CS 4240: Compilers

Lecture 8: Constant Propagation, Unreachable Code Elimination

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

February 4, 2019

ANNOUNCEMENTS & REMINDERS

- Project 1 is due by 11:59pm on Wednesday, 2/13/19 on Canvas
  - Must be submitted as zip file including instructions on how to build and run your project
  - 100 points total, with an extra credit option for 15 points
    - Extra credit relates to use of copy propagation
  - 5% of course grade
- Feb 6th lecture will be an in-class help session for Project 1ed by the TAs
- Feb 11th lecture will be a guest lecture by Prof. Tom Conte on the MIPS processor architecture, the target for Project 2
- From Feb 13th onwards, we will focus on Back-end topics (Chapters 11-13) that are relevant to Project 2
- MIDTERM EXAM: Wednesday, March 13, 4:30pm - 5:45pm
- FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm

Worksheet – 7 Solution

From lecture given on 1/30/2019

<u>Q1.</u> Identify which of the uses in the four print statements (S1, S2, S3, S4) can be identified as constant via constant propagation, using only reaching definitions analysis.

<u>Q2.</u> Identify any additional constants that you can identify in the print statements using insights beyond the use of reaching definitions.

```
x =10;
y = 2;
if (y > 1) y = 3;
print(x);            // S1
do {
    z = 4;
    x = x-1;
} while (x > 0);
print(z);            // S2
print(y);            // S3
print(x);            // S4
```

Q1

- **def1 (x=10)** is the only def of **x** to reach **S1**

→ **x** can be identified as constant, 10, in **S1**
(model uses of unitialized variables by adding a dummy def at start)

- **def4 (z=4)** is the only def of **z** to reach **S2**

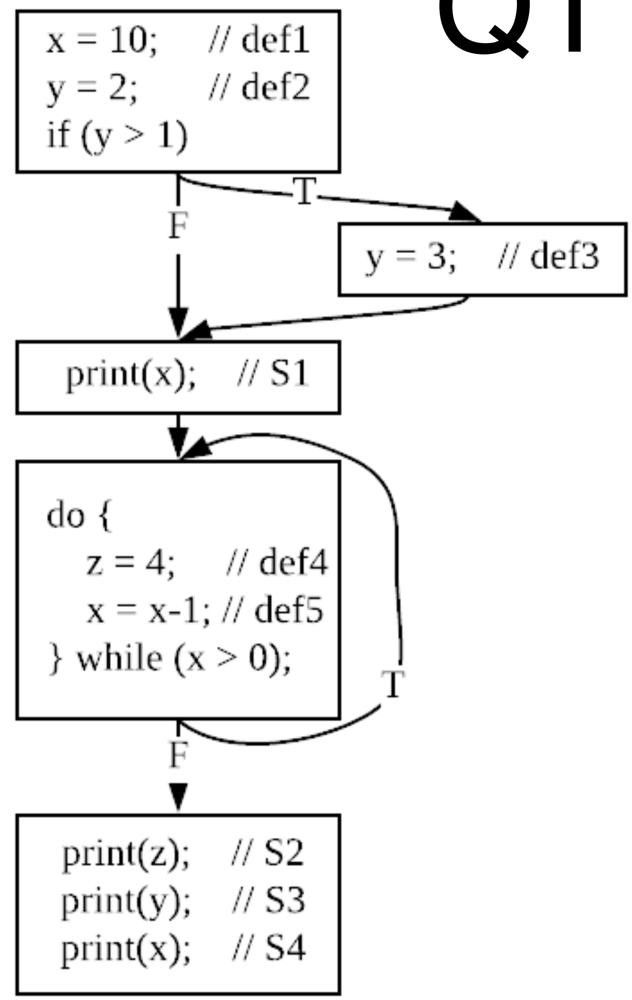
→ **z** can be identified as constant, 4, in **S2**

- Both **def2 (y=2)** and **def3 (y=3)** reach S3

→ we cannot conclude that y is constant in S3 by just using reaching definitions

- **def5** is the only def to reach **S5**, but its rval is not constant.

<u>**Uses in S1 & S2 can be identified as constant,**</u>
using only reaching definitions analysis.



