COS320 Control Flow Theory

Static Single Assignment Form

SSA: Each %vid appears on the lhs of at most one assignment in a CFG

ϕ-Nodes: SSA in conditional branching

```
if (x<0) {
    y := y - x;
} else {
    y := y + x;
}
return y
```

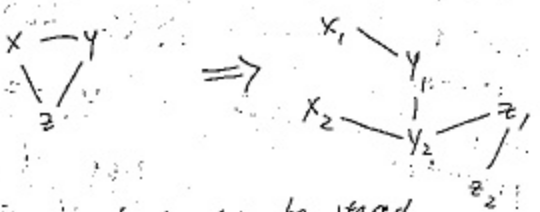
$\Rightarrow$

```
if (x0<0) {
    y1 := y0 - x0;
} else {
    y2 := y0 + x0;
}
y3 = φ(y1, y2);
return y3
```

Well-formedness condition: vids must be defined before use.

Impact of SSA

① Register allocation: SSA form reduces register pressure.
(Since when constructing interference graph, we have a lot more nodes from subscripted variables, but w.o. too many edges... i.e. interference graph is sparse.)

ex 

In addition, each x_i can be stored in a different memory location.

★ Interference graphs for SSA are <u>chordal</u>
(every cycle contains a chord)
- Chordal graphs can be optimally colored in polynomial-time

★ <u>BUT</u> optimal translation out of SSA is intractible

- Yet another advantage: Make optimizations more powerful

Ex: Dead Assignment elimination

Eliminate assignment instructions that are never used:

while some %x has no uses do
  remove definition of %x from CFG

☆ SSA form → can eliminate more assignments

ex:
```
x = 0
x = 1
return 2*x
```
→ SSA →
```
x0 = 0      (can eliminate)
x1 = 1
return 2*x1
```

Last advantage (of SSA): For dataflow analysis, In particular const propagation.

Goal of Const prop: determine a Const Env at each instruction. Solve this via a constraint system.

Computational time of Const prop (dense):

Memory: $O(|N| \times |Var|)$

$O(1)$ per node × each Const env has size $O(Var)$

Runtime: $O(|N| \times |Var|)$ — height of abstract domain: 3 x |Var|

- Processing each node is $O(1)$ × each node processed $O(Var)$ times.

Q: Can we do better?

Sparse const. propagation using SSA

Idea: SSA connects variable definitions directly to their uses

* Don't store value of every variable at every program point.

Idea: One "global" const. env. for everything rather than const. env. for each var.

* Don't propagate changes thru irrelevant blocks.

Think of SSA as graph: edges = control flow other than data flow

Recall: define rhs(x) = rhs of unique assignment to x

succ(x) = {y : rhs(y) = x}

* Local specification for const prop:

scp is smallest function mapping Var → {⊤, ⊥} ∪ ℤ

① such that if b contains no assignments to x then scp(x) = ⊤

② if instruction %x = e, scp(x) = eval(e, scp)

↑ evaluates const expression rhs (e.g. do a join of)

Augment worklist for graph in SSA form:

$$scp(x) = \begin{cases} \perp & x \text{ has assignment} \\ \top & \text{otherwise} \end{cases}$$

Initially work := { x ∈ Uid ; x is defined }

eval is basically ⇐ Big difference: one const env for entire program.

& past operator: compute past env from pre env

either uid takes some value at all pts of program or it doesn't at any point in program

while work ≠ ∅ do

  Pick %x from work

  if rhs(x) = φ(y, z) then
    v = scp(y) ⊔ scp(z)
  else v = eval(rhs(x), scp)
  fi

Runtime of sparse const prop

memory: $O(|N|) = O(|Var|)$ ⇐ Due to variable store increase amount

time: $O(|N|) = O(|Var|)$

A: Get a asymptotic improvement.

Computing SSA form

high-level algorithm:

- replace $x = e$ w/ $x_i = e$ for unique i
- replace each use of y w/ appropriate y_i where y_i is appropriate reaching definition
- If multiple reaches then use ϕ-instruction to merge.

