SSA-based Optimization
(Objectives)

- Given an CFG with \( \phi \)-nodes, the student will be able to perform global common subexpression elimination (redundancy elimination) using a dominator-based approach.
- Given a CFG in SSA form, the student will be able to perform global constant propagation.
- Given a CFG in SSA form, the student will be able to perform strength reduction by finding loop, calculating loop invariants, finding induction variables and then applying the strength reduction transformation.
- Given a CFG in SSA form, the student will be able to perform dead-code elimination.
- Given a CFG in SSA form, the student will be able to perform global value numbering.
Dominator-based Global Common Subexpression Elimination

- A limited form of global CSE
  - used before dependence based optimization and other SSA-based optimizations
  - no code motion
  - redundancy found only along paths in the dominator tree
- In SSA all syntactically equivalent expression are semantically equivalent.
- Method:
  - keep a block structured table of available expression
    - StartBlock - add a scope in the expression table for this block.
    - EndBlock - remove the scope for the current block
  - perform CSE on the way back up the dominator tree while constructing SSA.
Algorithm

OPTRENAME(b) {
    for each $T_0 = \phi(T_1, \ldots, T_n) \in \Phi(b)$
        push NewName() on NameStack($T_0$)
    StartBlock(b)
    for each $I \in b$ in execution order {
        for each $T \in$ Operand($I$)
            replace $T$ by Top(NameStack($T$))
        if $I$.expr() \in AVAIL { // insert if \not\in AVAIL
            $T = I$.val()
            push GetTarget(AVAIL, i) on NameStack($T$)
            DEAD \cup \{I\}
        }
    }
}
Algorithm

else
    push NewName() on Top(NameStack(I.lval()))
}
for each $s \in \text{succ}(b)$ {
    $j = \text{WhichPredecessor}(s,b)$
    for each $T_0 = \phi(T_1, ..., T_n) \in \Phi(s)$
        replace $T_j$ with Top(NameStack($T_j$))
}
for each $c \in \text{children}(b)$
    OPTRENAME($c$)
Algorithm

for each $I \in b$ in reverse order {
    $X = \text{Pop}(\text{NameStack}(I.lval()))$
    if $I \in \text{DEAD}$
        remove $I$
    else
        replace $I.lval()$ with $X$
} 

for each $T_0 = \phi(T_1, \ldots, T_n) \in \Phi(b)$
    replace $T_0$ by $\text{Pop}(\text{NameStack}(T_0))$
EndBlock(b)
Example

1. $x = y + z$
   if ($p$)

2. $w = y + z$
   $y = w$

3. $v = y + z$

4. $y = \phi(y,y)$
   $w = \phi(w,w)$
   $v = \phi(v,v)$
   $x = y + z$
Constant Propagation

- Propagate constants globally on a sparse representation
  - cheaper than previous algorithm
- Incorporate the effects of branch folding
  - if a block cannot be reached, it will be ignored
- Meet operations occur at φ-nodes
- Algorithm in book wrong
  - evaluates instruction from blocks not executed
Algorithm

Procedure ConstProp {
    mark all edges in CFG not executable
    initialize all nodes in SSA

    Graph to T
    Work = ∅; Visited = ∅;
    Blocks = {ENTRY}

    while Work ≠ ∅ ∧ Blocks ≠ ∅{
        while Work ≠ ∅{
            take I from Work
            EvalInstruction(I)
        }
        take b from Blocks
        for each I ∈ Φ(b){
            EvalInstruction(I)
        }
        if b ∈ Visited {
            Visited := Visited ∪ {b}
            for each I ∈ b
                EvalInstruction(I)
        }
    }
}
Algorithm

EvalInstruction(I) {
    if I is an arithmetic instruction or ϕ-node {
        evaluate I
        if result lowered
            for each j ∈ Uses(I.val()) {
                propagate result
                if j.Block() ∈ Visited
                    Work ⊔= {j}
            }
    }
    else if I is a branch
        for each possible destination of I, S
            if edge from I.block() to S is not executable {
                mark it as executable
                Blocks ⊔= {S}
            }
}
Example

1. $x_0 = 1$ if $x_0 > 3$

2. $x_1 = x_0 + 5$

3. $x_2 = \phi(x_0, x_1)$
   $z_0 = x_2 + 10$
   print $z_0$
Strength Reduction

- Replace multiplication of a regularly varying variable by a constant in a loop with an addition.

