Introduction to Data-flow Analysis

Last Time

- Control flow analysis
- BT discussion

Today

- Introduce iterative data-flow analysis
  - Liveness analysis
  - Introduce other useful concepts

Data-flow Analysis

Idea

- **Data-flow analysis** derives information about the **dynamic** behavior of a program by only examining the **static** code

Example

- How many registers do we need for the program on the right?
- Easy bound: the number of variables used (3)
- Better answer is found by considering the **dynamic** requirements of the program

```
     a := 0
L1:  b := a + 1
     c := c + b
     a := b * 2
     if a < 9 goto L1
     return c
```

Liveness Analysis

Definition

- A variable is **live** at a particular point in the program if its value at that point will be used in the future (**dead**, otherwise).
  ∴ To compute liveness at a given point, we need to look into the future

Motivation: Register Allocation

- A program contains an unbounded number of variables
- Must execute on a machine with a bounded number of registers
- Two variables can use the same register if they are never in use at the same time (*i.e,* never simultaneously live).
  ∴ Register allocation uses liveness information

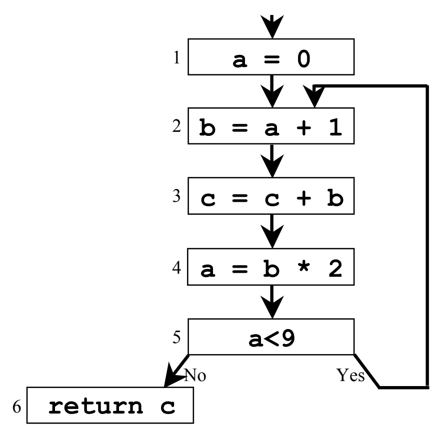
Control Flow Graphs (CFGs)

Simplification

- For now, we will use **CFG's** in which nodes represent program statements rather than basic blocks

Example

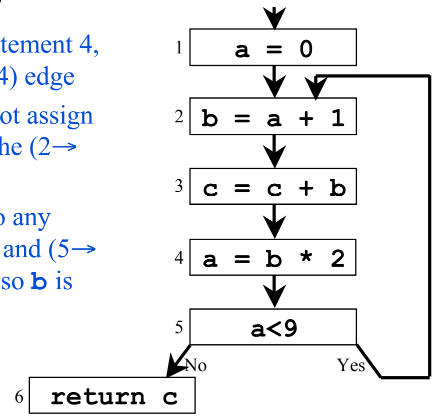
```
     a := 0
L1:  b := a + 1
     c := c + b
     a := b * 2
     if a < 9 goto L1
     return c
```



Liveness by Example

What is the live range of `b`?

- Variable **`b`** is read in statement 4, so **`b`** is live on the (3 → 4) edge
- Since statement 3 does not assign into **`b`**, **`b`** is also live on the (2→3) edge
- Statement 2 assigns **`b`**, so any value of **`b`** on the (1→2) and (5→2) edges are not needed, so **`b`** is dead along these edges



`b`’s live range is (2→3→4)

Liveness by Example (cont)

Live range of `a`

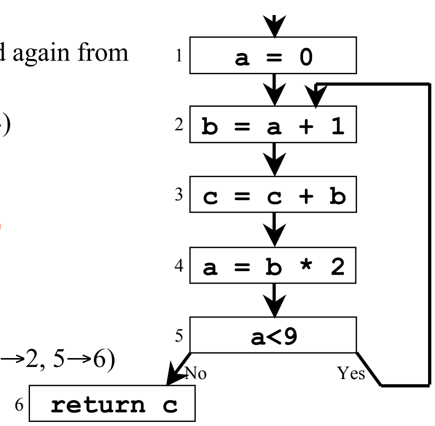
- **`a`** is live from (1→2) and again from (4→5→2)
- **`a`** is dead from (2→3→4)

Live range of `b`

- **`b`** is live from (2→3→4)

Live range of `c`

- **`c`** is live from (entry→1→2→3→4→5→2, 5→6)