Q2

- **def2** reaches the if-condition expression at line 3.
- By propagating **def2** to **the if-condition**, we can conclude that the if-condition always evaluates to TRUE, thereby ensuring that **def3 (y=3)** is the only def to reach S3
  → **we can conclude that y=3 at S3 by removing unreachable control flow edges**
- Since x starts with a value > 0, and is decremented by 1 in each iteration of the **do-while loop**

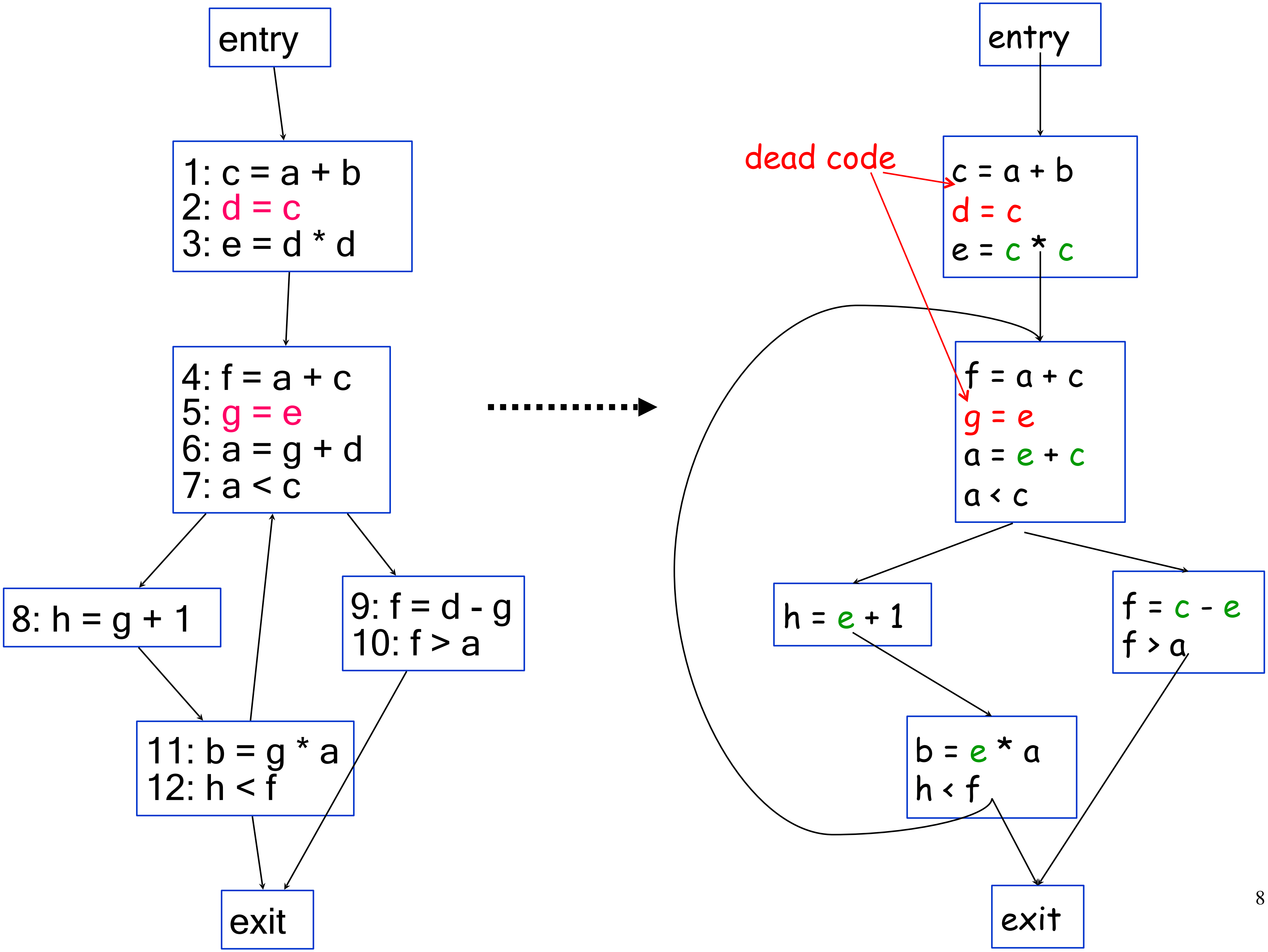
→ **we can conclude that x=0 when the loop exits, and that S4 will print x=0**
(This analysis is beyond the scope of the data flow analyses that we will learn in this course)

```
1   x = 10;             // def1
2   y = 2;              // def2
3   if (y > 1) y = 3;   // def3
4   print(x);           // S1
5   do {
6       z = 4;          // def4
7       x = x-1;        // def5
8   } while (x > 0);
9   print(z);           // S2
10  print(y);           // S3
11  print(x);           // S4
```

