编译原理

10. 活跃变量分析

rainoftime.github.io
浙江大学
计算机科学与技术学院

Content

Outline

1. Compiler Optimizations **(不考)**
2. Dataflow Analysis
3. Liveness Analysis
4. More Discussions

1. Compiler Optimizations

Example: Optimization Levels in Clang

优化等级	简要说明
-Ofast	在-O3级别的基础上，开启更多激进优化项，该优化等级不会严格遵循语言标准
-O3	在-O2级别的基础上，开启了更多的高级优化项，以编译时间、代码大小、内存为代价获取更高的性能。
-Os	在-O2级别的基础上，开启降低生成代码体量的优化
-O2	开启了大多数中级优化，会改善编译时间开销和最终生成代码性能
-O/-O1	优化效果介于-O0和-O2之间
-O0	默认优化等级，即不开启编译优化，只尝试减少编译时间

Example: Compiler Optimizations

- Space optimization: reduce memory use
- Time optimization: reduce execution time
- Power optimization: reduce power usag

```
int bar(){
   int a = 10*10;    //这里在编译时可以直接计算出100这个值，这叫做“常数折叠”
   int b = 20;     //这个变量没有用到，可以在代码中删除，这叫做“死代码删除”
   if (a>0){          //因为a一定大于0，所以判断条件和else语句都可以去掉
      return a+1; //这里可以在编译器就计算出是101
   }
   else {
      return a-1; //死分支，编译器会将该分支删除掉
   }
}
int a = bar();     //这里可以直接换成a=101
```

Example: Compiler Optimizations

- **把常量提前计算出来**
  - 常量折叠（Constant Folding）
  - 常量传播（Constant Propagation）
- **用低代价的方法来做计算**
  - 代数简化（Algebra Simplification）
  - 强度折减（Strength Reduction）
- **消除重复计算**
  - 拷贝传播（Copy Propagation）
  - 值编号（Value Numbering）
  - 公共子表达式消除（Common Subexpression Elimination，CSE）
- **向量化运行**
  - 超字级并行（Superword-Level Parallelism，SLP)
  - 循环向量化（Loop Vectorization）
- **循环优化**
  - 循环不变代码外提（Loop-Invariant Code Motion，LICM）
  - 循环展开（Loop Unrolling）
  - 循环合并或拆分（Loop Fusion/Distribution）
  - 代码提升（Code Hoisting）
- **减少调用开销**
  - 内联优化（Function Inline）
  - 尾递归调用优化（Tail Call optimization）
- **控制流优化**
  - 死代码消除（Dead-code Elimination DCE）
  - If简化（If Simplification）

Granularities/Scopes of Optimizations

- **Local**
  - Work on a single basic block
  - Consider multiple blocks, but less than whole procedure
- **Intraprocedural (or “global’)**
  - Create on an entire procedure
- **Interprocedural (or “whole-program”)**
  - Operate on > 1 procedure, up to whole program
  - Sometimes, at link time (LTO, link time optimization)

Regardless of Optimization Level

- **Analyze program to gather "facts"**
  - Perform "program analysis" on the program's **IR**

- Possible values of variables
- v is a constant k
- v points only to vars in set S
- what are upper/lower bounds on the value of x at point p?
- var v may/may not be used subsequently (live/dead)
- value assigned at def-site d: x = … may be used at use-site u: … x … (d reaches u)

- **Apply transformation (e.g., optimizations)**

Example: Analyses for Optimizations

- 为了实现上述优化，编译器会对代码做一些分析。常见的优化分析方法总结如下：

控制流分析（Control-Flow Analysis）	• 哪些语句构成了一个基本块，基本块之间跳转关系，哪个结构是一个循环结构，等等。
数据流分析（Data-Flow Analysis）	• 数据流分析可以帮忙梳理出数据的活跃情况，引用情况等。常被用于做常量优化、向量化优化等
别名分析（Alias Analysis）	• 不同的指针可能指向同一地址。编译器需知道不同变量是否是别名关系，以便决定能否做某些优化

Example: IRs for Analyses and Optimizations

At each of these scopes, the compiler uses various intermediate representations (IRs) for performing the analyses and optimizations

- **Local**
  - E.g., dependence graph
- **Intraprocedural (or global)**
  - E.g., control-flow graph
- **Interprocedural (or who-program)**
  - E.g., Call graph

2. Dataflow Analysis

The Control Flow Graph (CFG)

- **CFG (Control Flow Graph):** A directed graph
  - Each node represents a statement
  - Edges represent control flow

- Statements may be
  - Assignments x := y op z or x := op z
  - Copy statements x := y
  - Branches goto L or if x relop y goto L
  - etc.

