Generalizing Data-flow Analysis

Last Time

- Introduction to data-flow analysis

Today

- Other examples of data-flow analysis
- Abstracting data-flow analysis
  - What's common among these different analyses?

Time Complexity

Consider a program of size N

- Has N nodes in the flow graph and at most N variables
- Each live-in or live-out set has at most N elements
- Each set-union operation takes O(N) time
- The **for** loop body
  - constant # of set operations per node
  - O(N) nodes ⇒ $O(N^2)$ time for the loop
- Each iteration of the **repeat** loop can only make the set larger
- Each set can contain at most N variables ⇒ $2N^2$ iterations

Worst case: $O(N^4)$

Typical case: 2 to 3 iterations with good ordering & sparse sets
⇒ O(N) to $O(N^2)$

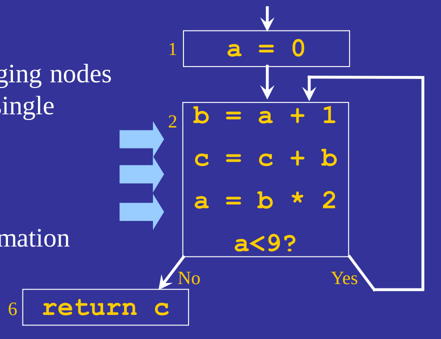
More Performance Considerations

Basic blocks

- Decrease the size of the CFG by merging nodes that have a single predecessor and a single successor into **basic blocks**

One variable at a time

- Instead of computing data-flow information for all variables at once using sets, compute a (simplified) analysis for each variable separately

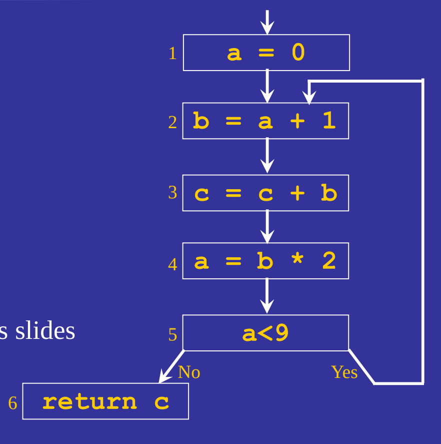


Representation of sets

- For dense sets, use a bit vector representation
- For sparse sets, use a sorted list (*e.g.*, linked list)

Conservative Approximation

			X	
node #	use	def	in	out
1		a	c	ac
2	a	b	ac	bc
3	bc	c	bc	bc
4	b	a	bc	ac
5	a		ac	ac
6	c		c	

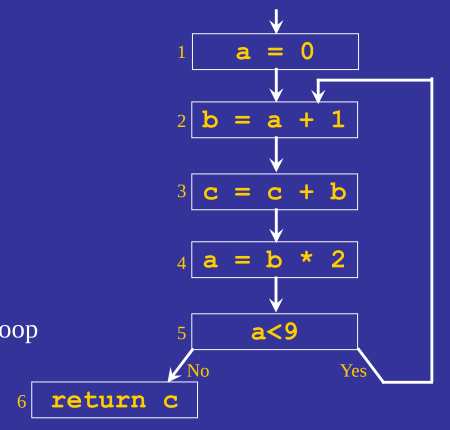


Solution X

- Our solution as computed on previous slides

Conservative Approximation (cont)

node #	use	def	X in	X out	Y in	Y out
1		a	c	ac	cd	acd
2	a	b	ac	bc	acd	bcd
3	bc	c	bc	bc	bcd	bcd
4	b	a	bc	ac	bcd	acd
5	a		ac	ac	acd	acd
6	c		c		c	



Solution Y

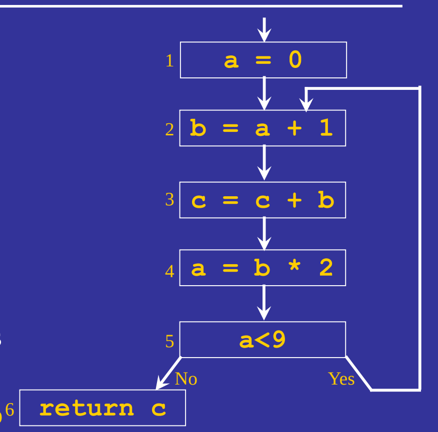
- Carries variable d uselessly around loop
- Does Y solve the equations? Yes
- Is d live? No
- Does Y lead to a correct program? Yes