Example

```c
i = 1
loop {
    j = 2*i
    i += 1
}
```

- Gets converted to

```c
j = 0;
i = 1
loop {
    j += 2
    i += 1
}
```

- Useful for enabling opportunities for auto-increment mode
- cheaper instructions
Method

1. Find loops in CFG
2. Find the variables in a loop that are loop invariant.
3. Find loop induction variables (vary regularly)
4. Reshape expressions into canonical form
5. Perform strength reduction
Step 1: Finding Loops

- **Defn**: A loop is a set of basic blocks, \( L \), such that if \( b_0, b_1 \in L \) then there is a path from \( b_0 \) to \( b_1 \) and from \( b_1 \) to \( b_0 \). A block \( b \in L \) is an entry block if \( b \) has a predecessor that is not in \( L \). A block \( b \in L \) is an exit block if \( b \) has a successor not in \( L \).
  - We will look at natural loops where the entry block dominates all other blocks in the loop (single entry).
- **Computing loops involves finding a block that has an incoming back edge (head dominates the tail).**
- **Modified from book, which does multiple entry loops (not natural)**
Loop Tree

- Organize the loops in a function hierarchically.
  - A loop L1 is a child of loop L2 in the loop tree iff L1 ⊆ L2
- The tree structure is recorded by (X is a loop or block)
  - LoopParent(X) - an attribute indicating which node in the tree of which this node is a child. It also indicates the loop in which a loop or block is contained. LoopParent(X) may be a special root node indicating that the loop is contained in no other loop.
  - LoopContains(X) - the set of children of a node in the loop tree. The blocks or loops contained in a loop.
  - LoopEntry(X) - the entry node of the loop.
Computing the Loop Tree

```plaintext
LoopTree() {
    compute post-order numbering for the CFG
    for each b ∈ G {
        LoopParent(B) = NIL
        LoopEntry(B) = B
        LoopContains(B) = B;
    }
    for each b ∈ G in postorder
        FindLoop(b)
        Make all nodes w/o parents have a Root node as parent
}
```
Computing the Loop Tree

FindLoop(b) {
    Loop = ∅; Found = false
    for each p ∈ pred(b)
        if b >> p {
            Found = true;
            if p ∈ Loop ∧ p ≠ b {
                Loop |= {p}
            }
        }
    if Found
        FindBody(Loop,b,single)
}
Computing the Loop Tree

```
FindBody(Generators,H,k) {
  Loop = ∅; Queue = ∅
  for each b ∈ Generators {
    L = LoopAncestor(b)
    if L ∉ Loop then {
      Loop ∪= {L}; Queue ∪= {L}
    }
  }
  while (Queue ≠ ∅) {
    b = Queue.Dequeue()
    Pred= pred(LoopEntry(b))
    for each p ∈ Pred
      if p ≠ H {
        L = LoopAncestor(p)
        if L ∉ Loop {
          Queue.Enqueue(p)
          Loop ∪= {p}
        }
      }
  }
}
```

```
Loop ⊆ {H}
X = new Loop Tree node
LoopContents(X) = Loop
LoopEntry(X) = b
LoopParent(X) = NIL
for each b ∈ Loop
  LoopParent(b) = X
```
Example

\[ b_0 \]
\[ b_1 \]
\[ b_2 \]
\[ b_3 \]
\[ b_4 \rightarrow b_5 \]
Step 2: Loop Invariants