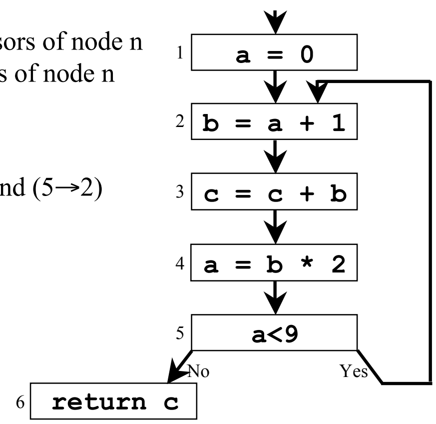


Variables **`a`** and **`b`** are never simultaneously live, so they can share a register

Terminology

Flow Graph Terms

- A CFG node has **out-edges** that lead to **successor** nodes and **in-edges** that come from **predecessor** nodes
- **pred[n]** is the set of all predecessors of node n
  succ[n] is the set of all successors of node n



Examples

- Out-edges of node 5: (5→6) and (5→2)
- succ[5] = {2,6}
- pred[5] = {4}
- pred[2] = {1,5}

Uses and Defs

Def (or definition)

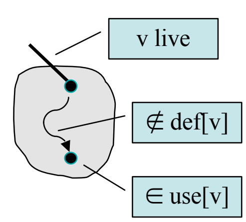
- An **assignment** of a value to a variable
- def[v] = set of CFG nodes that define variable v
- def[n] = set of variables that are defined at node n

`a = 0`

`a < 9?`

Use

- A **read** of a variable's value
- use[v] = set of CFG nodes that use variable v
- use[n] = set of variables that are used at node n



More precise definition of liveness

- A variable v is live on a CFG edge if
  (1) ∃ a directed path from that edge to a use of v (node in use[v]), and
  (2) that path does not go through any def of v (no nodes in def[v])

The Flow of Liveness

Data-flow

– Liveness of variables is a property that flows through the edges of the CFG

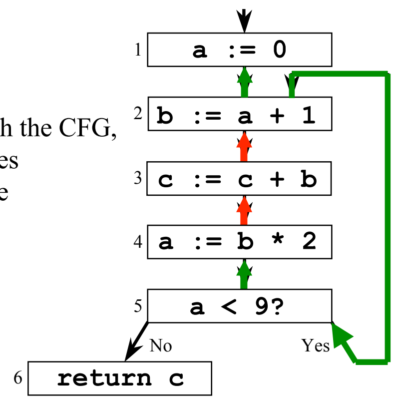
Direction of Flow

– Liveness flows **backwards** through the CFG, because the behavior at future nodes determines liveness at a given node

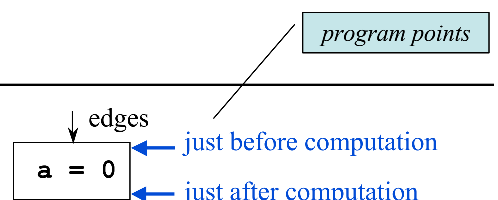
– Consider a

– Consider b

– Later, we'll see other properties that flow **forward**

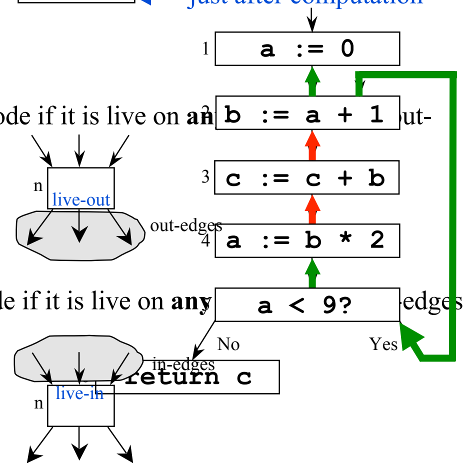


Liveness at Nodes



We have liveness on edges

– How do we talk about liveness at nodes?



