CS 526

Advanced

Compiler

Construction

http://misailo.cs.Illinois.edu/courses/cs526

STATIC SINGLE ASSIGNMENT

The slides adapted from Vikram Adve

SSA-Based Optimizations

- Dead Code Elimination (DCE)
- Sparse Conditional Constant Propagation (SCCP)
- Loop-Invariant Code Motion (LICM)
- Global Value Numbering (GVN)
- Strength Reduction of Induction Variables
- Live Range Identification in Register Allocation

Constant Propagation

Goals

Whenever there is a statement of the form v = Const, the uses of v can be replaced by Const.

Safety

Analysis: Explicit propagation of constant expressions

Transformation: Most languages allow removal of computations

Profitability

Fewer computations, almost always

Opportunity

Symbolic constants, conditionally compiled code, …

Simple Constant Propagation

```
Worklist = All statements in the SSA program
While not Worklist = ∅
      remove a statement S from Worklist
      if S has the form v = φ(c1, … cn) and c1=…=cn=Const,
            replace S by v = c
      if S is v = Const
            Delete S from the program
            For each Statement T ∈ Uses (v)
                  substitute v with C in T
                  Worklist = Worklist ∪ {T}
```

Extensions of the Algorithm

Copy propagation:

- Assignments `x = y` or `x = φ(y)` can be replaced by a simple use of y.

Constant folding:

- Assignments of the form x = a © b can be immediately evaluated if a and b are constants, and the statement replaced with x = c (c = a © b)

Constant conditions:

- If a condition if (x © y) always evaluate to true or false, then keep only one branch.

Conditional Constant Propagation: SCCP

Goals

Identify and replace SSA variables with constant values

Delete infeasible branches due to discovered constants

Safety

Analysis: Explicit propagation of constant expressions

Transformation: Most languages allow removal of computations

Profitability

Fewer computations, almost always (except pathological cases)

Opportunity

Symbolic constants, conditionally compiled code, …

Example I

```
J = 1;
...
if (J > 0)
    I = 1; // Always produces 1
else
    I = 2;
```

Example 2

```
I = 1;
...
while (...) {
    J = I;
    I = f(...);
    ...
    I = J; // Always produces 1
}
```

We need to proceed with the assumption that everything is constant until proved otherwise.

Example 3

```
I = 1;
...
while (...) {
    J = I;
    I = f(...);
    ...
    if (J > 0)
        I = J; // Always produces 1
}
```

For Ex. 1, we could do constant propagation and condition evaluation separately, and repeat until no changes. This separate approach is not sufficient for Ex. 3.

Conditional Constant Propagation*

Advantage:

Simultaneously finds constants + eliminates infeasible branches.

Optimistic

Assume every variable may be constant, until proven otherwise.

(Pessimistic ≡ initially assume nothing is constant.)

Sparse: Only propagates variable values where they are actually used or defined (using def-use chains in SSA form).

Iterative:

Build the list of constant definitions and uses using a worklist algorithm.

-sccp: Sparse Conditional Constant Propagation

Sparse conditional constant propagation and merging, which can be summarized as:

- Assumes values are constant unless proven otherwise
- Assumes BasicBlocks are dead unless proven otherwise
- Proves values to be constant, and replaces them with constants
- Proves conditional branches to be unconditional

Note that this pass has a habit of making definitions be dead. It is a good idea to run a DCE pass sometime after running this pass.

* Constant Propagation with Conditional Branches; M. Wegman and K. Zadeck, TOPLAS'91

Before next transformation…

Control Dependence

Def. Postdomination: node *p* postdominates a node *d* if all paths to the exit node of the graph starting at *d* must go through *p*

Def. In a CFG, **node Y is control-dependent on node B** if

- There is a non-empty path N0 = B,N1,N2, ...,Nk = Y such that Y postdominates N1 . . .Nk, and
- Y does not strictly postdominate B

Def. The Reverse Control Flow Graph (RCFG) of a CFG has the same nodes as CFG and has edge Y $\rightarrow$ X if X $\rightarrow$ Y is an edge in CFG.

- p is a postdominator of d iff p dominates d in the RCFG.