- **Defn:** A variable is loop invariant if it is either not computed in a loop or its operands are invariant.
- **Compute** variant(T), the innermost loop in which T is not invariant.
  - if T = φ(...), T is defined to be variant in the innermost loop containing it.
  - for pure functions like add, variant in the innermost loop that one of the operands in variant
  - for a LOAD, it varies in the innermost loop in which a store operation might modify the same location.
- Walk the dominator tree in preorder
Finding Loop Invariants

CalcLoopInvariants(b) {
    for each $T_0 = \phi(T_1, ..., T_n) \in \Phi(b)$
        variant($T_0$) = LoopParent(b)
    for each $I \in b$ in order {
        Varying = Root
        for each $T \in$ Operands($I$) {
            TVarying = LoopNearestAncestor(variant($T$), b)
            if LoopNearestAncestor(Varying, TVarying) == Varying
                Varying = TVarying
        }
        variant($I$.val()) = Varying
    }
}
Finding Loop Invariants

LoopNearestAncestor(L1,L2) {
  if is_ancestor(L2,L1)
    return L2
  L = L1
  while !is_ancestor(L,L2)
    L = LoopParent(L)
  return L
}
Example

1. $l_0 = 1$

2. $l_k = \phi(t_0, t_2)$ if $l_1 \leq n$

3. $t_0 = 2$
   $j_0 = 1$

4. $j_1 = \phi(j_0, j_2)$
   $t_2 = \phi(t_0, t_2)$ if $j_1 \leq m$

5. $x_0 = l_1 * 4$
   $t_2 = x_0 + t_1$
   $j_3 = j_1 + 1$

6. $k_0 = 1$

7. $k_1 = \phi(k_0, k_2)$ if $k_1 \leq m$

8. $x_1 = t_1 + 3$
   $k_2 = k_1 + 1$

9. $l_2 = l_1 + 1$
Example: Loop Tree

```
R
  |____ b₁
  |     |____ L₁
  |     |     |____ b₁₀
  |     |     |     |____ b₂
  |     |     |     |     |____ b₃
  |     |     |     |____ L₂
  |     |     |     |     |____ b₆
  |     |     |____ L₃
  |     |     |     |____ b₉
  |____ b₁₀
  |     |____ L₁
  |     |     |____ b₁₀
  |     |     |     |____ b₂
  |     |     |     |     |____ b₃
  |     |     |     |____ L₂
  |     |     |     |     |____ b₆
  |     |     |____ L₃
  |     |     |     |____ b₉
  |____ b₁₀
```
Example: Dominator Tree
Step 3: Finding Induction Variables

- Defn: A temporary $T$ is a candidate temporary for loop $L$ iff $T$ is computed in $L$ and the computation has one of the following forms:
  a) $T = T_i \pm T_j$ where one operand is a candidate in $L$ and the other is loop invariant
  b) $T = \pm T_k$ where $T_k$ is a candidate in $L$ or is loop invariant in $L$
  c) $T = \phi(T_1, \ldots, T_n)$ where each of the operands is either a candidate in $L$ or a loop invariant in $L$
Algorithm: Finding Induction Variables

CalcCandidates(L) {
    Candidates = Φ
    Work = Φ
    for each b ∈ L {
        for each I ∈ Φ(b) ∪ b of the form T = ...
            if Typeof(T) is integer
                if T has candidate syntax {
                    Candidates ⊇ {T}
                    Work ⊇ {T}
                }
    }
Algorithm

while Work ≠ ∅ {
    take T from Work
    CandidatePrune(T)
    if T ∈ Candidates
        for each I ∈ Uses(T) where I ∈ L {
            U = I.val()
            if (U ∈ Candidates ∧ U ∉ Work)
                Work ⊎ {U}
        }
}
}
Algorithm