Recap of Data Flow Analyses that we've studied so far

- Reaching definitions analysis
  - $OUT[S] = GEN[S] \cup (IN[S] - KILL[S])$
  - $IN[S] = \bigcup_{p \in predecessors} OUT[p]$
- Computation of dominator sets
  - $dom(n) = \cap \{ dom(m) \mid m \in pred(n) \} \cup \{n\}$
- Available expressions analysis
  - $Avail(b) = \bigcap_{x \in pred(b)} (DEExpr(x) \cup (Avail(x) - ExprKill(x))$
- Copy propagation
  - $OUT[S] = GEN[S] \cup (IN[S] - KILL[S])$
  - $IN[S] = \bigcap_{p \in pred(S)} OUT[p]$

Copy Propagation reveals opportunities for Dead Code Elimination



Other related topics we have learned so far

- Redundancy elimination driven by
  - Local Value Numbering (LVN)
  - Superlocal Value Numbering (SVN)
  - Dominator Value Numbering Technique (DVNT)
  - Available Expression Analysis
- Static Single Assignment (SSA) form
  - Increases efficiency of data flow analyses

Revisiting Constant Propagation

» Goal: Produce an algorithm that will propagate all constants in a procedure, replacing constant expressions with the result of evaluating the expression at compile time

» We can approach this problem with a few different strategies based on what we've learned so far

1. Value Numbering — evaluate constant expressions in hashmap, as part of value numbering process
2. Single definition — if a use of variable x is reached by a single definition which has the form, "x = <constant>", then the use can be directly replaced by the constant
3. Copy propagation — restrict attention to copy statements of the form, "x = <constant>"
4. Reaching definitions — model the rval of each def as a set of values, and take the union of all sets reaching a use

   ==> this is the approach that we will study in detail today

Constant Propagation over DEF-USE Chains

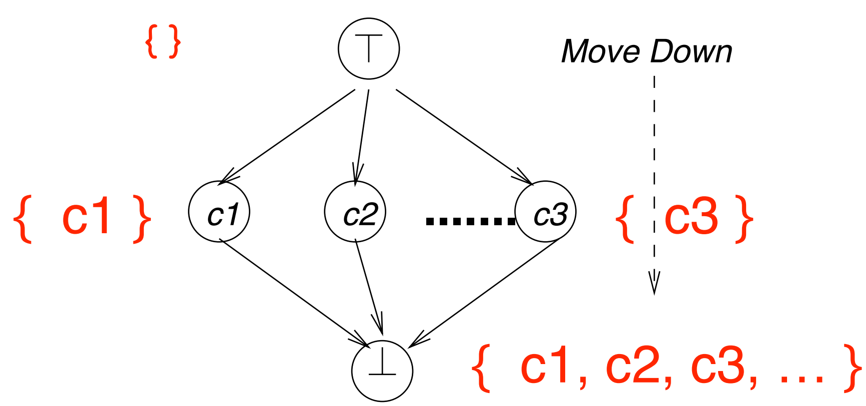
// Initialization
Compute Def-Use chains
Worklist $\leftarrow$ Ø
For i $\leftarrow$ 1 to number of operations
    if in_1 of operation i is a constant c_i
        then Value(in_1,i) $\leftarrow$ c_i
        else Value(in_1,i) $\leftarrow$ Top
    if in_2 of operation i is a constant c_j
        then Value(in_2,i) $\leftarrow$ c_j
        else Value(in_2,i) $\leftarrow$ Top
    if (Value(in_1,i) is a constant &
        Value(in_2,i) is a constant)
        then Value(out,i) $\leftarrow$ evaluate op i
            Worklist $\leftarrow$ Worklist $\cup$ {i }