Computing Control Dependence

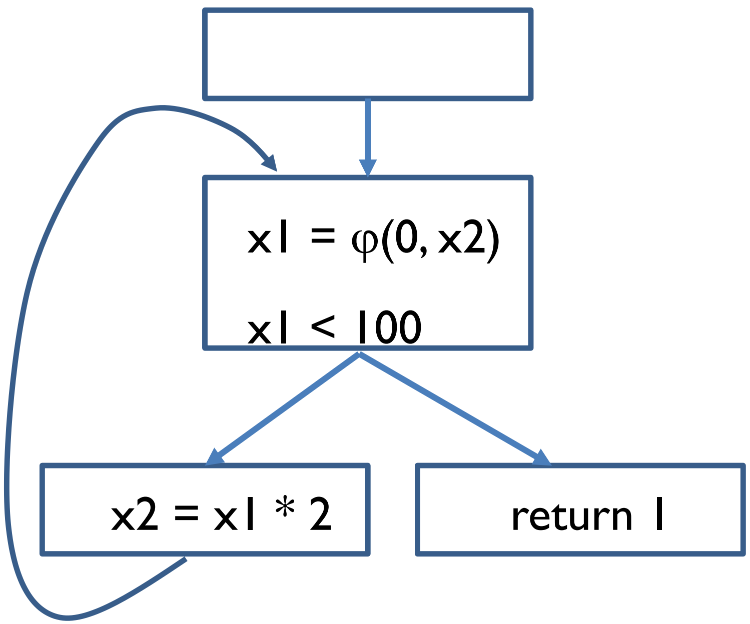
Key observation: Node Y is control-dependent on B *iff* $B \in DF(Y)$ in RCFG.

Algorithm:

1. Build RCFG
2. Build dominator tree for RCFG
3. Compute dominance frontiers for RCFG
4. Compute $CD(B) = \{Y \mid B \in DF(Y)\}$.

CD(B) gives the nodes that are control-dependent on B.

Aggressive Dead Code Elimination



Ordinary DCE:

- x2 is live because used in def. of x1.
- x1 is live because used in def. of x2.

Yet…

Idea: Analogous to SCCP, be optimistic and assume a statement is dead unless proven otherwise (i.e., contributes to the output)

Transform Passes

This section describes the LLVM Transform Passes.

-**adce**: Aggressive Dead Code Elimination

ADCE aggressively tries to eliminate code. This pass is similar to DCE but it assumes that values are dead until proven otherwise. This is similar to SCCP, except applied to the liveness of values.

Aggressive Dead Code Elimination

In each step, mark a statement as live if:

1. It is the **output** statement (e.g., return)
2. It has **known side effects** (e.g., assignment to global variable or calling a function with side effects)
3. It **defines a variable x** <u>used by an already live statement</u>
4. It is a **conditional branch**, and some other, already live statement is <u>control dependent</u> on the branch (and its block)

The algorithm then converges to a set of live variables

- Caveat: the algorithm may remove "empty" infinite loops

Induction Variable Substitution

Auxiliary Induction Variable

An auxiliary induction variable in a loop

```
for (int i = 0; i < n; i++) { … }
```

is any variable `j` that can be expressed as

```
c × i + m
```

at every point where it is used in the loop, where `c` and `m` are loop-invariant values, but `m` may be different at each use.

Optimization Goals

Identify linear expression for each auxiliary induction variable

- More effective dependence analysis, loop transformations
- Substitute linear expression in place of every use
- Eliminate expensive or loop-invariant operations from loop

Induction Variable Substitution

Auxiliary Induction Variable

```
for (int i = 0; i < n; i++) {
    j = 2*i + 1;
    k = -i;
    l = 2*i*i + 1;
    c = c + 5;
}
```

Induction Variable Substitution

Auxiliary Induction Variable

```
for (int i = 0; i < n; i++) {
    j = 2*i + 1;        // Y
    k = -i;             // Y
    l = 2*i*i + 1;      // N
    c = c + 5;          // Y*
}
```

Reminder: Strength Reduction

Goal: Replace expensive operations by cheaper ones

Primitive Operations: Many Examples

n * 2 → n << 1 (similarly, n/2)

n ** 2 → n * n

Recurrences

Example: …=a[i] to (base(a) + (i-1) * 4)

Such recurrences are common in array address calculations

Induction Variable Substitution

Strategy

- Identify operations of the form:

  $$x \leftarrow iv \times c, x \leftarrow iv \pm c$$

  iv: induction variable or another recurrence

  c : loop-invariant variable

- Eliminate **multiplications** from the loop body
- Eliminate induction variable if the **only remaining use** is in the loop **termination test**