Ex

```
y = 0
while x >= 0 {
    x = x - 1
    y = y + x
}
return y
```

⇒

```
y0 = 0
while true {
    x2 = φ(x0, x1)
    y2 = φ(y0, y1)
    if (x2 < 0) break;
    x1 = x2 - 1
    y1 = y2 + x1
}
return y2
```

Question where to insert φ statements?

easy solution place eagerly. place φ statement for each variable location at each join point.

- A join point is a node in CFG w/ more than one predecessor.

Better solution Place ϕ statement for variable x at location n exactly when the path convergence criterion holds.

PC criterion ∃ pair of nonempty paths P_1, P_2 ending at n s.t.
- start node of both P_1 and P_2 defines x
- Only node shared by P_1, P_2 is n.

→ Every point satisfying this criterion is a join point.

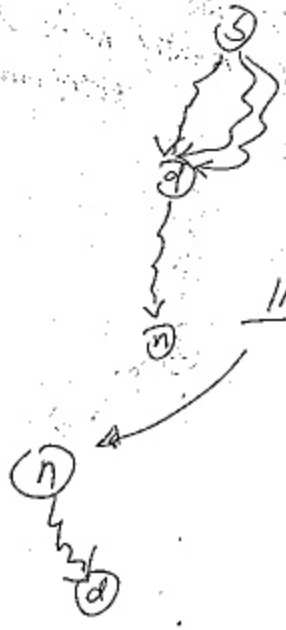
path convergence criterion can be implemented using iterated dominance frontiers.

☆ Dominance.

Let $G = (N, E, s)$ be a control flow graph.

We say a node $d \in N$ dominates $n \in N$ if every path from s to n must pass thru d.

☆ d "must happen" before n



- $n \in N$ dominates itself

Strict dominance

d s. dominates n if $d \neq n$

Immediate dominance

d imm. dominates n if
- d strictly dominates n
- but does not strictly dominate any strict dominator of n.

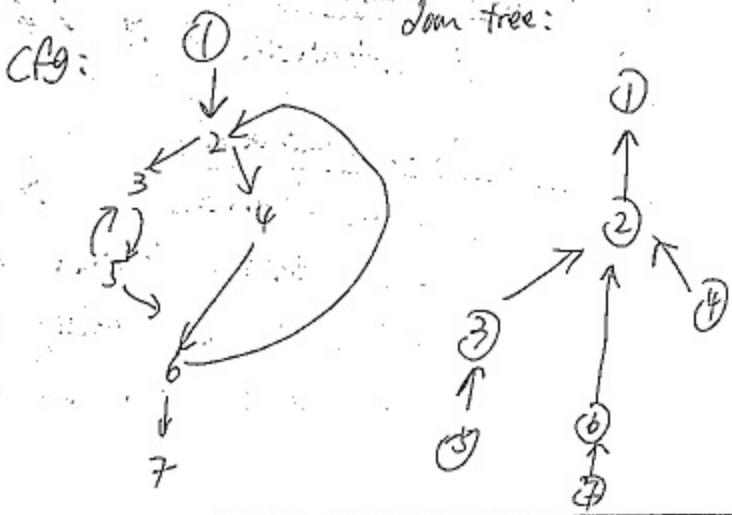
Observe dominance is partial order on N

- Every node dominates itself (reflexive)
- If n_1 dom n_2 and n_2 dom n_3 then n_1 dom n_3 (transitive)
- If n_1 dom n_2 $\wedge$ n_2 dom n_1 then $n_1 = n_2$ (Antisymmetric)

  Pf $s \rightsquigarrow n_1$ no longer than $s \rightsquigarrow n_2$

  So $n_1 = n_2$ (otherwise $s \rightsquigarrow n_2$ is a shorter path and n_1 dom n_2).

The dominator tree (the Hasse diagram of the dom partial order) ★entry node dom everything



Example of dominance analysis: See course website.

Dominance and SSA

Reformulate w.f. criteria in terms of dominance:

- ☆ if x is the i-th argument to ϕ function in a block n, then the definition of x must dominate the i-th predecessor of n.
- ☆ if x in non-ϕ block then def of x dominates n.

(i.e. x must be defined at n if n uses x).

Dominator Analysis

Let $G = (N, E, s)$ be a cfg.

