CS 526

Advanced
Compiler
Construction

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

STATIC SINGLE ASSIGNMENT

The slides adapted from Vikram Adve

SSA Construction Algorithm

Steps:

1. Compute the dominance frontiers
2. Insert φ-functions
3. Rename the variables

Insert φ-functions

```
for each variable V
    HasAlready ← ∅                  //already processed nodes
    EverOnWorkList ← ∅              //nodes that have been on work list (never removed)
    WorkList ← ∅                    //nodes on the work list (never removed)
    for each node X that may modify V    // initialize work list
        EverOnWorkList ← EverOnWorkList ∪ {X}
        WorkList ← WorkList ∪ {X}
```

Insert φ-functions

```
for each variable V
    HasAlready ← ∅                  //already processed nodes
    EverOnWorkList ← ∅              //nodes that have been on work list (never removed)
    WorkList ← ∅                    //nodes on the work list (never removed)
    for each node X that may modify V     // initialize work list
        EverOnWorkList ← EverOnWorkList ⋃ {X}
        WorkList ← WorkList ⋃ {X}

    while WorkList ≠ ∅
        remove X from WorkList
        for each Y ∈ DF(X)          // Process nodes on the dominance frontier
            if Y ∉ HasAlready then
                insert a ϕ-node for V at Y
                HasAlready ← HasAlready ⋃ {Y}
                if Y ∉ EverOnWorkList then
                    EverOnWorkList ← EverOnWorkList ⋃ {Y}
                    WorkList ← WorkList ⋃ {Y}
```

Insert φ-functions

```
j=1;

while (j < X)
    ++j;

N = j;
```

Basic Block <a>

```
j = 1;

if (j >= X) goto E;
```

Basic Block <b>

```
S:
j = j+1;
if (j < X) goto S;
```

Basic Block <c>

```
E:
N = j;
```

Renaming Variables

Renaming definitions is easy – just keep the counter for each variable.

To **rename each use** of V :

(a) Use in a non-φ-functions: Use immediately dominating definition of V (+ φ nodes inserted for V).

preorder on Dominator Tree!

(b) Use in a φ-function operand: Use the definition that immediately dominates incoming CFG edge (not φ)

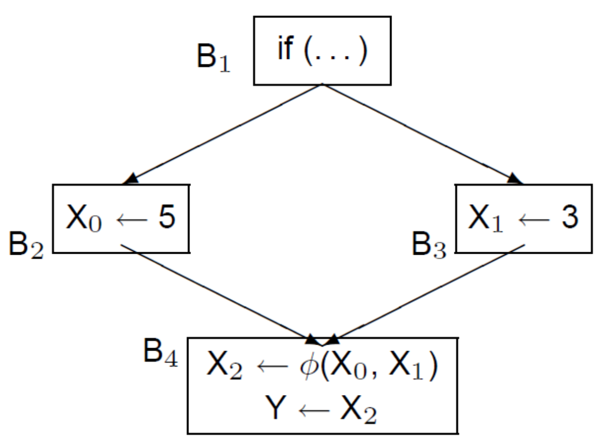
rename the φ-operand when processing the predecessor basic block!

Translating Out of SSA Form

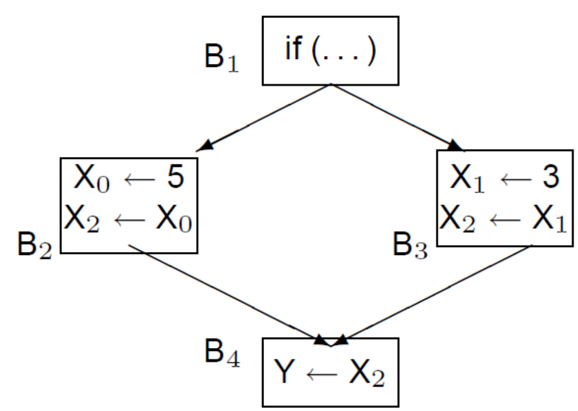
Overview:

1. Dead-code elimination (prune dead φs)
2. Replace φ-functions with copies in predecessors
3. Register allocation with copy coalescing

Before Step 2



After Step 2



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 N_0 = B,N_1,N_2, ...,N_k = Y such that Y postdominates $N_1 \ldots N_k$, 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.

Computing Control Dependence

Key observation:
Node Y is control-dependent on Node B *iff*
B ∈ 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 | B ∈ DF(Y)}.