Induction Variable Substitution

```
do i = 1 to 100
   sum = sum + a(i)
enddo
```

Source code

```
    sum = 0.0
    i = 1
L:  t1 = i - 1
    t2 = t1 * 4
    t3 = t2 + a
    t4 = load t3
    sum = sum + t4
    i = i + 1
    if (i <= 100) goto L
```

Intermediate code

$$
\begin{array}{ll}
 & \mathtt{sum}_0 = 0.0 \\
 & \mathtt{i}_0 = 1 \\
\mathtt{L:} & \mathtt{sum}_1 = \phi(\mathtt{sum}_0, \mathtt{sum}_2) \\
 & \mathtt{i}_1 = \phi(\mathtt{i}_0, \mathtt{i}_2) \\
 & \mathtt{t1}_0 = \mathtt{i}_1 - 1 \\
 & \mathtt{t2}_0 = \mathtt{t1}_0 * 4 \\
 & \mathtt{t3}_0 = \mathtt{t2}_0 + \mathtt{a} \\
 & \mathtt{t4}_0 = \mathtt{load}\ \mathtt{t3}_0 \\
 & \mathtt{sum}_2 = \mathtt{sum}_1 + \mathtt{t4}_0 \\
 & \mathtt{i}_2 = \mathtt{i}_1 + 1 \\
 & \mathtt{if}\ (\mathtt{i}_2 <= 100)\ \mathtt{goto}\ \mathtt{L}
\end{array}
$$

SSA form

Induction Variable Substitution

```
      sum₀ = 0.0
      i₀ = 1
L:    sum₁ = φ(sum₀,sum₂)
      i₁ = φ(i₀,i₂)
      t1₀ = i₁ - 1
      t2₀ = t1₀ * 4
      t3₀ = t2₀ + a
      t4₀ = load t3₀
      sum₂ = sum₁ + t4₀
      i₂ = i₁ + 1
      if (i₂ <= 100) goto L
```

SSA form

```
      sum₀ = 0.0
      i₀ = 1
      t5₀ = a
L:    sum₁ = φ(sum₀,sum₂)
      i₁ = φ(i₀,i₂)
      t5₁ = φ(t5₀,t5₂)
      t4₀ = load t5₀
      sum₂ = sum₁ + t4₀
      i₂ = i₁ + 1
      t5₂ = t5₁ + 4
      if (i₂ <= 100) goto L
```

After strength reduction

Induction Variable Substitution

```
    sum0 = 0.0
    i0 = 1
    t50 = a
L:  sum1 = φ(sum0,sum2)
    i1 = φ(i0,i2)
    t51 = φ(t50,t52)
    t40 = load t50
    sum2 = sum1 + t40
    i2 = i1 + 1
    t52 = t51 + 4
    if (i2 <= 100) goto L
```

$$
\begin{array}{ll}
 & sum_0 = 0.0 \\
 & i_0 = 1 \\
 & t5_0 = a \\
L: & sum_1 = \phi(sum_0, sum_2) \\
 & i_1 = \phi(i_0, i_2) \\
 & t5_1 = \phi(t5_0, t5_2) \\
 & t4_0 = \text{load } t5_0 \\
 & sum_2 = sum_1 + t4_0 \\
 & \boxed{\begin{array}{l} i_2 = i_1 + 1 \\ t5_2 = t5_1 + 4 \end{array}} \\
 & \text{if } (i_2 \le 100) \text{ goto } L
\end{array}
$$

After strength reduction

$$
\begin{array}{ll}
 & sum_0 = 0.0 \\
 & t5_0 = a \\
L: & sum_1 = \phi(sum_0, sum_2) \\
 & t5_1 = \phi(t5_0, t5_2) \\
 & t4_0 = \text{load } t5_0 \\
 & sum_2 = sum_1 + t4_0 \\
 & t5_2 = t5_1 + 4 \\
 & \boxed{\text{if } (t5_2 \le 396 + a) \text{ goto } L}
\end{array}
$$

After induction variable substitution

Induction Variable Substitution (recap)

```
        sum₀ = 0.0
        t5₀ = a
L:      sum₁ = φ(sum₀,sum₂)
        t5₁ = φ(t5₀,t5₂)
        t4₀ = load t5₀
        sum₂ = sum₁ + t4₀
        t5₂ = t5₁ + 4
        if (t5₂ <= 396 + a) goto L
```

After induction variable substitution

```
        sum₀ = 0.0
        i₀ = 1
L:      sum₁ = φ(sum₀,sum₂)
        i₁ = φ(i₀,i₂)
        t1₀ = i₁ - 1
        t2₀ = t1₀ * 4
        t3₀ = t2₀ + a
        t4₀ = load t3₀
        sum₂ = sum₁ + t4₀
        i₂ = i₁ + 1
        if (i₂ <= 100) goto L
```

Before induction variable substitution

References

Cocke and Kennedy, CACM 1977 (superseded by the next one).

