

Introduction to Data-flow analysis

Last Time

- Implementing a Mark and Sweep GC

Today

- Control flow graph construction
- Liveness analysis

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

```
1   a := 0
2  L1: b := a + 1
3   c := c + b
4   a := b * 2
5   if a < 9 goto L1
6   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

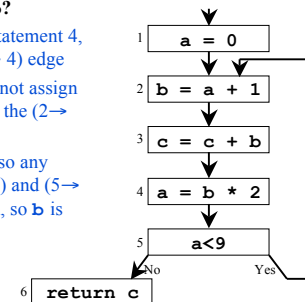
- A program contains an unbounded number of variables
- Must execute on a machine with a bounded number of registers
- Two variables can use the same register if they are never in use at the same time (*i.e.*, never simultaneously live).
- ∴ Register allocation uses liveness information

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

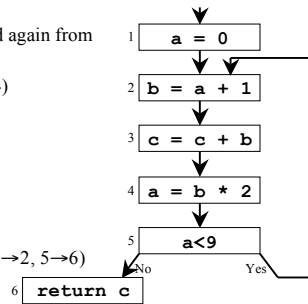
- a is live from (1→2) and again from (4→5→2)
- a is dead from (2→3→4)

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

Control Flow Graphs (CFGs)

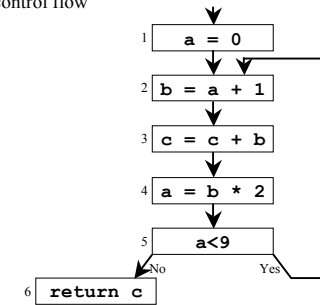
Definition

- A CFG is a graph whose nodes represent program statements and whose directed edges represent control flow

Example

```

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



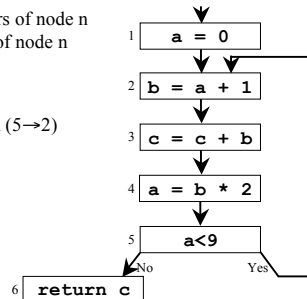
Terminology

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)

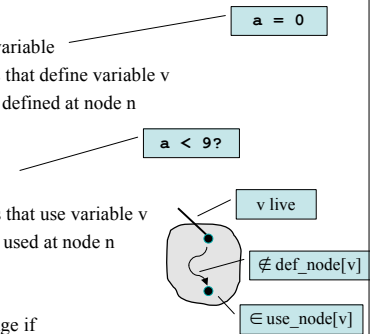
- An **assignment** of a value to a variable
- def_node[v] = set of CFG nodes that define variable v
- def[n] = set of variables that are defined at node n

Use

- A **read** of a variable's value
- use_node[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) \exists a directed path from that edge to a use of v (node in use_node[v]), and
 - (2) that path does not go through any def of v (no nodes in def_node[v])



The Flow of Liveness

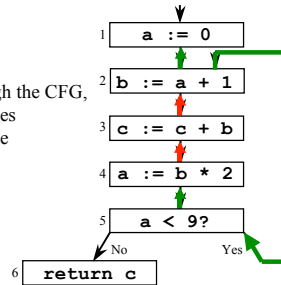
Data-flow

- 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

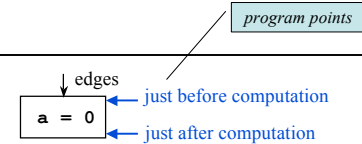
- Consider **a**
- Consider **b**
- Later, we'll see other properties that flow **forward**



Liveness at Nodes

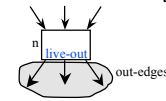
We have liveness on edges

- How do we talk about liveness at nodes?

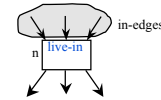


Two More Definitions

- A variable is **live-out** at a node if it is live on **any** of that node's out-edges



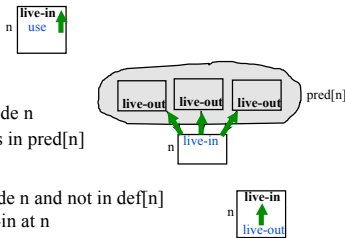
- A variable is **live-in** at a node if it is live on **any** of that node's in-edges



Computing Liveness

Rules for computing liveness

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

$$(1) \text{in}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \quad (3)$$

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

Solving the Data-flow Equations

Algorithm

```

for each node n in CFG
    in[n] = ∅; out[n] = ∅
repeat
    for each node n in CFG
        in'[n] = in[n]
        out'[n] = out[n]
        in[n] = use[n] ∪ (out[n] - def[n])
        out[n] = ∪s ∈ succ[n] in[s]
    until in'[n]=in[n] and out'[n]=out[n] for all n
  
```

} initialize solutions

} save current results

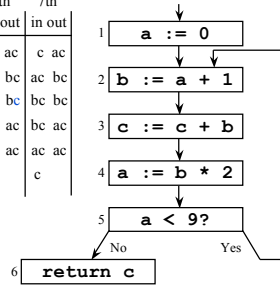
} solve data-flow equations

} test for convergence

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

Example

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

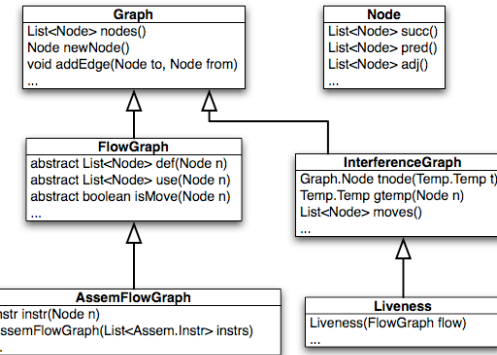


Data-flow Equations for Liveness

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

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

Liveness in the MiniJava compiler



Concepts

Liveness

- Used in register allocation
- Generating liveness
- Flow and direction
- Data-flow equations and analysis

Control flow graphs

- Predecessors and successors

Defs and uses

Next Time

Reading

- Ch. 8.4, 9.2-9.25, intro to data-flow analysis
- Ch 8.8, register allocation

Lecture

- Register allocation

Suggested Exercises

- For last week:
 - 7.4.1, what would heap look like (draw pointers as arrows) with a singly-linked free list? how would the best-fit algorithm work?
 - 7.5.2, how does type safety or lack thereof affect GC?
- This week:
 - 8.4.1, 9.2.1, 9.2.3, 8.8.1