
   3. We record this by value numbering
      a. We represent potential values with a number
      b. These values change as we proceed through the computation
      c. Two values are known to be equivalent if they have the same value number
      d. Different value numbers imply that things might be different
         i. But they don’t have to be

B. Objective
   1. Determine if two instructions are equivalent
      a. If they are and the value we want is available, we don’t have to execute the second instruction; we can just use the prior value
      b. This works if the prior value is a constant
      c. This works for duplicate computations
   2. Two constants are equivalent if they are identical constants
   3. Two temporaries are equivalent if they have the same value number
   4. Two instructions without side effects are equivalent if they have the same operator and have equivalent inputs
   5. Two load instructions are equivalent if they load equivalent addresses and no store operation has occurred between them that might modify that storage location.
   6. Otherwise two instructions are different

C. Data structures for value numbering
   1. Map from Argument to value number
      a. Argument is either temporary or constant
      b. Used for getting value numbers of operands
   2. Map from Temporary to defining instruction
      a. Tells what instruction defines that temporary
   3. Value numbers for instructions map
a. Op code and value numbers of the operands -> value number of result

D. Basic implementation
1. Insert an instruction into the value numbering
   a. Get the value of the instruction
   b. Add it to the current data structures
2. Getting the value of an instruction
   a. Returns instruction value and value number
      i. Instruction value is the index for the instruction table
      ii. Value number can be undefined (generate new)
   b. Case on the op code
      i. MOVE: just copy the value number
      ii. STORE: kill all relevant loads; set the value number for the corresponding load
      iii. CALL: kill all loads; generate new number for the result
      iv. LDC/STRING: get result from constant table
      v. ALLOC: new number
      vi. ADD: handle commutativity
      vii. Other operators are handled similarly
      viii. Can also handle other arithmetic rules here (A+0,A*1, ...)
   c. Look up instruction with operands in instruction table
      i. Use old value if there, add new value if not
3. Killing loads
   a. For all value table entries, if the base instruction is a load, then if the addresses can be the same, set its value number to undefined (0)
   b. This is language dependent

IV. Local Optimization Loop
A. For each instruction in evaluation order
   1. If the result is already available (based on value numbering)
      a. Either eliminate the op code or generate a move instruction from old value to new value
      b. Or track where to get the value from
   2. Check identities using current constant map and value numbering
   3. Do constant propagation
   4. Insert instruction into value numbering
B. Identities without side effects
   1. \(X + 0 = 0 + X = X - 0 = X\)
2. \(0 - X = -X\)
3. \(X \times 1 = 1 \times X = X / 1 = X ^{\text{**}} 1 = X\)
4. \(A\, \text{and} \, \text{TRUE} = \text{TRUE}\, \text{and} \, A = A\)
5. \(A\, \text{or} \, \text{FALSE} = \text{FALSE}\, \text{or} \, A = A\)
6. \(A\, \text{and} \, A = A\, \text{or} \, A = A\)

C. Identities with side effects
1. \(X - X = 0 \times X = X \times 0 = 0\)
2. \(0 / X = 0 \text{ if } X \neq 0\)
3. \(X / X = 1 \text{ if } X \neq 0\)
4. \(A == A = \text{true}...\)

D. Operator optimizations
1. \(A + (-A) = 0\)
2. \(-(-A) = A\)
3. \((-A)/(-B) = A/B\)
4. \(A - B \, <\text{rop}> \, 0 = A \, <\text{rop}> \, B\)
5. ...

E. Coding these up
1. These can be hand coded
   a. Fast, done once, but a bit of code to maintain
2. Alternatively, you can create a table-driven algorithm
3. Basic data comes from value numbering information
   a. From this instruction
   b. Temporaries go to defining instruction
   c. Constant operands get constant values from constant map
4. Result is to change or remove some instructions

F. Local optimization here is limited
1. Want to know something about the rest of the routine
   a. Where variable definitions come from
   b. What are the value numbers for temporaries outside the block
2. What definition (assignment) of a temporary reaches a block
3. What uses does a definition actually reach (where is it used)
4. In general, might be more than one definition
5. We’ll get back to this as we get deeper into optimization

G. Actual loop
1. For each instruction in execution order
2. Check identities
   a. If there are any changes, rescan
3. Check constant arithmetic
   a. Replace instruction with LDC
   b. Rescan
H. Dead code elimination
   1. If a computed value is local (temporary, not global)
   2. And that temporary is not a future operand in the block
   3. Then the computation can be discarded
   4. Unless there are side effects (e.g. calls)
I. Why bother with all this?