// Iteration using meet operator, $\wedge$, in a
// semi lattice
while (Worklist $\neq$ Ø)
    remove a definition i from WorkList
    for each j $\in$ USES(out,i)
        set x so that <u>out</u> of <u>i</u> is <u>in_x</u> of j
        Value(in_x,j) $\leftarrow$ Value(out,i)
        for each k $\in$ DEFS(in_x,j), k $\neq$ i
            Value(in_x,j) $\leftarrow$ Value(in_x,j)
                $\wedge$ Value(out,,k)
        if (Value(in_1,j) is a constant &
            Value(in_2,j) is a constant)
            then Value(out,j) $\leftarrow$ evaluate op j
                Worklist $\leftarrow$ Worklist $\cup$ {j }
            else if (Value(in_1,j) is $\perp$ or
                Value(in_2,j) is $\perp$)
            then Value(out,j) $\leftarrow$ $\perp$

The Lattice Structure

A lattice value denotes a set of possible constant values.
We are interested in the case when the set is a singleton.

- We have a unique symbol $\perp$ representing the fact that a contant value *cannot* be guaranteed
- Several (potentially unbounded number of) constant symbols $\mathcal{C}_i$ that denote the space of all possible constants
- These constants are dominated by a unique $\top$ symbol that represents that fact that the corresponding variable/expression to which it is assigned *may* potentially be reducible to a constant



The Intuition

- We start out with all the nodes being assigned $\top$
- The idea is to move down the lattice towards $\perp$ and see whether the analysis stabilizes at a constant $\mathcal{C}_i$ in between or whether it reaches $\perp$
- The rules for combination are as follows where *Anysymbol* denotes $\top, \perp$ or one of the constants $\mathcal{C}_i$
  1. $Anysymbol \sqcap \top = Anysymbol$
  2. $Anysymbol \sqcap \perp = \perp$
  3. $\mathcal{C}_i \sqcap \mathcal{C}_i = \mathcal{C}_i$
  4. $\mathcal{C}_i \sqcap \mathcal{C}_j = \perp, i \neq j$