Concept invented in 1970 by:

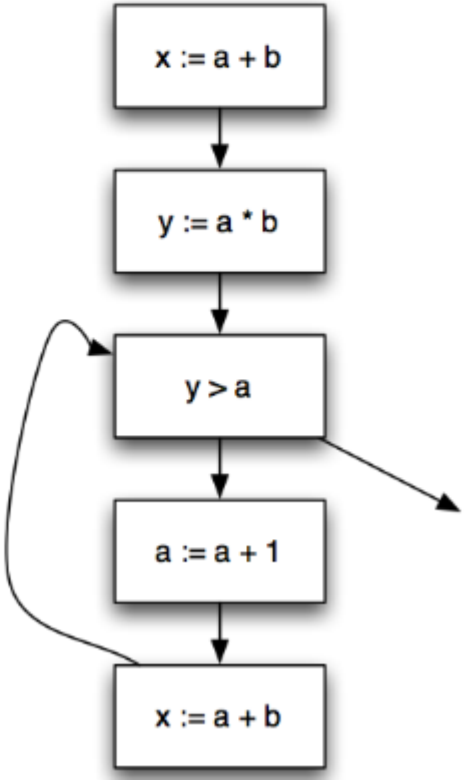
Frances Allen (1932–2020), IBM,

(1st woman to receive Turing Award in 2006!)

Example: Control Flow Graph (CFG)

- node: a statement.
- edge: control flow.

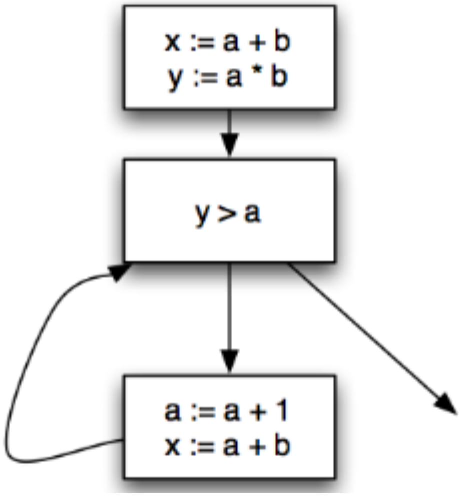
```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b;
}
```



Variants on Control Flow Graph (CFG)

- May group statements into basic blocks
  - A basic block: a sequence of instructions with unique entry and exit

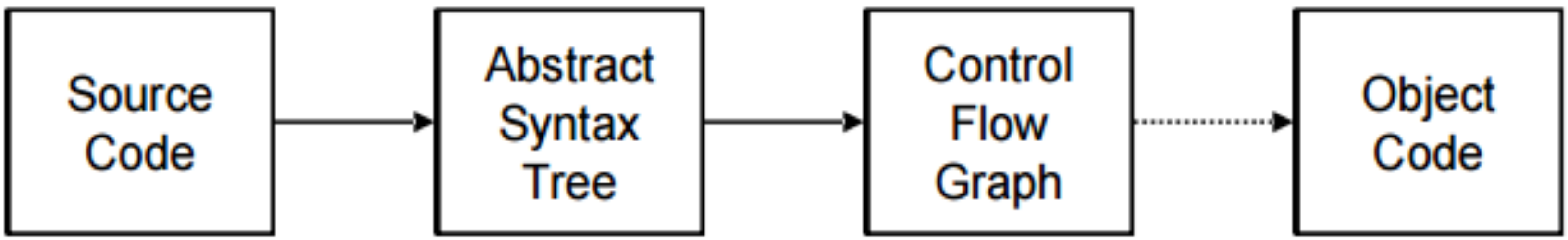
```
x := a + b;
y := a * b;
while (y > a + b) {
    a := a + 1;
    x := a + b;
}
```



We'll use single-statement blocks in this lecture.

Dataflow Analysis

- A framework for deriving information about the dynamic behavior of a program without running it
  - E.g., liveness information of variables

