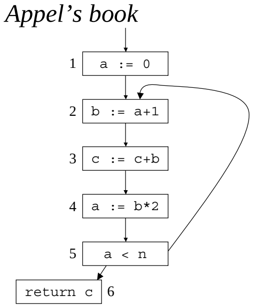
CSE401: Analysis

David Notkin
Autumn 2000

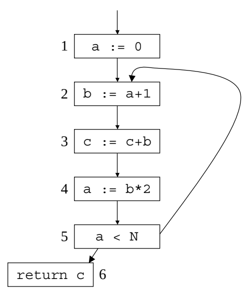
Let's make data flow concrete:
example from Appel's book

- A variable is live if its current value will be used
  - Variable `b` is used in 4, so it is live on the (3,4) edge
  - Since 3 doesn't assign into `b`, `b` is also live on (2,3)
  - Statement 2 assigns to `b`, so the contents of `b` on the (1,2) edge aren't needed by anyone: `b` is dead on that edge
  - So variable `b` is live on (2,3) and (3,4), but nowhere else



`a`'s liveness?

- Live on (1,2) and also on (4,5) and (5,2)
- But it's not live on (2,3) and (3,4), even though it has a defined value
  - It's not used before a new assignment to `a` is made
- (Since `a` and `b` are never live on the same edges, they could share a register)



def-use

Node #	def	use
1	{a}	
2	{b}	{a}
3	{c}	{b,c}
4	{a}	{b}
5		{a} N
6		{c}

Treat as constant

Liveness defined by def-use

- A variable is live on an edge if there is a directed path from the edge to a use of the variable that does not go through any def
  - That is, it isn't killed by another def

1) If a statement uses a variable, the variable is live on entry to that node
2) If a variable is live on entry to a node, then it is live on exit from all predecessor nodes
3) If a variable is live on exit from a node and is not defined by the node, then it is live on entry to the node

In equation form

- `in[n] = use[n]` $\cup$ `(out[n] – def[n])`
  - Variables live at node n are those used in n plus those that are live when they leave n except those defined in n
- `out[n] =` $\cup_{s \in succ[n]}$ `in[s]`
  - Find all nodes that are successors of n; any variable that leaves n live enters those nodes live

- Our goal is to compute liveness --- `in` and `out` --- given `def` and `use`
- Note that `in` is defined in terms of `out`, and `out` in terms of `in`

Algorithm:

solve these equations by iteration

for each n
  in[n] := {}; out[n] := {};
repeat for each n
  in'[n] := in[n]; out'[n] := out[n]
  in[n] := use[n] $\cup$ (out[n]-def[n])
  out[n] := $\cup_{s \in succ[n]}$ in[s]
until in'[n] = in[n] and out'[n] = out[n] for all n

rows: nodes • columns: iterations

			#1		#2		#3		#4		#5		#6		#7	
	use	def	in	out	in	out	in	out	in	out	in	out	in	out	in	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	c	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	

Note: Node order (left column) reversed!

Change order of iterations: from 6 to 1

			#1		#2		#3		#4		#5		#6		#7	
	use	def	in	out	in	out	in	out	in	out	in	out	in	out	in	out
6		a		c		c		c								
5	a	b	c	ac	ac	ac	ac	ac	These iterations aren't needed: fixed point already reached							
4	bc	c	ac	bc	ac	bc	ac	bc								
3	b	a	bc	bc	bc	bc	bc	bc								
2	a		bc	ac	bc	ac	bc	ac								
1	c		ac	c	ac	c	ac	c								

Some observations

- The iteration order is key to performance
  - For this data flow computation, since it is in some sense naturally "backwards", it tends to be more efficient iterating over the CFG in "reverse"
  - A simple depth-first search can be used to find an effective ordering
- In practice, with efficient representations, the algorithm is usually O(N) or O(N^2) for a program of N nodes
  - Bit vectors are commonly used to represent the sets of variables; good for dense sets
  - Sorted linked lists are also used; good for sparse sets
- It turns out that there are multiple solutions to the dataflow equations: however, there is one *least* (minimal) solution
- Finally, remember this is conservative: it may show a variable is live when it in fact never is