The meet operator, $\sqcap$ or $\wedge$

corresponds to the set union

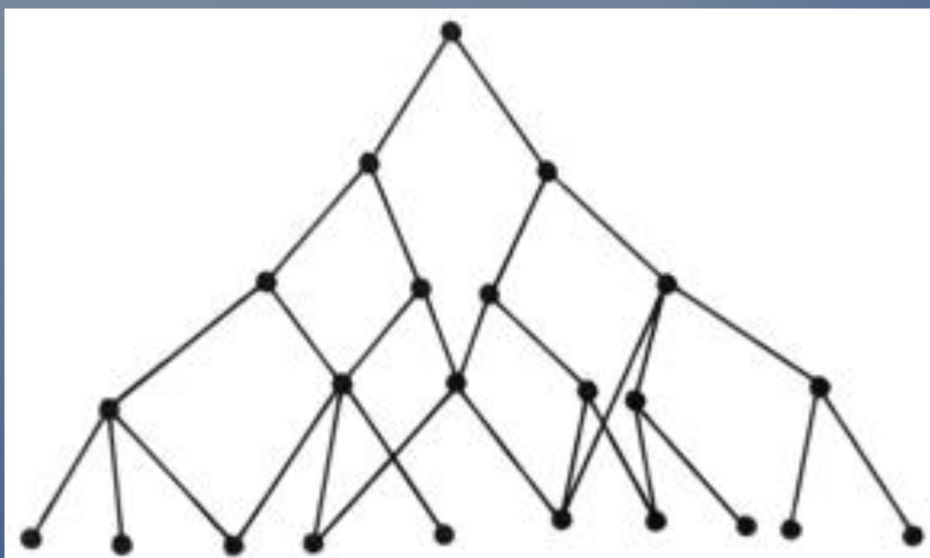
operation on sets denoted

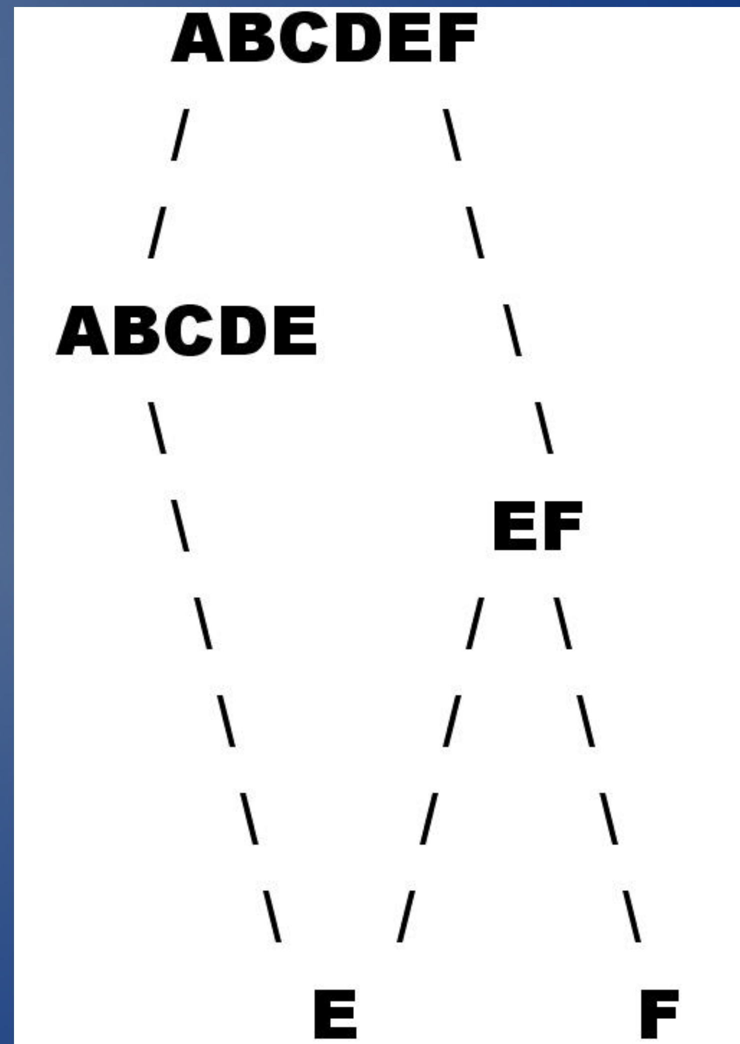
by the lattice values. It is performed at any point with two or more reaching definitions for the same variable (a phi function in SSA form!)

Background: What is a Semilattice?

- Some domain of values . . .
- Example: {a, b, c}
- . . . a meet operator: $\wedge$ . . .
- . . . and a top element: T
- The top element is defined as T $\wedge$ a = a, for all elements a in the domain.
- Optionally, there may be a bottom element: $\perp$, where $\perp \wedge a = \perp$, for all elements a in the domain.

What do they look like?





Partial Ordering

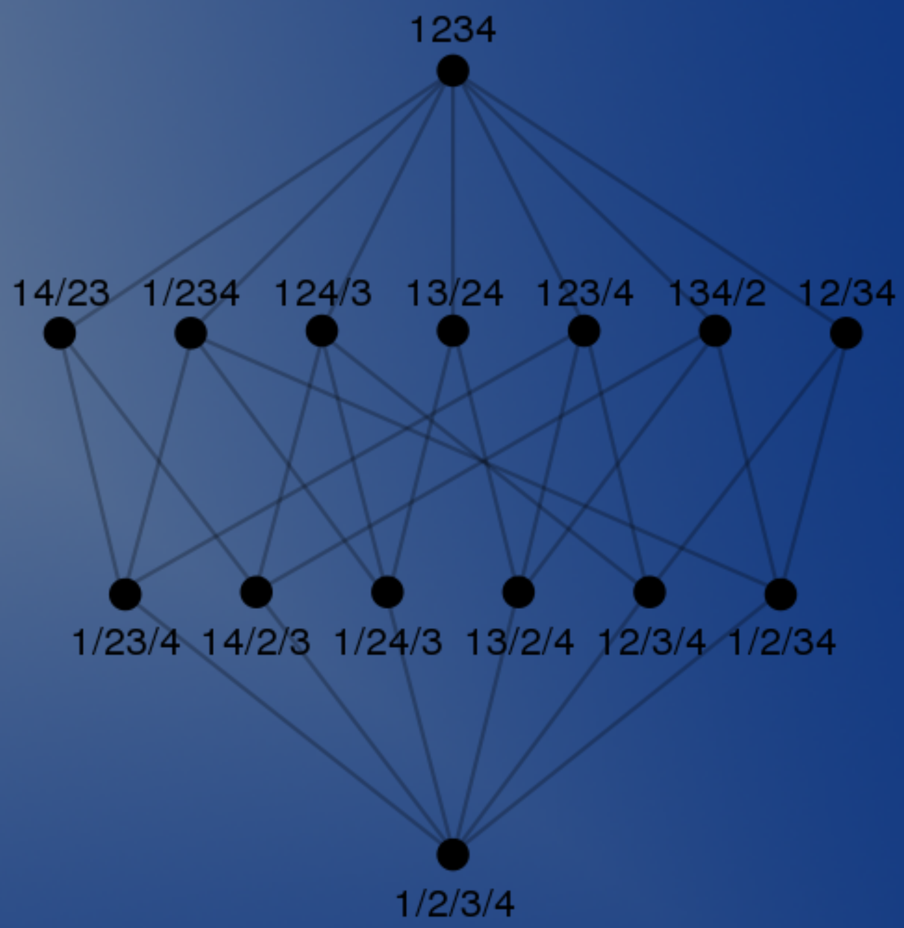
- The meet operator causes all semilattices to be partially ordered.
- If an ordered set is one where the operation $a < b$ applies to any two elements, a partially ordered one is where the operation $a \leq b$ applies.
- $\leq$ does not necessarily mean less than or equal to! It can be any ordering.
- $a \leq b$ can mean:
  - a is smaller than or equal to b
  - a is a refinement of or the same as b
  - a has the same or more (!) elements than b
  - . . . as long as the relationship is always the same for all elements