  Imprecise conservative solutions ⇒ sub-optimal but correct programs

February 2, 2015 Data-flow Analysis 5

Conservative Approximation (cont)

node #	use	def	X in	X out	Y in	Y out	Z in	Z out
1		a	c	ac	cd	acd	c	ac
2	a	b	ac	bc	acd	bcd	ac	b□
3	bc	c	bc	bc	bcd	bcd	b□	b□
4	b	a	bc	ac	bcd	acd	b□	ac
5	a		ac	ac	acd	acd	ac	ac
6	c		c		c		c	



Solution Z

- Does not identify c as live in all cases
- Does Z solve the equations? No
- Does Z lead to a correct program? No

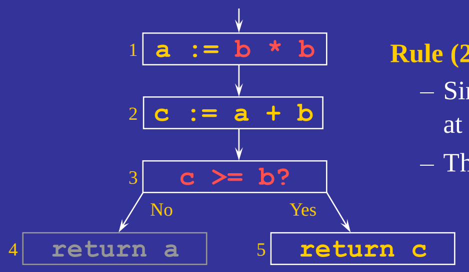
Non-conservative solutions ⇒ incorrect programs

February 2, 2015 Data-flow Analysis 6

The Need for Approximations

Static vs. Dynamic Liveness

– In the following graph, `b*b` is always non-negative, so `c >= b` is always true and `a`'s value will never be used after node 2



Rule (2) for computing liveness

– Since `a` is live-in at node 4, it is live-out at nodes 3 and 2

– This rule ignores actual control flow

No compiler can statically know all of a program's dynamic properties!

February 2, 2015 Data-flow Analysis 7

Generalizing Data-flow Analysis

Last Time

Introduction to data-flow analysis

Today

Other examples of data-flow analysis

Abstracting data-flow analysis

What's common among these different analyses?

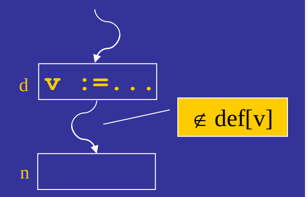
February 4, 2015 *Generalizing Data-flow Analysis* *8*

Calvin Lin
The University of Texas at Austin

Reaching Definitions

Definition

– A definition (statement) d of a variable **`v`** **reaches** node n if there is a path from d to n such that **`v`** is not redefined along that path

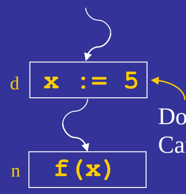


Uses of reaching definitions

– Build use/def chains

– Constant propagation

– Loop invariant code motion



Does this def of **`x`** reach n?
Can we replace n with **`f(5)`**?

```
1   a = . . .;
2   b = . . .;
3   for (. . .) {
4       x = a + b;
5       . . .
6   }
```

Reaching definitions of **`a`** and **`b`**

To determine whether it's legal to move statement 4 out of the loop, we need to ensure that there are no reaching definitions of **`a`** or **`b`** inside the loop

Computing Reaching Definitions

Assumption

– At most one definition per node

– We can refer to definitions by their node number

Gen[n]: Definitions that are generated by node n (at most one)

Kill[n]: Definitions that are killed by node n

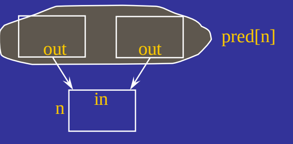
Defining Gen and Kill for various statement types

statement	Gen[s]	Kill[s]	statement	Gen[s]	Kill[s]
s: t = b op c	{s}	def[t] – {s}	s: goto L	{}	{}
s: t = M[b]	{s}	def[t] – {s}	s: L:	{}	{}
s: M[a] = b	{s}	{*} – {s}	s: f(a,…)	{}	{}
s: if a op b goto L	{}	{}	s: t=f(a, …)	{s}	def[t] – {s}