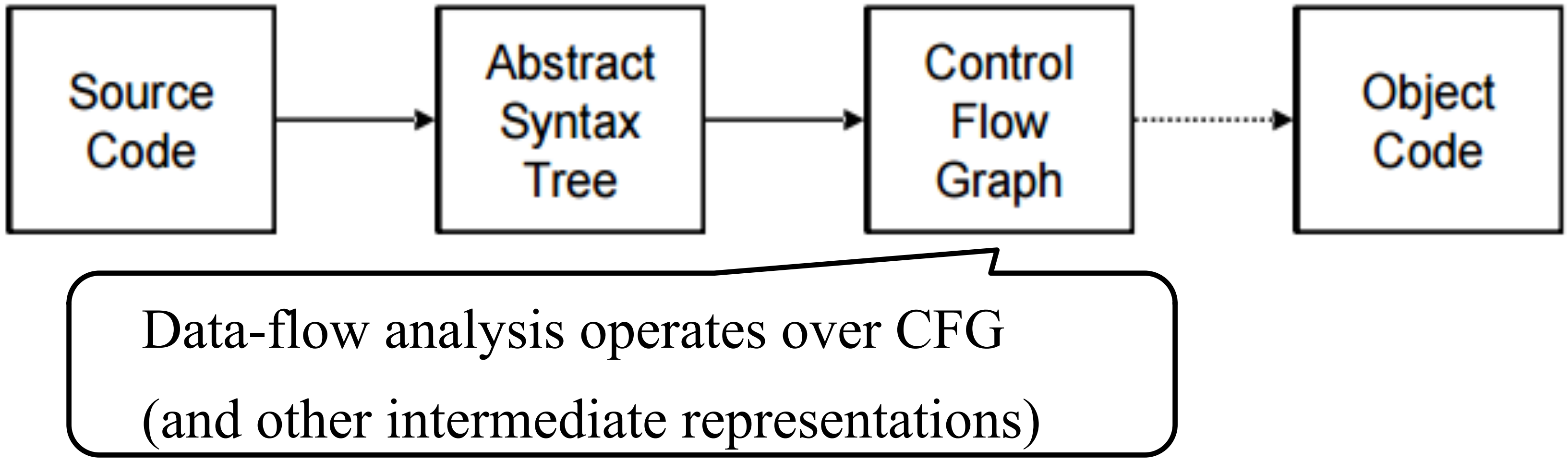


Data-flow analysis operates on CFG

(and other intermediate representations)

Dataflow Analysis

- A framework for deriving information about the dynamic behavior of a program without running it
  - "Dataflow facts" : liveness, types, …
  - Applications in optimization/verification/…

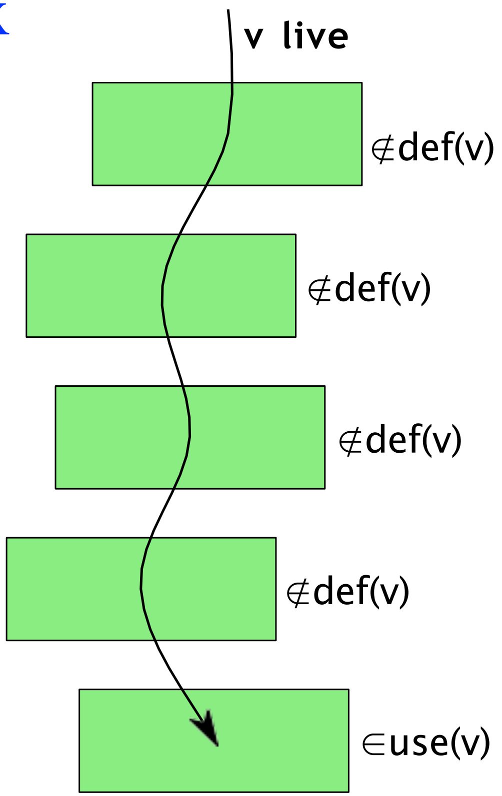


3. Liveness Analysis

- Liveness Variables
- Dataflow Equations for Liveness
- Solving the Equations

Live Variables

- A variable x is live at statement s if
  - There exists a statement s' that uses x
  - There is a path from s to s'
  - That path has no intervening assignment to x



Motivation Example: Register Allocation

- Low level IRs assume an infinite number of “abstract registers”
  - Good for code generations
  - But bad for execution on a real machine: machine has a finite number of registers

- The goal of register allocation is to put infinite variables into a finite machine registers
  - Many register allocation alg. need **liveness analysis**

Motivation Example: Register Allocation

Consider this three-address code

```
1: a = 1
2: b = a + 2
3: C = b + 3
4: return c
```

- Three variables: a, b, and c.
- And assume that the target machine has only one register: r.
- Is it possible to put all three variables “a”, “b” and “c” in register “r”?