Meet the Meet Operator

- The meet operator can be any binary operator that has the following attributes for all elements in the domain:
  - $a \wedge a = a$
  - $a \wedge b = b \wedge a$
  - $a \wedge (b \wedge c) = (a \wedge b) \wedge c$
- Good candidates for the meet operator are union U and intersection ∩
- Remember $T \wedge a = a$
- If T is empty, ∧ is union
- If T is all the elements in the domain, ∧ is intersection

Semilattice vs. Lattice

- Lattices have all the qualities of semilattices.
- They have two binary operators: a join operator in addition to the meet operator.
- The join operator is usually denoted as: V
- The join operator is complementary to meet: if meet means union, join means intersection, and vice versa.



Sparse Constant Propagation over SSA Form

∀ expression, e
  Value(e) ← TOP if its value is unknown; c_i if its value is known; BOT if its value is known to vary
WorkList ← Ø

∀ SSA edge s = <d,u>
  if Value(d) ≠ TOP then
    add s to WorkList

while (WorkList ≠ Ø)
  remove s = <d,u> from WorkList
  let o be the operation that uses u
  if Value(o) ≠ BOT then
    t ← result of evaluating o
    if t ≠ Value(o) then
      Value(o) ← t
      ∀ SSA edge <o,x>
        add <o,x> to WorkList

i.e., o is "a ← b op u" or "a ← u op b"

The Algorithm

Evaluating a Ø-function:

Ø($x_1, x_2, x_3, \ldots x_n$) is

Value(x_1) ∧ Value(x_2) ∧ Value(x_3) ∧ … ∧ Value(x_n)

where

TOP ∧ x = x ∀ x

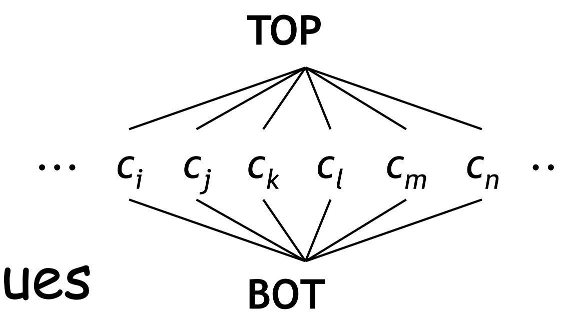
$c_i \wedge c_j = c_i$ if $c_i = c_j$

$c_i \wedge c_j$ = BOT if $c_i \neq c_j$

BOT ∧ x = BOT ∀ x

Sparse Constant Propagation over SSA Form

How long does this algorithm take to halt?



- Initialization is two passes
  - |ops| + |edges| (|edges| ≤ 2 x |ops|)
- In propagation, Value(x) can take on 3 values
  - **TOP**, c_i , **BOT**
  - Each edge can be on the WorkList twice
  - 2 x |args| = 4 x |ops| evaluations, 2 x |ops| WorkList pushes & pops

This algorithm is much simpler than the original algorithm based on def-use chains

Initialization to TOP?