Data-flow Equations for Reaching Definitions

The In set

- A definition reaches the beginning of a node if it reaches the end of **any** of the predecessors of that node



The Out set

- A definition reaches the end of a node if (1) the node itself **generates** the definition **or** if (2) the definition reaches the beginning of the node and the node does **not kill** it

(1) (2)

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Recall Liveness Analysis

Data-flow equations for liveness

$$in[n] = \mathbf{use}[n] \cup (out[n] - \mathbf{def}[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

Liveness equations in terms of Gen and Kill

$$in[n] = \mathbf{gen}[n] \cup (out[n] - \mathbf{kill}[n])$$

$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

A use of a variable generates liveness
A def of a variable kills liveness

Can define almost any data-flow analysis in terms of Gen and Kill

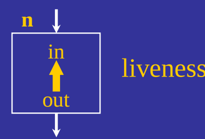
Gen: New information that's added at a node
Kill: Old information that's removed at a node

Direction of Flow

Backward data-flow analysis

– Information at a node is based on what happens later in the flow graph
  i.e., in[] is defined in terms of out[]

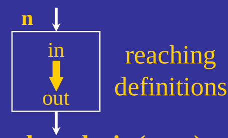
$$in[n] = gen[n] \cup (out[n] - kill[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$



Forward data-flow analysis

– Information at a node is based on what happens earlier in the flow graph
  i.e., out[] is defined in terms of in[]

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$
$$out[n] = gen[n] \cup (in[n] - kill[n])$$



Some problems need both forward and backward analysis (rare)

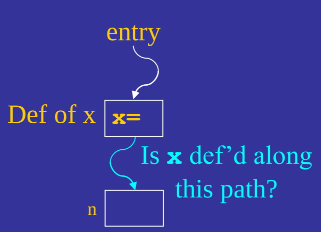
– *e.g.*, Partial redundancy elimination

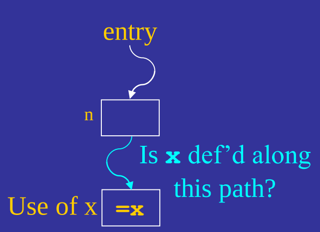
Data-flow Equations for Reaching Definitions

Symmetry between reaching definitions and liveness

– Swap in[] and out[] and swap the directions of the arcs

Reaching Definitions	Liveness
$in[n] = \bigcup_{p \in pred[n]} out[s]$	$out[n] = \bigcup_{s \in succ[n]} in[s]$
$out[n] = gen[n] \cup (in[n] - kill[n])$	$in[n] = gen[n] \cup (out[n] - kill[n])$





Liveness vs. Reaching Definitions

What are the key differences?

- Direction of flow
- Flow values
  - Liveness operates on variables
  - Reaching definitions operates on definitions (statements)

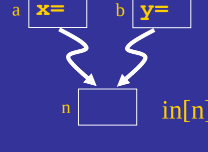
A Better Formulation of Reaching Definitions

Problem

- Reaching definitions gives you a set of definitions (nodes)
- Doesn't tell you what variable is defined
- Expensive to find definitions of variable v (Why?)

Solution

- Reformulate to include the variable
  e.g., Use a set of (var, def) pairs



in[n] = {(x,a),(y,b),...}

Merging Flow Values

Liveness and reading definitions

- Merge **flow values** via set union

Reaching Definitions

$in[n] = \bigcup_{p \in pred[n]} out[s]$

$out[n] = gen[n] \cup (in[n] - kill[n])$

Liveness

$out[n] = \bigcup_{s \in succ[n]} in[s]$

$in[n] = gen[n] \cup (out[n] - kill[n])$

Why Use Union?

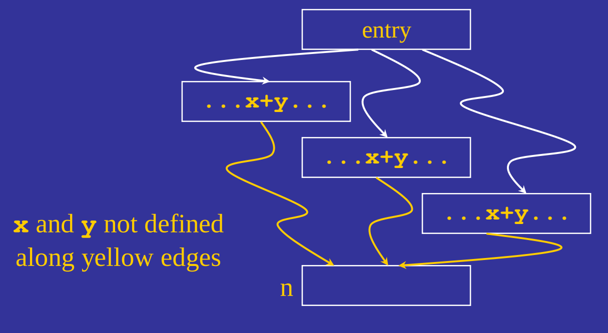
When might this be inappropriate?