CandidatePrune(T) {
    I = T.instruction()
    case on form of I {
        T = \( \phi(T_1,\ldots,T_n) \):
            for i = 1, n
                if \( T_i \notin \text{Candidates} \land \text{invariant}(T_i,L) \) {
                    Candidates = \{T\}
                    return
                }
        }
    }
}
Algorithm

\[ T = T_i + T_j: \text{ if } T_i \in \text{Candidates} \land \text{invariant}(T_j,L) \]
\[ \text{return} \]
\[ \text{else} \]
\[ \text{if } T_j \in \text{Candidates} \land \text{invariant}(T_i,L) \]
\[ \text{return} \]
\[ \text{else} \{ \]
\[ \text{Candidates} := \{T\} \]
\[ \text{return} \]
\[ \} \]
Algorithm

\[ T = \pm T_k: \text{ if } T_k \not\in \text{Candidates} \land \text{linvariant}(T_k, L) \{ \\
\quad \text{Candidates} \setminus \{T\} \\
\quad \text{return} \\
\} \]

}
Example

- Detect induction variables in previous example
Induction Sets

- Consider a graph where candidates are nodes and an edge is between two nodes, T and U, if T is used to compute U. An induction temporary is a temporary in a SCC in this graph. An induction set is the set of temporaries in the SCC.
Example

\[ i_0 = 1 \]

\[ j_1 = \phi(j_2, i_0) \]
\[ i_1 = \phi(i_2, i_0) \]
\[ j_2 = i_1 \times 2 \]
\[ i_2 = i_1 + 1 \]
Algorithm

CalcInduction(L) {
    CalcCandidates(L)
    Construct candidate graph, G
    compute SCC(G)
    Anchors = \{T \mid T \text{ is a target of a } \phi\text{-node in LoopEntry}(L)\}
    for each s ∈ SCC(G)
        if |s| > 1 ∧ Anchors ∩ s ≠ ∅
            add s to InductionSets
}
Example

- Compute the induction variables in the previous example.
Step 4: Reshape Expression

- Use commutative, associative, and distributive properties to reshape expressions contained in n loops as

\[
E = E' + (LC_1 + (LC_2 + ... + LC_n)) \\
E' = E'' + FD_1*I_1 + FD_2*I_2 + ... + FD_m*I_m
\]

where \(LC_i\) is invariant in \(L_i\), \(I_i\) is the induction variable of \(L_i\) and \(FD_i\) is a loop invariant expression.
- \(LC_i\) can be moved outside of \(L_i\)
- Can cause an increase in cost (invariants into loops)
**Strength Reduction**

Consider an expression of the form $E = FD_i^*I_i + LC_i$
Let $IS_i$ be the induction set of $I_i$
Create temporaries $E_o,...,E_{q'}$, one for each element of $IS_i$ plus any initial values coming in from outside the loop.
for all $T_j \in T_k \neq c$ in the loop such that $T_j, T_k \in IS_i$
insert $E_j = E_k \pm FD_i^*c$ after this point
for all $T_j \neq T_k$ in the loop such that $T_j, T_k \in IS_i$
insert $E_j = \pm E_k$ after this point
replace uses of $E$ with the correspond $E_j$ whose definition reaches the use
replace $E = FD_i^*I_i + LC_i$ with the assignment $E = E_j$. If the block containing this assignment is executed on every path through the loop to a loop exit, it can be moved after the loop following each loop exit.
Handling $\phi$-nodes

- Given $T_0 = \phi(T_1, \ldots, T_n)$, $T_0 \in IS_i$, create a new $\phi$-node $E' = \phi(...)$
- for each predecessor block $P_j$
  - if the temporary $T_i$ is in the induction set of $T_0$, put the temporary holding $E$ at the end of $P_j$ in the $j^{th}$ position of the $\phi$-node for $E'(P_j$ must be in the loop because $T_j$ is in the induction set).
  - if $T_j$ is not in the induction set for $T_0$, insert the computation $E_j = FD_j^*T_j + c$ at then end of $P_j$, and place $E_0$ into the corresponding entry in the $\phi$-node for $E'$ ($P_j$ is not in the loop).
  - change $E'$ to be the exposed use of a temporary for $E$. 
Example