Motivation Example: Register Allocation

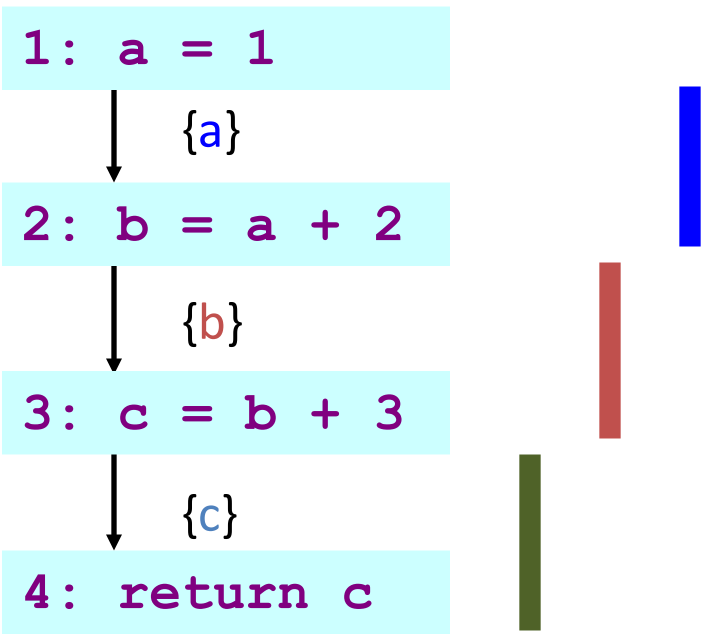
Consider this three-address code

```
1: a = 1
   {a}
2: b = a + 2
   {b}
3: c = b + 3
   {c}
4: return c
```

- Calculate which variable is “live” at a given program point.
- The “liveness” info. gives **live ranges**.
- Live ranges of
  - a: 1->2
  - b: 2 -> 3
  - c: 3 -> 4

Motivation Example: Register Allocation

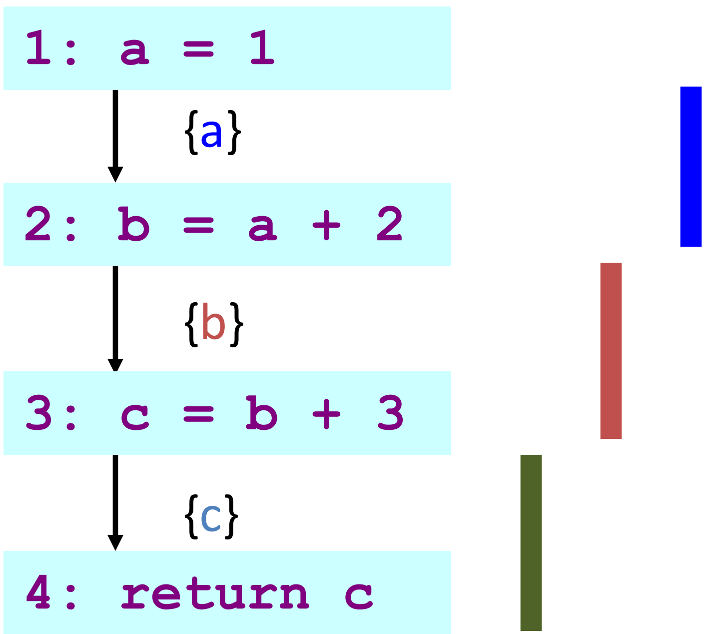
Consider this three-address code



- Calculate which variable is “live” at a given program point.
- The “liveness” info. gives **live ranges**.
- **Live ranges of a, b, c** don’t overlap, thus all three variables can be put into ONE register!

Motivation Example: Register Allocation

Consider this three-address code



- Register allocation:

a => **r**

b => **r**

c => **r**

- Code rewriting:

r = 1

r = r + 2

r = r + 3

return r

More Application of Liveness Information

- **Redundant instruction elimination**
  – Remove unused assignments
- **IR construction**
  – Optimize SSA construction
- **Security**
  – Detect the use of uninitialized variables
- ?

3. Liveness Analysis

- Liveness Variables
- Dataflow Equations for Liveness
- Solving the Equations

Undecidability of (Static) Program Analysis

- We cannot precisely compute live variables. To see why, consider the following code:

```
1: x = 10; // is x live here?
2: f();
3: return x;
```

Undecidability of (Static) Program Analysis

- We cannot precisely compute live variables. To see why, consider the following code:

```
1: x = 10; // is x live here?
2: f();
3: return x;
```

- It seems to be obvious that x is live after Line 1
- However, suppose that f() never returns!
  - In that case the value of x is not needed
  - In other words, x is live if f() halts

Undecidability of (Static) Program Analysis

- We cannot precisely compute live variables. To see why, consider the following code:

```
1: x = 10; // is x live here?
2: f();
3: return x;
```

- It seems to be obvious that x is live after Line 1
- However, suppose that f() never returns!
  - In that case the value of x is not needed
  - In other words, x is live if f() halts
- Since the **halting problem** is undecidable, so is the live variable problem (at least if we want precise results)!