Two More Definitions

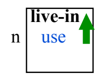
– A variable is **live-out** at a node if it is live on any out-edges

– A variable is **live-in** at a node if it is live on **any** in-edges

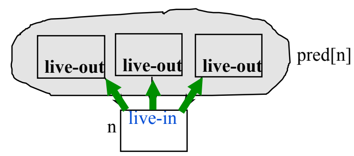
Computing Liveness

Rules for computing liveness

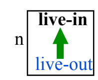
(1) Generate liveness:
If a variable is in use[n],
it is live-in at node n



(2) Push liveness across edges:
If a variable is live-in at a node n
then it is live-out at all nodes in pred[n]



(3) Push liveness across nodes:
If a variable is live-out at node n and not in def[n]
then the variable is also live-in at n



Data-flow equations

$$\underset{(1)}{} \boxed{\mathbf{in[n] = use[n]}} \cup \boxed{\mathbf{(out[n] - def[n])}} \ (3)$$

$$\mathbf{out[n]} = \boxed{\bigcup_{s \in succ[n]} \mathbf{in[s]}} \ (2)$$

Solving the Data-flow Equations

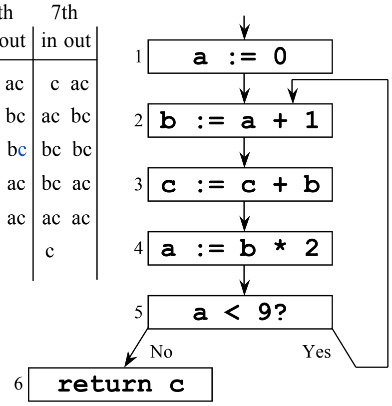
Algorithm

for each node n in CFG
  in[n] = ∅; out[n] = ∅ } initialize solutions
repeat
  for each node n in CFG
    in'[n] = in[n]
    out'[n] = out[n] } save current results
    in[n] = use[n] ∪ (out[n] – def[n])
    out[n] = $\bigcup_{s \in succ[n]}$ in[s] } solve data-flow equations
until in'[n]=in[n] and out'[n]=out[n] for all n } test for convergence

This is **iterative data-flow analysis** (for liveness analysis)

Example

node #	use	def	1st in	1st out	2nd in	2nd out	3rd in	3rd out	4th in	4th out	5th in	5th out	6th in	6th out	7th in	7th out
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	



Data-flow Equations for Liveness

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

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

Example (cont)

Data-flow Equations for Liveness

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

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

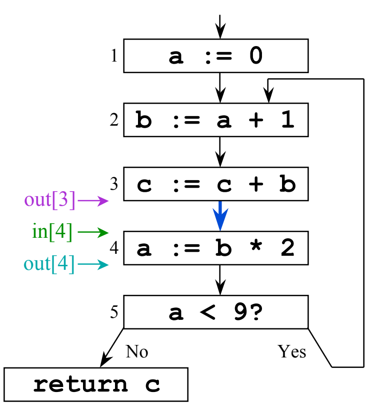
Improving Performance

Consider the **(3→4)** edge in the graph:

out[4] is used to compute in[4]

in[4] is used to compute out[3] . . .

So we should compute the sets in the order: out[4], in[4], out[3], in[3], . . .

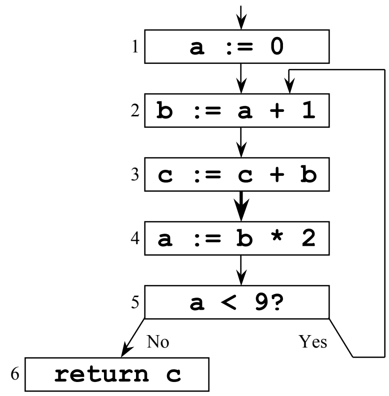


The order of computation should follow the direction of flow

Iterating Through the Flow Graph Backwards

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

Converges much faster!