Allen, Cocke and Kennedy, “Reduction of Operator Strength,” In Program Flow Analysis: Theory and Applications, 1981.

Classical Approach

- ACK: Classic algorithm, widely used.
- works on “loops” (Strongly Connected Regions) of flow graph
- uses def-use chains to find induction variables and recurrences

Cooper, Simpson & Vick, 2001, “Operator Strength Reduction,” Trans. Prog. Lang. Sys. 23(5), Sept. 2001.

SSA-based algorithm

- Same effectiveness as ACK, but faster and simpler
- Identify induction variables from SCCs in the SSA graph

Value Numbering

Code:

```
a = x + y

b = x + y

a = 1
c = x + y
d = y + x
e = d - 1
f = e + 1
```

- **Analysis:** Determining equivalent computations (variables, expressions, consts)
- **Transformation:** Eliminates duplicates with a semantics-preserving optimization
- Form of redundancy elimination

Value Numbering

- Assign an **identifying number** to each variable / expression / constant:

  x and y have same **id number**

  $\Leftrightarrow$ x = y **for all** inputs

- Use algebraic identities to simplify expressions
- Discover redundant computations & replace them
- Discover constant values, fold & propagate them

Value Numbering

- Use algebraic identities to simplify expressions
  - Commutativity (a+b = b+a), a+b+c = c+b+a, (a+b)^2 = a^2+2ab+b^2…
- Discover redundant computations and replace them
  - E.g., `y=2*x; z=2*x+1 => y=2*x; z=y+1`
- Discover constant values, fold & propagate them
  - After SCCP: e.g., `x=1; y = x+1 => y = 1+1`
  - Evaluate constant expression (`y = 2`) then propagate

Local Value Numbering

- Each variable, expression, & constant gets a **"value number"** (hash code)

  Same value number ⇒ same value

- **Prerequisites:** low-level intermediate code and existing basic blocks
- Equivalence based solely on facts from within the **single basic block**
- If an instruction's value number is already defined, instr. can be eliminated & subsequent references subsumed
- Constant folding is simple

Local Value Numbering

```
a = x + y
```

V1 ← hash(+,VN[x],VN[y]),
Name[V1] ← a

```
b = x + y
```

hash(+,VN[x],VN[y]) == V1
So, replace x+y with a. Transformed: b = a

```
a = 1
```

Name[V1] ←∅ (can we be more precise?)

```
c = x + y
d = y + x
e = d - 1
f = e + 1
```

Can we replace?

Challenges:
tracking where each value resides
commutativity ⇒ ???
identities (e.g., Vx OR Vx × 1): ⇒
instr. gets value number of operand (Vx)

Local Value Numbering

```
a1 = x + y
```

$V1 \leftarrow$ hash(+,VN[x],VN[y]),
Name[V1] $\leftarrow$ a

```
b = x + y
```

hash(+,VN[x],VN[y]) == V1
So, replace x+y with a. Transformed: b = a1

```
a2 = 1
```

Name[V1] $\leftarrow \emptyset$ (don't need anymore)

```
c = x + y
d = y + x
e = d - 1
f = e + 1
```

b = a1

c = a1

d = a1

Challenges:
tracking where each value resides
commutativity $\Rightarrow$???
identities (e.g., Vx OR Vx × 1): $\Rightarrow$
instr. gets value number of operand (Vx)

Local Value Numbering

```
a1 = x + y
```

V1 ← hash(+,VN[x],VN[y]),
Name[V1] ← a

```
b1 = a1
```

hash(+,VN[x],VN[y]) == V1
So, replace x+y with a. Transformed: b = a1

```
a2 = 1
```

Name[V1] ←Ø (don't need anymore)

```
c1 = a1
```

b = a1

```
d1 = a1
```

c = a1

```
e = d1 - 1
f = e + 1
```

d = a1

Challenges:
What happens with e and f ?