Approximating the "Exact Solutions"

- When we do (static) program analysis, we are usually **approximating** facts about the programs

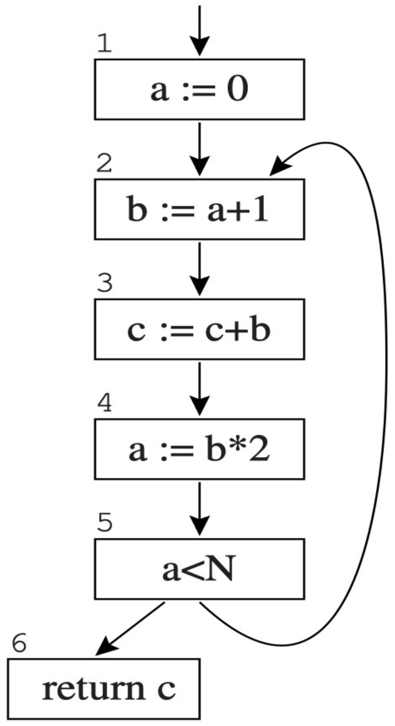
- For **liveness analysis**
  - **Overapproximates** the true set of live variables by finding all of the variables that *may* be needed later

Typically, formulated as a **dataflow analysis** problem

Dataflow analysis operates on **CFG**
(and other intermediate representations)

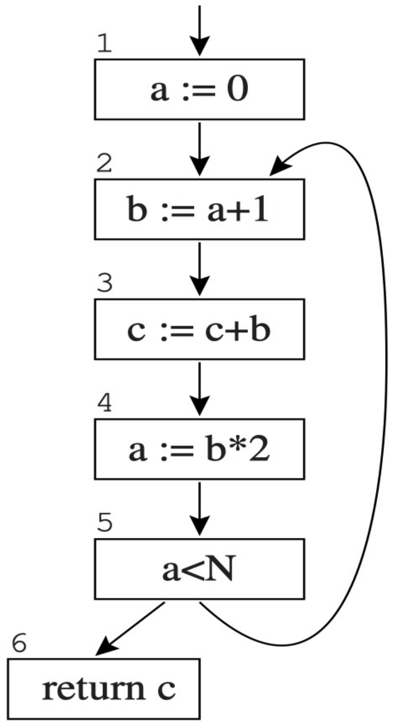
Control-Flow Graph (CFG) Terminology

- Liveness of variables "flows" around the edges of the control-flow graph (CFG)
- A CFG node has
  - **out-edges**: lead to successor nodes
  - **in-edges**: come from predecessor nodes
  - **pred[n]**: the predecessors of node n
  - **succ[n]**: the successors of node n
- Example
  - out-edges of node 5: ?
  - succ[5] = ?
  - in-edges of 2 are ?
  - pred[2] = ?



Control-Flow Graph (CFG) Terminology

- Liveness of variables “flows” around the edges of the control-flow graph (CFG)
- A CFG node has
  - **out-edges**: lead to successor nodes
  - **in-edges**: come from predecessor nodes
  - **pred[n]**: the predecessors of node n
  - **succ[n]**: the successors of node n
- Example
  - out-edges of node 5: 5->6, 5-2
  - succ[5] = {2, 6}
  - in-edges of 2 are 5->2, 1->2
  - pred[2] = {1, 5}



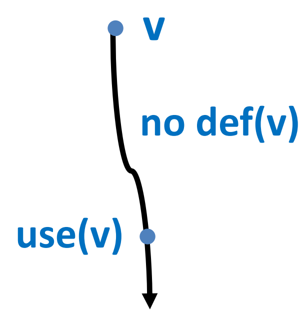
Dataflow Analysis Cont.

To compute the "dataflow facts" (e.g., liveness)

1. Set up **local equations** at each CFG node
   - For a CFG node n, we write
     - **in[n]**: facts that are true on all in-edges to the node
     - **out[n]**: facts true on all out-edges
   - Define **transfer functions** that transfer information from one node to another
2. **Solve the equations** to compute the desired information
   - Iteratively update **in[n]** and **out[n]**

Recap: Liveness Variables

- **Liveness variable**: a variable x is live at statement s if
  - There exists a statement s' that **uses** x
  - There is a path from s to s'
  - That path has no intervening **assignment/definitions** to x