Solving the Data-flow Equations (reprise)

Algorithm

for each node n in CFG
  in[n] = ∅; out[n] = ∅ — Initialize solutions

repeat
  for each node n in CFG **in reverse topsort order**
    in'[n] = in[n]
    out'[n] = out[n] — Save current results
    $\mathbf{out[n]} = \bigcup_{s \in succ[n]} \mathbf{in[s]}$
    $\mathbf{in[n]} = \mathbf{use[n]} \cup (\mathbf{out[n]} - \mathbf{def[n]})$ — Solve data-flow equations

until in'[n]=in[n] and out'[n]=out[n] for all n — Test for convergence

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

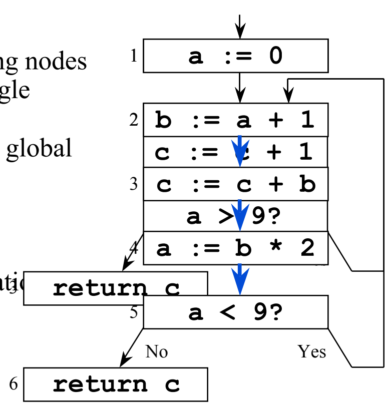
Worst case: $O(N^4)$

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**
(requires local analysis before and after global analysis)



One variable at a time

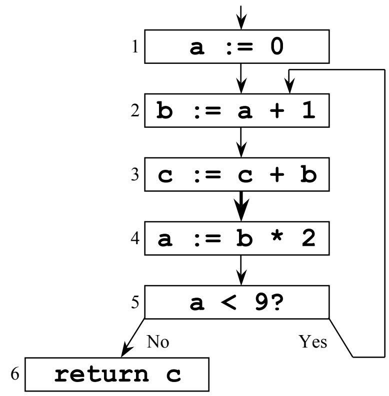
– Instead of computing data-flow information for all variables at once using sets, compute a (simplified) analysis for each variable separately

Representation of sets

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

Conservative Approximation

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	

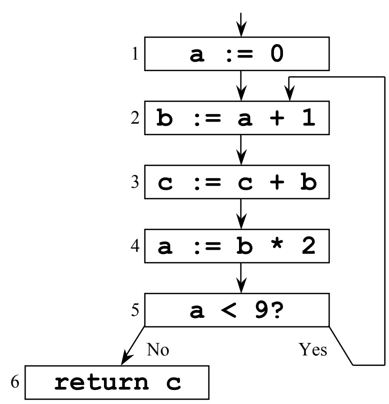


Solution X

– Our solution as computed on previous slides

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 Y

– Carries variable d uselessly around the loop

– Does Y solve the equations?

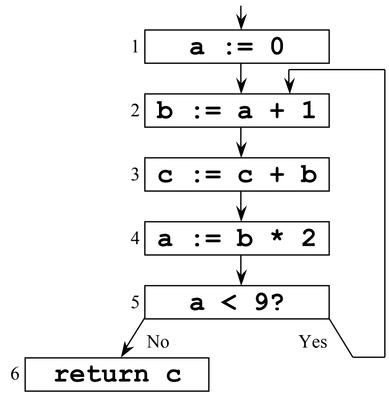
– Is d live?

– Does Y lead to a correct program?

Imprecise conservative solutions ⇒ sub-optimal but correct programs

Conservative Approximation (cont)

			X		Y		Z	
node #	use	def	in	out	in	out	in	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?

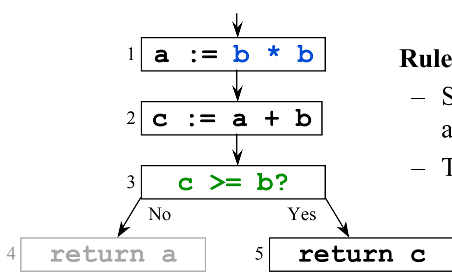
– Does Z lead to a correct program?

Non-conservative solutions ⇒ incorrect programs

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 a program's dynamic properties!

Concepts

Liveness

- Use in register allocation
- Generating liveness
- Flow and direction
- Data-flow equations and analysis
- Complexity
- Improving performance (basic blocks, single variable, bit sets)

Control flow graphs

- Predecessors and successors

Defs and uses

Conservative approximation

- Static versus dynamic liveness

Next Time

Lecture

- Generalizing data-flow analysis