Local Value Numbering

For each instruction i : x ← y op z in the block

V1 ← VN[y]

V2 ← VN[z]

let v = hash(op,V1,V2)

if (v exists in hash table)

replace RHS with Name[v]

else

enter v in hash table

VN[x] ← v

Name[v] ← ti (new temporary)

replace instruction with: "ti ← y op z; x ← ti"

Local Value Numbering (LVN)

Simplifications

- If all operands have the same value number i.e. z=x op y, and VN[x] = VN[y]
  - if *op* is MAX, MIN, AND, OR, . . replace *op* with a copy operation (z=x)
  - if *op* tests equality, replace it with z=true
  - if *op* tests inequality replace it with z=false
- if all operands are constants and we haven't already simplified the expression, then immediately evaluate the resulting constant and propagate constants down
- if one operand is constant and we haven't yet simplified the expression:
  - if a constant operand is zero, replace ADD and OR with another operand; replace MULT, AND with zero
  - if constant operand is one, replace MULT with assignment of another operand

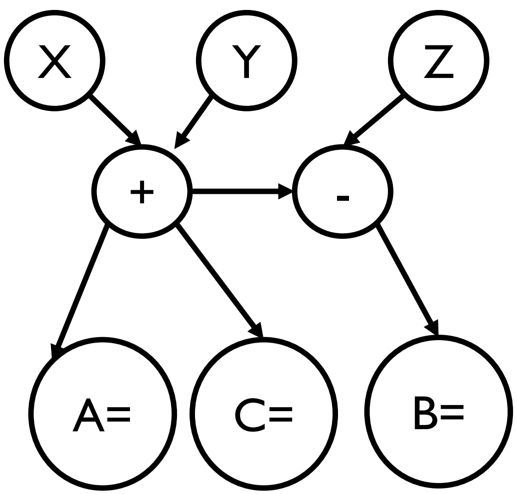
Local VN *Simplifications*

- If the operands have the same value number i.e. z=x op y, and VN[x] = VN[y]
  - if *op* is MAX, MIN, AND, OR, . . replace *op* with a copy operation (z=x)
  - if *op* tests equality, replace it with z=true
  - if *op* tests inequality replace it with z=false
- if all operands (x,y) are constants and we haven't already simplified the expression, then immediately evaluate the resulting constant and propagate constants down
- if one operand is constant and we haven't yet simplified the expression:
  - if a constant operand is zero, replace ADD and OR with another operand; replace MULT, AND with zero
  - if constant operand is one, replace MULT with assignment of another operand
- If *op* commutes, reorder its operands into **ascending order by value number** (canonical form)

Local VN *Analogy*

- Constructing a DAG from a forest (set of trees)
- Each expression is a node in a dag, edges are uses of the expression in the instructions
- Start from the leading instruction of the basic block
- Collapse nodes that are repeated into a single node and connect the edges to all uses

```
a = x + y
b = (x + y) - z
c = y + x
```



Global Value Numbering

```
W = X + Y;
if (...) {
  Z = X + Y;
  X = 1;
} else {
  Z = X + Y - 1;
}


Z = X + Y - 1;    // ??
```

Global Value Numbering

```
W1 = X1 + Y1;
if (...) {
  Z1 = X1 + Y1;
  X2 = 1;
} else {
  Z2 = X1 + Y1 – 1;
}
X3 = Phi(X1, X2)
Z3 = Phi(Z1, Z2)
Z4 = X3 + Y1 – 1;   // ??
```

Global Value Numbering

```
W1 = X1 + Y1;
if (...) {
  Z1 = X + Y;
  W2 = 1;
} else {
  Z2 = X + Y - 1;
}
W3 = Phi(W1, W2)
Z3 = Phi(Z1, Z2)
Z4 = X1 + Y1 - 1;   // ??
```

Global Value Numbering

```
T1 = X1 + Y1; W1 = T1;
if (...) {
  Z1 = X1 + Y1;
  W2 = 1;
} else {
  Z2 = X1 + Y1 - 1;
}
W3 = Phi(W1, W2)
Z3 = Phi(Z1, Z2)
Z4 = X1 + Y1 - 1;   // ??
```

Global Value Numbering

```
T1 = X1 + Y1; W1 = T1;
if (...) {
  Z1 = T1;
  W2 = 1;
} else {
  Z2 = T1 – 1;  // X1 + Y1 - 1
}
W3 = Phi(W1, W2)
Z3 = Phi(Z1, Z2)
Z4 = X1 + Y1 – 1;   // ??
```