The Sparse Conditional Constant (SCP) algorithm initializes everything to TOP

What does that accomplish?

$x_0 \leftarrow 12$

while (…)

$x_2 \leftarrow ø(x_0, x_4)$

…

$y \leftarrow x_2 * 17$

$z \leftarrow x_2$

$x_3 \leftarrow …$

$… \leftarrow x_3$

$x_4 \leftarrow z$

It is clear, after some thought, that x_2 is always 12 at the definition of y

Can SCP discover x_2's value?

- Depends on the initialization of Value(x_2)
- We obtain different results when we start with **TOP** & **BOT**

M. Wegman and F.K. Zadeck, "Constant propagation with conditional branches," **ACM TOPLAS** 13(2), 1991, pages 181-210. [pg 347 in textbook]

Initialization to TOP?

The Sparse Conditional Constant algorithm initializes everything to TOP

What does that accomplish?

12	$x_0 \leftarrow 12$
	while (…)
$\perp$	$x_2 \leftarrow ø(x_0, x_4)$
	…
$\perp$	$y \leftarrow x_2 * 17$
$\perp$	$z \leftarrow x_2$
?	$x_3 \leftarrow$ …
?	… $\leftarrow x_3$
$\perp$	$x_4 \leftarrow z$

Can SCP discover x_2's value?

- Depends on the initialization of Value(x_2)
- We obtain different results when we start with **TOP** & **BOT**

Initialization with BOT

- **BOT** from x_4 lowers 12 from x_0 to **BOT** in x_2
- **BOT** becomes flows around the loop and confirms the value

We call this pessimistic

M. Wegman and F.K. Zadeck, "Constant propagation with conditional branches," **ACM TOPLAS** 13(2), 1991, pages 181-210. [347 in EaC2e]

Initialization to TOP?

The Sparse Conditional Constant algorithm initializes everything to TOP

What does that accomplish?

12	$x_0 \leftarrow 12$
	while (…)
$\top$	$\quad x_2 \leftarrow ø(x_0, x_4)$
	$\quad$ …
$\top$	$\quad y \leftarrow x_2 * 17$
$\top$	$\quad z \leftarrow x_2$
?	$\quad x_3 \leftarrow$ …
?	$\quad$ … $\leftarrow x_3$
$\top$	$\quad x_4 \leftarrow z$

Can SCP discover x_2's value?

- Depends on the initialization of Value(x_2)
- We obtain different results when we start with **TOP** & **BOT**

Initialization with TOP

- **TOP** $\wedge$ 12 is 12, so x_2 gets 12
- 12 flows from x_2 to z to x_4, and then back into the ø-function
- **TOP**, in essence, confirms the value of 12 around the loop

We call this optimistic initialization

M. Wegman and F.K. Zadeck, “Constant propagation with conditional branches,” **ACM TOPLAS** 13(2), 1991, pages 181-210. [347 in EaC2e]

Initialization to TOP?

The Sparse Conditional Constant algorithm initializes everything to TOP

What does that accomplish?

```
12   x0 ← 12
     while ( … )
⊤        x2 ← ø(x0 , x4)
         …
⊤        y ← x2 * 17
⊤        z ← x2
?        x3 ← …
?        … ← x3
⊤        x4 ← z
```

Can SCP discover x_2's value?

- Depends on the initialization of Value(x_2)
- We obtain different results when we start with **TOP** & **BOT**

Optimism versus Pessimism

- In general, optimism helps values flow into loops
- No real data on how often it matters, but it is no more costly

M. Wegman and F.K. Zadeck, "Constant propagation with conditional branches," **ACM TOPLAS** 13(2), 1991, pages 181-210. [347 in EaC2e]

Sparse Constant Propagation

What happens when SCP propagates a value into a branch?