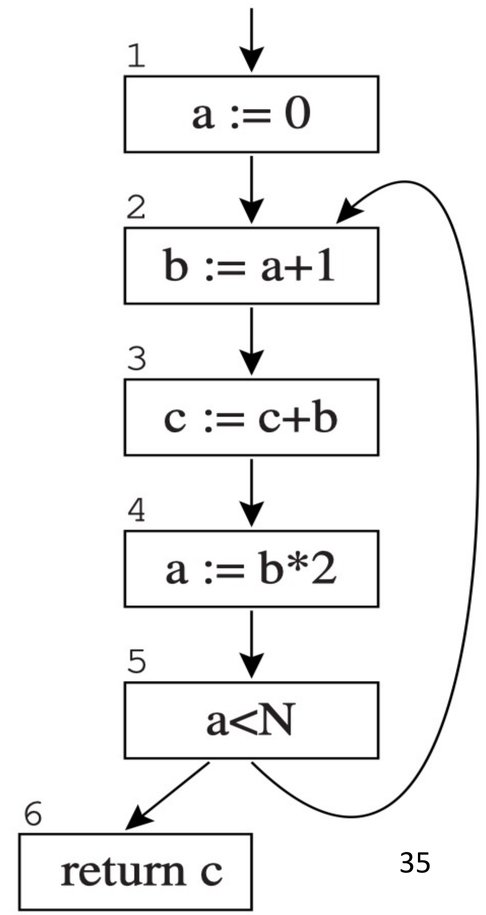


What are "uses" and "defs", and how to define in and out

Uses and Defs

- An assignment to a variable or temporary **defines** that variable.
- An occurrence of a variable on the right-hand side of an assignment (or in other expressions) **uses** a variable.

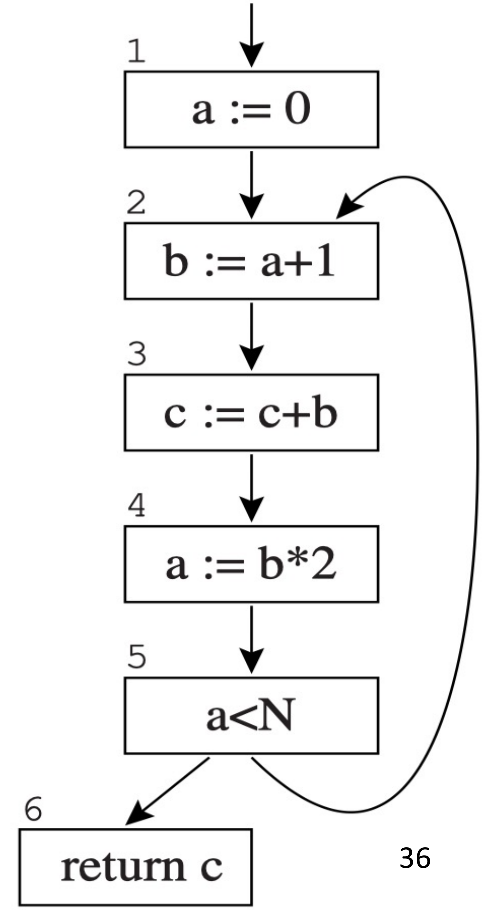
- **def** of a variable
  - the set of graph nodes that define it
- **def** of a graph node
  - the set of variables that it defines
- **use** of a variable or graph node is similar.
- Example
  - def(3) = **?**, def(a) = **?**
  - use(3) = **?**, use(a) = **?**



Uses and Defs

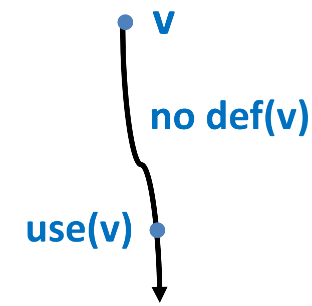
- An assignment to a variable or temporary **defines** that variable.
- An occurrence of a variable on the right-hand side of an assignment (or in other expressions) **uses** a variable.

- **def** of a variable
  - the set of graph nodes that define it
- **def** of a graph node
  - the set of variables that it defines
- **use** of a variable or graph node is similar.
- Example
  - def(3) = {c}, def(a) = {1, 4}
  - use(3) = {b, c}, use(a) = {2, 5}

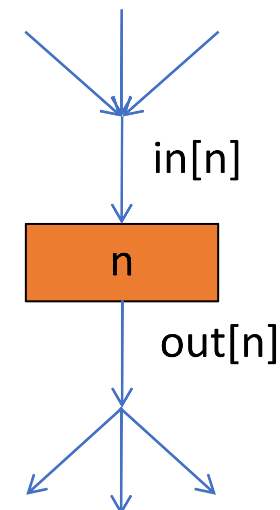


Liveness Facts

- A variable is **live on an edge** if there is a directed path from that edge to a **use** of the variable that does not go through any **def**.
  - This variable will be used later
  - This variable will not be re-defined before being used