February 4, 2015 Generalizing Data-flow Analysis 17

Available Expressions

Definition

- An expression, **`x+y`**, is **available** at node n if every path from the entry node to n evaluates **`x+y`**, and there are no definitions of **`x`** or **`y`** after the last evaluation



February 4, 2015 Generalizing Data-flow Analysis 18

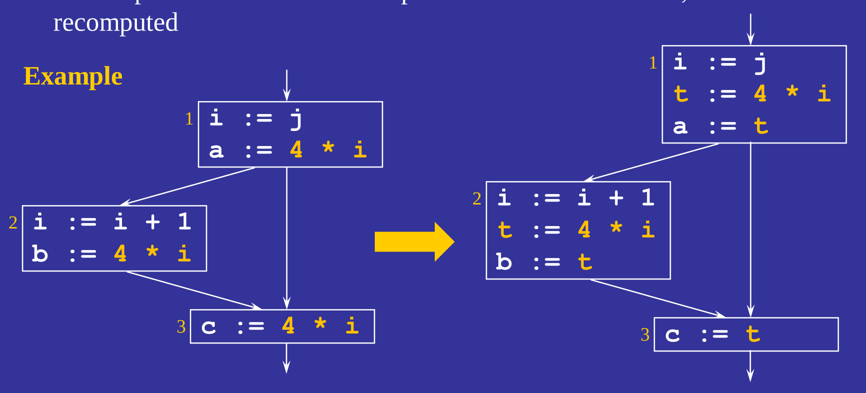
Available Expressions for CSE

How is this information useful?

Common Subexpression Elimination (CSE)

– If an expression is available at a point where it is evaluated, it need not be recomputed

Example



February 4, 2015 Generalizing Data-flow Analysis 19

Must vs. May Information

Must information

– Implies a guarantee

May information

– Identifies possibilities

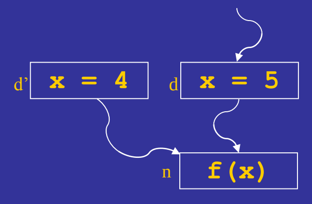
Liveness? Available expressions?

	May	Must
safe	overly large set	overly small set
desired information	small set	large set
Gen	add everything that might be true	add only facts that are guaranteed to be true
Kill	remove only facts that are guaranteed to be false	remove everything that might be false
merge	union	intersection
initial guess	empty set	universal set

February 4, 2015 Generalizing Data-flow Analysis 20

Reaching Definitions: Must or May Analysis?

Consider constant propagation



We need to know if d' might reach node n

Exercise: Define Available Expressions Analysis

Must or may Information?	Must
Direction?	Forward
Flow values?	Sets of expressions
Initial guess?	Universal set
Kill?	Set of expressions killed by statement s
Gen?	Set of expressions evaluated by s
Merge?	Intersection

Available Expressions (cont)

Data-Flow Equations

$\text{in}[n] = \bigcap_{p \in \text{pred}[n]} \text{out}[p]$

$\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$

Plug it in to our general DFA algorithm

for each node n
  in[n] = υ; out[n] = υ
repeat
  for each n
    in′[n] = in[n]
    out′[n] = out[n]
    $\text{in}[n] = \bigcap_{p \in \text{pred}[n]} \text{out}[p]$
    $\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$
until in′[n]=in[n] and out′[n]=out[n] for all n

Exercise: Reaching Constants

Goal
- Compute value of each variable at each program point (if possible)

Flow values
- Set of (variable,constant) pairs

Merge function
- Intersection

Gen and Kill
- Effect of node n `x = c`
  - kill[n] = {(x,d)| ∀d}
  - gen[n] = {(x,c)}
- Effect of node n `x = y + z`
  - kill[n] = {(x,c)| ∀c}
  - gen[n] = {(x,c) | c=valy+valz, (y, valy) ∈ in[n], (z, valz) ∈ in[n]}