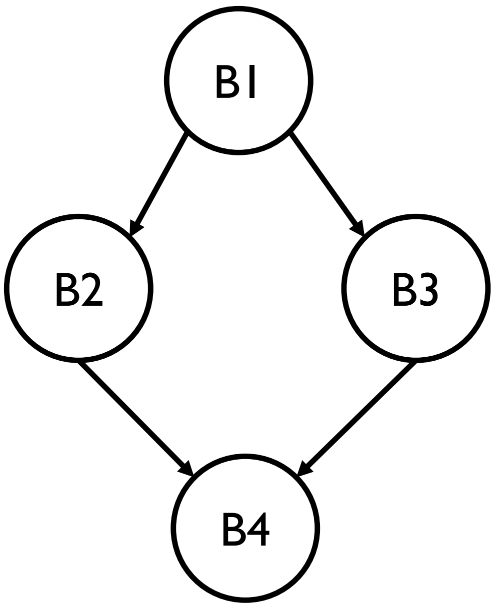
Global Value Numbering

```
T1 = X1 + Y1; W1 = T1;
if (...) {
  Z1 = T1;
  W2 = 1;
} else {
  Z2 = T1 – 1;  // X1 + Y1 - 1
}
W3 = Phi(W1, W2)
Z3 = Phi(Z1, Z2)
Z4 = T1 – 1;   // ??
```

Global Value Numbering (DVTN)

The Dominator-based VN Technique **(DVNT)**

- B2, B3 can be value-numbered using B1's table
- How about B4? Yes, can use the expressions from B1 (dominator node) but needs to invalidate the expressions killed in B2, B3
- Still based on hashing
- **BUT:** difficult to merge these tables
    - A variable may be redefined in B2, B3, or both



Example – Need for Global VN

```
X0 = 1
Y0 = 1
while (. . .) {
      X1 = ϕ(X0, X2)
      Y1 = ϕ(Y0, Y2)
      X2 = X1 + 1
      Y2 = Y1 + 1
}
```

Instruction Congruence

Instructions i & j are **congruent** iff

1. They are the same instruction
2. They are assignments of constants, which are equal (e.g. $x:=c_i$, $y:=c_j$ and $c_i==c_j$)
3. They have one or multiple operands, e.g.,

   $z_i = x_i$ op y_i

   $z_j = x_j$ op y_j

   same operator and their operands are **congruent** (x_i congruent to x_j and y_i congruent to y_j), taking into consideration commutativity of op.

A Global Approach (Alpern, Wegman & Zadeck)

Prerequisite: Computation in SSA Form

Algorithm:

1. partition instructions into congruence classes by opcode
2. *worklist* ← all classes
3. while (*worklist* is not empty)
   a) remove a **class c** from *worklist*
   b) for each **class s** that uses some **x ∈ c**
      while (s != ∅) do
      i. split **s** into **s & s′**: all users of c in one class
      ii. put smaller of s or s′ onto *worklist*
4. pick a representative instruction for each partition and perform replacement

Properties of the Algorithm

- Cannot prove congruences that involve different operators:
  - $5 \times 2 \sim= 7 + 3$ or
  - $3 + 1 \sim= 2 + 2$ or
  - $x \times 1 \sim= x$
- Need separate pass to transform code (partitioning must complete first)
- Powerful technique but ignores compile-time costs

- Alternative: SCC Based Algorithm (see references)
  - SCC often beats AWZ in practice

References

Long history in literature

- form of redundancy elimination (compare CSE)
- local version using hashing: late 60's Cocke & Schwartz, 1969
- algorithms for blocks, extended blocks, dominator regions, entire procedures, and (maybe) whole programs
- easy to understand algorithm for single block
- larger scopes cause more complex algorithms

1. Alpern, Wegman & Zadeck, "Detecting Equality of Variables in Programs," *Proceedings POPL* 1988

2. Cooper & Simpson, "SCC-Based Value Numbering," *Rice University TR CRPC-TR95636-S*, 1995.

3. Briggs, Cooper, Simpson, "Value Numbering," *Software–Practice and Experience*, June 1997 (supplementary only).

Optimizations where we will need more information

- Copy Propagation
- Global Common Subexpression Elimination (GCSE)
- Partial Redundancy Elimination (PRE)
- Redundant Load Elimination
- Dead or Redundant Store Elimination
- Code Placement Optimizations