Def Let dom be a function mapping node $n \in N \rightarrow$ $dom(n) \subseteq N$ a subset of nodes that dominate it.

Local specification

dom is "largest" (least in superset order) function such that

1. $dom(s) = \{s\}$
2. $\forall (p \rightarrow n) \in E$, $dom(n) \supseteq \{n\} \cup dom(p)$

* Universal gen/kill analysis.

at n, gen(n) generates n itself
kill(n) kills nothing (intersection removes any non dom nodes).

- dominance can be solved as dataflow analysis

In practice: Lengauer-Tarjan for near-linear time performance.

Dominance frontier

of a node n is $\{m :$ n dominates a predecessor of m, but not strictly dominate m itself$\}$.

$DF(n) = \{m : (\exists p \in pred(m), n \in dom(p)) \wedge (m = n \vee n \notin dom(m))\}$

↑ Join point

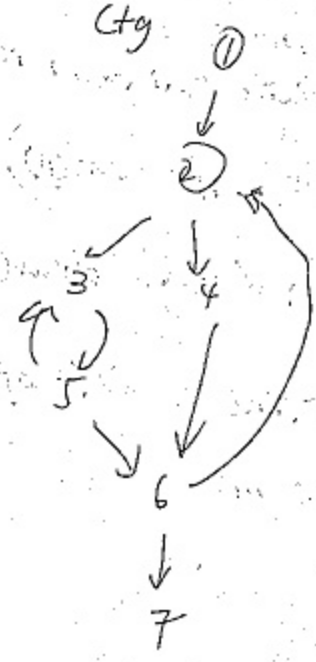
- Whenever node n contains a def of some var x then any m

Example

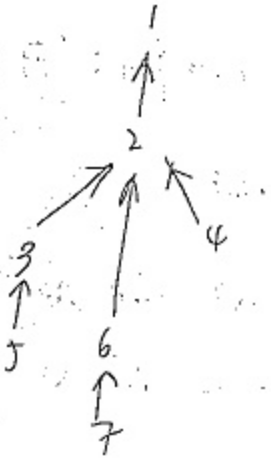
$DF(1) = \emptyset$

$DF(2) =$ ← dominates everything except 0.

Ctg



Dominator tree



$DF(3) = \{3, 6\}$ (3 dom 5

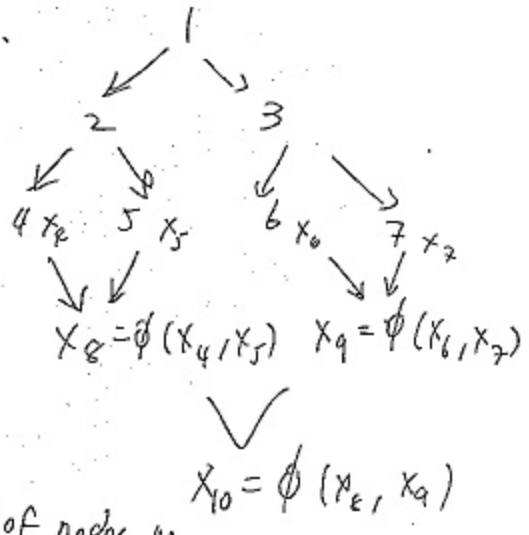
Dominance frontier not enough

Idea:

if we have def of x at node n, then we insert ϕ-node for x at every $m \in DF(n)$.

But not sufficient...

Insertion of ϕ-nodes create new conflicts



Iterated DF: for set of nodes M

$$DF(M) = \bigcup_{m \in M} DF(M)$$

Iterated dom frontier $IDF(M) = \bigcup_i IDF_i(M)$ where

$$IDF_0(M) = DF(M)$$
$$IDF_{i+1}(M) = IDF_i(M) \cup IDF(IDF_i(M))$$

Transforming OUT of SSA

- ϕ statement not executable so must be removed, in order to generate code.
- For each ϕ-statement:

  $x = \phi(x_1 \dots x_k)$ in block n, n must have exactly k predecessors $p_1 \dots p_k$
- Insert a new block along each edge $p_i \longrightarrow n$ which executes $\%x = \%x_i$

(program no longer satisfies SSA property!)

- Use graph coalescing register allocator; it is often possible to eliminate these moves.