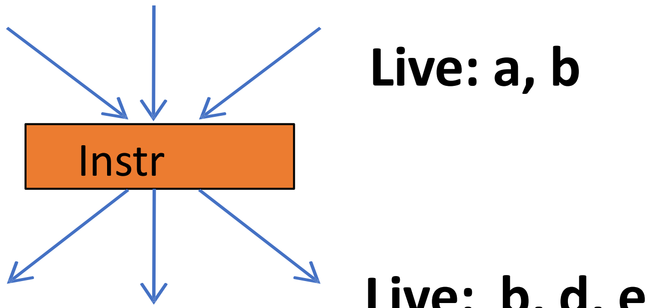


- **Live-in**: a variable is **live-in** at a node if it is live on any of the in-edges of that node;
- **Live-out**: A variable is **live-out** at a node if it is live on any of the out-edges of the node.



Liveness Facts

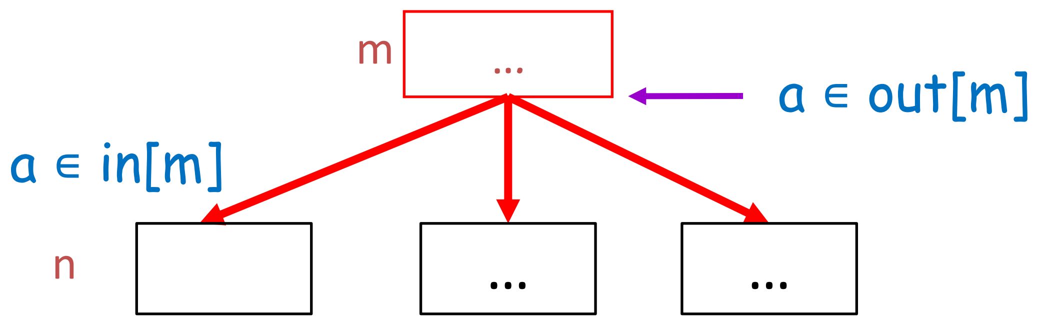
- **Notations**
  - in[n] : the **live-in** set of node n (the variables that are live-in at node n)
  - out[n] : the **live-out** set of node n (the variables that are live-out at node n)



How to calculate in[n] and out[n]
(i.e., define the dataflow equations)?

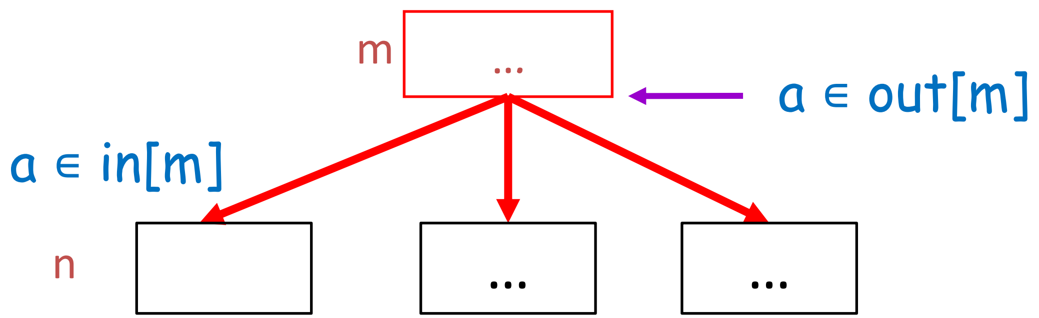
Dataflow Equations for Liveness

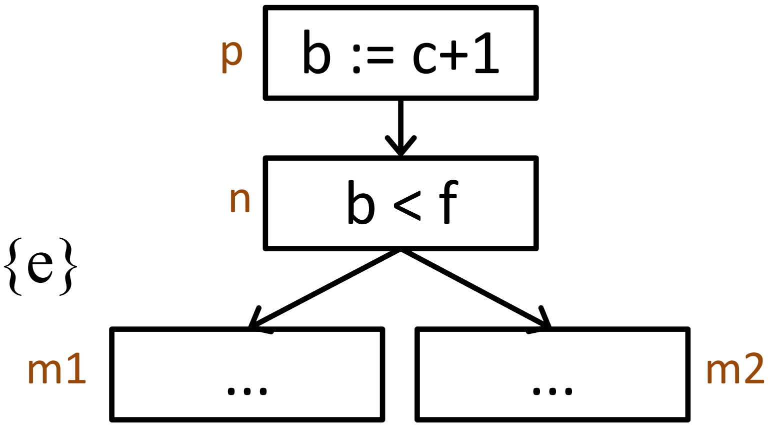
- **Rule 1:** If $a \in in[n]$, then for $\forall m \in pred[n]$, $a \in out[m]$
  – If a variable is *live-in* at a node n, then it is *live-out* at all nodes m in *pred*[n]