\[ i_0 = 1 \]

\[ j_1 = \phi(j_2, j_0) \]

\[ i_1 = \phi(i_2, i_0) \]

\[ j_2 = i_1 \times 2 \]

\[ i_2 = i_1 + 1 \]

\[ i_0 = 1 \]

\[ e_0 = 2 \times i_0 \]

\[ j_1 = \phi(e_2, j_0) \]

\[ i_1 = \phi(i_2, i_0) \]

\[ e_1 = \phi(e_2, e_0) \]

\[ i_2 = i_1 + 1 \]

\[ e_2 = e_1 + 2 \]

\[ j_2 = e_1 \]
Dead-code Elimination

- Use the SSA graph (sparse) to detect dead code.
- Method
  - remove instructions that do not directly or indirectly use data that is observable outside the procedure.
  - allow for branches that are never taken (can eliminate loops this way)
    - uses control dependence
Control Dependence

- Use the idea of postdominators
- Def^n: A block X postdominates a block B iff every path from B to Exit contains X.
- Def^n: pdom(B) represents the immediate postdominator of B and is the parent of B in the postdominator tree.
- Compute postdominators by computing the dominator relation on the reverse control flow graph.
Example
Control Dependence

- Consider two block B and X. When does B control the execution of X?
  1. If B has only one successor block, it does not control the execution of anything. B must have multiple successors.
  2. B must have some path leaving it that leads to the Exit block and avoids X. X cannot postdominate B.
  3. B must have some path leaving it that leads to X.
  4. B should be the latest block that has these properties.
Control Dependence

- A block $X$ is control dependent on an edge $(B, S)$ iff there is a non-empty path from $B$ to $X$ such that $X$ postdominates each block on the path except $B$. And, $X = B$ or $X$ does not postdominate $B$.
- Compute control dependence by find the dominance frontier of every node in the reverse control-flow graph.
Computing Control Dependence

foreach \( n \in \text{PDT} \) in postorder{
    DF(n) = \( \emptyset \)
    for each \( c \in \text{child}(n) \)
        for each \( m \in \text{DF}(c) \)
            if \(!n \gg m\)
                DF(n) \subseteq \{m\}
        for each \( m \in \text{succ}(n) \)
            if \(!l(n \gg m)\)
                DF(n) \subseteq \{m\}
}
Example
Algorithm

EliminateDeadCode()
    WorkList = ∅
    Necessary = ∅
    for each B ∈ N do
        for each I ∈ B do
            if (I stores into external data) ∨
               (I is an i/o instruction) ∨ (I is a call) {  
                Necessary ⊎ {I}
                WorkList ⊎ {I}
            }


Algorithm

while WorkList ≠ ∅ {
    take I from WorkList
    b = I.ContainingBlock()
    for each C on which B is control dependent {
        J = branch in C
        if J ∉ Necessary {
            Necessary ∪= {J}
            WorkList ∪= {J}
        }
    }
}
Algorithm

for each $T \in \text{Operand}(I)$
    $J = \text{Definition}(I)$
    if $J \in \text{Necessary}$
        Necessary $\cup= \{J\}$
        WorkList $\cup= \{J\}$
    
} 

for each $B \in N$
    for each $I \in B$
        if $I \in \text{Necessary}$
            remove $I$
        else if $I$ is a branch $\land I \notin \text{Necessary}$
            change branch to immediate postdominator of block

}
Example

1. $T_0 = I_0 = \text{branch1}$

2. $T_0 = \text{branch3}$

3. $I_1 = \phi(I_0, I_2)$
   $I_2 = I_1 + 1$
   \text{branch2}

4. write($T_0$)
Global Value Numbering

- Apply value numbering to a global context for better redundancy elimination.
- Associate a field for each temporary to hold its value number.
- If two temporaries have the same value number then they are equivalent.
- If there are no loops a reverse postorder walk of the CFG is sufficient (all operands defined before used).
- \( \phi \)-nodes can only be equivalent in the same basic block
  - need control-flow information to compare \( \phi \)-nodes from different blocks.
Global Value Numbering