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

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

(Sparse) 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 1

```
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 ($\top$), until proven otherwise.

(In contrast, Pessimistic would initially assume nothing is constant ($\bot$).)

Sparse:

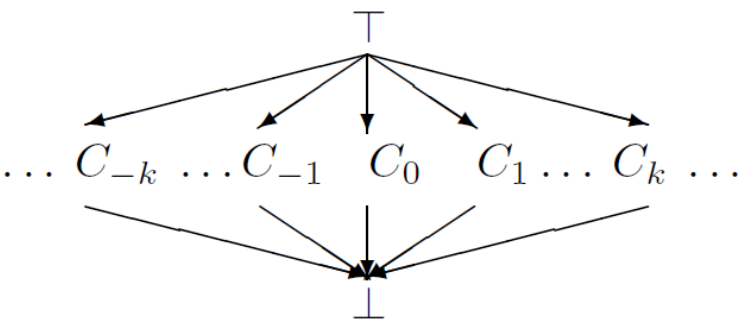
Only propagates variable values where they are actually used or defined (using def-use chains in SSA form).

SSA vs. def-use chains:

Much faster: SSA graph has fewer edges than def-use graph

Paper claims SSA catches more constants (not convincing)

Conditional Constant Propagation



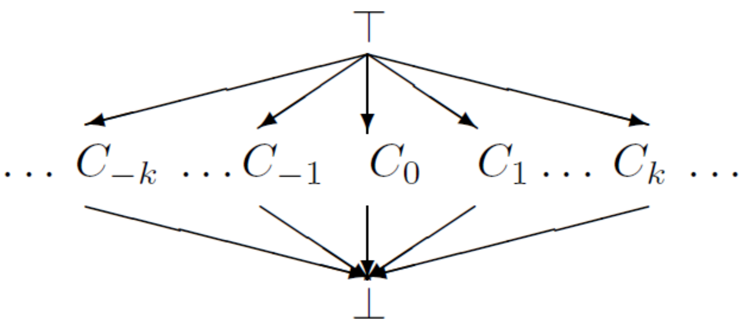
Lattice L

Lattice $L \equiv \{\top, C_i, \bot\}$.
$\top$ intuitively means "*May be constant.*"
$\bot$ intuitively means "*Not constant.*"

Reminder: Definition of Lattice

1) Partially ordered set (L, $\prec$) i.e., the pair (set + partial order relation)
2) Every two elements have a **join** (least upper bound)
3) Every two elements have a **meet** (greatest lower bound

Conditional Constant Propagation



Lattice L

Lattice $L \equiv \{\top, C_i, \perp\}$.
$\top$ intuitively means "*May be constant.*"
$\perp$ intuitively means "*Not constant.*"

Meet Operator, $\sqcap$

$$\top \sqcap X = X, \quad \forall X \in L$$
$$\perp \sqcap X = \perp, \quad \forall X \in L$$
$$C_i \sqcap C_j = \begin{cases} C_i, & \textit{iff } i = j, \\ \perp, & \textit{otherwise} \end{cases}$$

Intuition: A Partial Order $\prec$

$\perp \prec C_i$ for any C_i.
$C_i \prec \top$ for any C_i.
$C_i \not\prec C_j$ (i.e., no ordering).

Meet of X and Y ($X \sqcap Y$) is the greatest value $\preceq$ both X and Y.

Conditional Constant Propagation

Assume:

Only assignment or branch statements

Every non-φ statement is in separate BB

Key Ideas

1. Constant propagation lattice = { ⊤,Ci,⊥ }
2. Initially:
   - every def. has value ⊤ ("may be constant").
   - every CFG edge is infeasible, except edges from s
   - Use 2 worklists: FlowWL (for edges) and SSAWL (for SSA edges)
3. Highlights:
   - Visit S only if some incoming edge is executable
   - Ignore ϕ argument if incoming CFG edge not executable
   - If variable changes value, add SSA out-edges to SSAWL
   - If CFG edge executable, add to FlowWL

SCCP Algorithm

```
Initialize(ExecFlags[], LatCell[], FlowWL, SSAWL);
while ((Edge E = GetEdge(FlowWL ∪ SSAWL)) != 0) {

   if (E is a flow edge && ExecFlag[E] == false) {
       ExecFlag[E] = true
       VisitPhi(ϕ) ∀ ϕ ∈ E->sink
       if (first visit to E->sink via flow edges)
          VisitInst(E->sink)
       if (E->sink has only one outgoing flow edge Eout)
          add Eout to FlowWL
   } else if (E is an SSA edge) {
       if (E->sink is a ϕ node)
          VisitPhi(E->sink, ExecFlags, SSAWL)
       else if (E->sink has 1 or more executable in-edges)
          VisitInst(E->sink)
   }
```

SCCP Algorithm

VisitPhi(ϕ) :

```
for (all operands Uk of ϕ) {
    if (ExecFlag[InEdge(k)] == true)
       LatCell(ϕ) ⊓ = LatCell(Uk)
    if (LatCell(ϕ) changed)
       add SSAOutEdges(ϕ) to SSAWL
}
```

VisitInst(S) : **[note: Many errors in Muchnick]**

```
val = Evaluate(S)
LatCell(S) = val
if (LatCell(S) changed) // cannnot be Top
   if (S is Assignment)
      add SSAOutEdges(S) to SSAWL
   else // S must be a Branch
      add one or both outgoing edges to FlowWL
```

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: (base + (i-1) * 4)

- Such recurrences are common in array address calculations
- Note: Aux. induction variables are just a special case

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

$$
\begin{array}{ll}
 & sum_0 = 0.0 \\
 & i_0 = 1 \\
L: & sum_1 = \phi(sum_0, sum_2) \\
 & i_1 = \phi(i_0, i_2) \\
 & t1_0 = i_1 - 1 \\
 & t2_0 = t1_0 * 4 \\
 & t3_0 = t2_0 + a \\
 & t4_0 = \text{load } t3_0 \\
 & sum_2 = sum_1 + t4_0 \\
 & i_2 = i_1 + 1 \\
 & \text{if } (i_2 <= 100) \text{ goto } L
\end{array}
$$

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
```

After strength reduction

```
     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

Induction Variable Substitution

```
        sum_0 = 0.0
        t5_0 = a
L:      sum_1 = φ(sum_0,sum_2)
        t5_1 = φ(t5_0,t5_2)
        t4_0 = load t5_0
        sum_2 = sum_1 + t4_0
        t5_2 = t5_1 + 4
        if (t5_2 <= 396 + a) goto L
```

After induction variable substitution

```
        sum_0 = 0.0
        i_0 = 1
L:      sum_1 = φ(sum_0,sum_2)
        i_1 = φ(i_0,i_2)
        t1_0 = i_1 - 1
        t2_0 = t1_0 * 4
        t3_0 = t2_0 + a
        t4_0 = load t3_0
        sum_2 = sum_1 + t4_0
        i_2 = i_1 + 1
        if (i_2 <= 100) goto L
```

SSA form

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

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