- **TOP** ⇒ we gain no knowledge
- **BOT** ⇒ either path can execute
- **TRUE** or **FALSE** ⇒ only one path can execute

But, the algorithm does not use this knowledge …

Using this observation, we can add an element of refining feasible paths to the algorithm that will take it beyond the standard limits of **DFA**

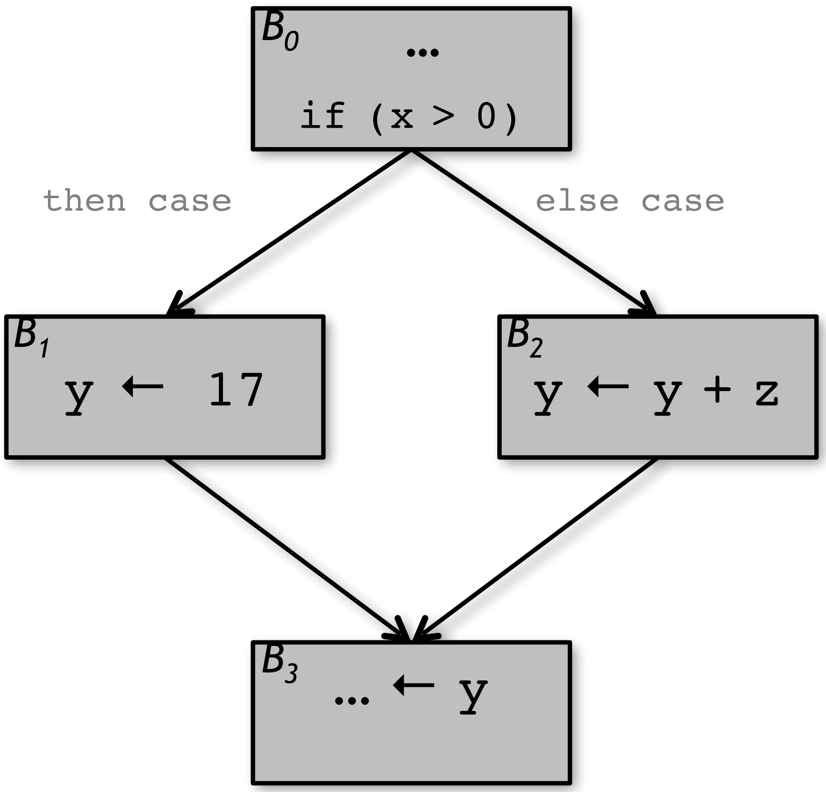
→ Until a block can execute, treat it as unreachable

→ Optimistic initializations allow analysis to proceed with unevaluated blocks

Result is an analysis that can use *limited symbolic evaluation* to combine constant propagation with unreachable code elimination

You can find the algorithm in M. Wegman and F.K. Zadeck, "Constant propagation with conditional branches," **ACM TOPLAS** 13(2), 1991, pages 181-210. [347 in EaC2e]. See also § 10.7.1 in **EaC2e**

Sparse Conditional Constant Propagation



Classic **DFA** assumes that all paths can be taken at runtime, including (B_0, B_2, B_3)

Can use constant-valued control predicates to refine the CFG

- If compiler knows the value of `x`, it can eliminate either the then or the else case
  - (x > 0) ⇒ y is 17 in B_3
  - (x > 0) ⇒ B_2 is unreachable
- This approach combines constant propagation with **CFG** reachability analysis to produce better results in each
- Example of Click's notion of *"combining optimizations"*
  - Predated & motivated Click

Sparse Conditional Constant Propagation

SSAWorkList ← Ø
CFGWorkList ← n_0

∀ block b
  clear b's mark
  ∀ expression e in b
    Value(e) ← TOP

Initialization Step

To evaluate a branch
  if arg is BOT then
    put both targets on CFGWorklist
  else if arg is TRUE then
    put TRUE target on CFGWorkList
  else if arg is FALSE then
    put FALSE target on CFGWorkList

To evaluate a jump
  place its target on CFGWorkList

while ((CFGWorkList ∪ SSAWorkList) ≠ Ø)
  while(CFGWorkList ≠ Ø)
    remove b from CFGWorkList
    mark b
    evaluate each Ø-function in b
    evaluate each op in b, in order

  while(SSAWorkList ≠ Ø)
    remove s = <u,v> from WorkList
    let o be the operation that contains v
    t ← result of evaluating o
    if t ≠ Value(o) then
      Value(o) ← t
      ∀ SSA edge <o,x>
        if x is marked, then
          add <o,x> to WorkList

Propagation Step

Proliferation of Data-flow Problems

If we continue in this fashion,

- We will need a huge set of data-flow problems
- Each will have slightly different initial information
- This seems like a messy way to go …

Desiderata

- Solve one data-flow problem
- Use it for <u>all</u> transformations

To help address this issue, researchers invented def-use chains, which led, eventually, to SSA