- What can we do about SCCs in the SSA graph?
  - The value number of some operands will not be known when trying to process an instruction.
  - This will happen at $\phi$-nodes
  - Solution: assume the best case (an unknown value number does not affect the result) and iterate
  - Process nodes in an SCC in reverse postorder (as other nodes)
Processing $\phi$-nodes

- There are 3 possibilities
  1. If a corresponding entry for the $\phi$-node/block is already in the value table, then assign the target of this $\phi$-node the same value_representative value.
  2. Consider the operands that do not have a value_representative value of NULL. If at least two of them have different values, assign the target a new value # and enter it into the value table.
  3. Consider the operands that do not have a value_representative value of NULL. If all of them have the same value, then give the target the same value number and enter it into the table.
Efficiency Improvements

- If a $\phi$-node already has a non-null value number that is different than its operands, then a new value number is not needed.
- When processing a SCC, use a temporary value table called a scratch table. Once the values in the scratch table have stabilized, move the results to the value table.
Algorithm

procedure CalcGlobalValue {
    compute the SCC of the SSA graph: C_1,...,C_s
    ValTab = Ø; ScratchTab = Ø;
    for each T ∈ Temporaries
        ValRep(T) = NULL;
    for i = 1, s
        if |C_i| > 1 {
            call CalcGlobalValueSCC(C_i)
            for each T ∈ C_i in reverse postorder {
                I = Definition(T); U = ValRep(T);
                apply algebraic simplification to I
                if (opcode(I), ValRep(Operands(I))) ∉ ValTab
                    ValTab ⊔= {(opcode(I), ValRep(Operand(i))}
            }
        }
Algorithm

// let I be the single instruction in C
else if I is a φ-node
  CalcφValue(I, ValTab)
else {
  apply algebraic simplification to I
  T = Target(I)
  if (opcode(I), ValRep(Operands(I)) ∈ ValTab {
    ValRep(T) = new value number;
    ValTab ⊔ {(opcode(I), ValRep(Operands(I)))
  }
  else
    ValRep(T) = value from ValTab
}
Algorithm

procedure CalcGlobalValueSCC(C) {
    change = false;
    repeat
        for each T in reverse postorder {
            I = Definition(T)
            if I is a o-node
                CalcValue(I, ScratchTab)
            else {
                if (opcode(I), ValRep(Operands(I)) ∈ ScratchTab
                    NewValue = value in ScratchTab
                else {
                    NewValue = new value number
                    ScratchTab := (opcode(I), ValRep(Operands(I)))
                }
                if NewValue ≠ ValRep(T) {
                    change = true; ValRep(T) = new value number
                }
            }
        } until not(change)
    }
}
Algorithm

procedure CalcOfValue(I, Table) {
    Let I be $T_0 = \phi(T_1, ..., T_n)$
    if I $\notin$ Table {
        if $\exists T_i, T_j$ | ValRep$(T_i) \neq$ NULL $\land$ ValRep$(T_j) \neq$ NULL $\land$
            ValRep$(T_i) \neq$ ValRep$(T_j)$
            ValRep$(T_0) =$ new value #
        else
            ValRep$(T_0) =$ ValRep$(T_i)$ where ValRep$(T_i) \neq$ NULL
        Tab $\cup =$ {opcode(I), ValRep(Operands(I))}
        change = true;
    }
}
Example

1 \[I_0 = 1, \quad J_0 = 1\]

2 \[I_1 = \phi(I_0, I_2), \quad J_1 = \phi(J_0, J_2), \quad U = I_1 - J_1, \quad I_2 = I_1 + 1, \quad J_2 = J_1 + 1\]
Now What?

- Give all temporaries with the same value # the same name, and convert to normal form
- Apply common subexpression elimination
  - dominator-based
  - traditional AVAIL-based
  - partial redundancy elimination