Dataflow Equations for Liveness

- **Rule 1:** If a ∈ in[n], then for ∀m ∈ pred[n], a ∈ out[m]
  - If a variable is *live-in* at a node n, then it is *live-out* at all nodes m in *pred*[n]

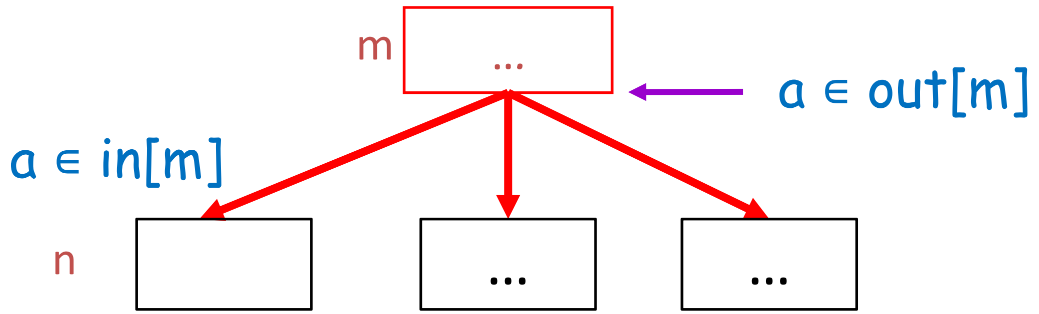




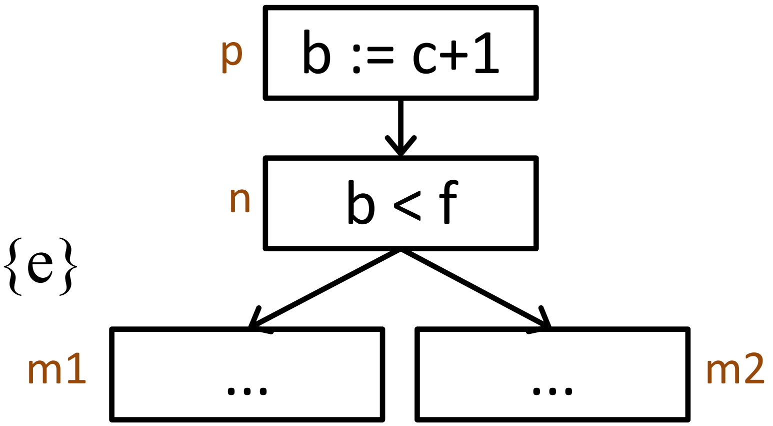
- Example
  - Suppose: in[m1] = {d}, in[m2] = {e}
  - **out[n] = ?**

Dataflow Equations for Liveness

- **Rule 1:** If a ∈ in[n], then for ∀m ∈ pred[n], a ∈ out[m]
  - If a variable is *live-in* at a node n, then it is *live-out* at all nodes m in *pred*[n]

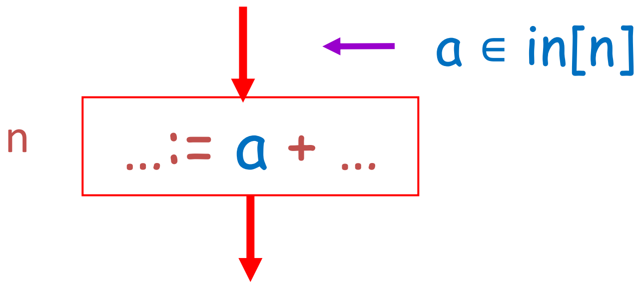


- Example
  - Suppose: in[m1] = {d}, in[m2] = {e}
  - **out[n] = {d, e}**



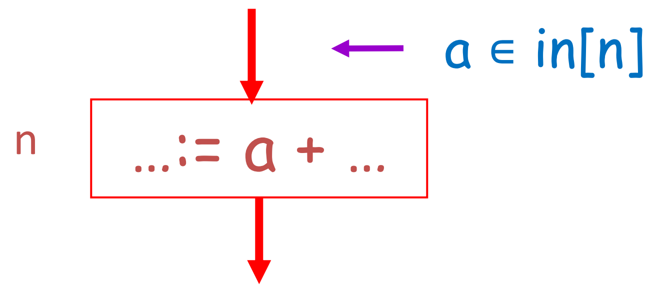
Dataflow Equations for Liveness

- **Rule 1:** If $a \in in[n]$, then for $\forall m \in pred[n]$, $a \in out[m]$
- **Rule 2:** If $a \in use[n]$, then $a \in in[n]$
  - If statement n uses variable a, then a is live on entry of n

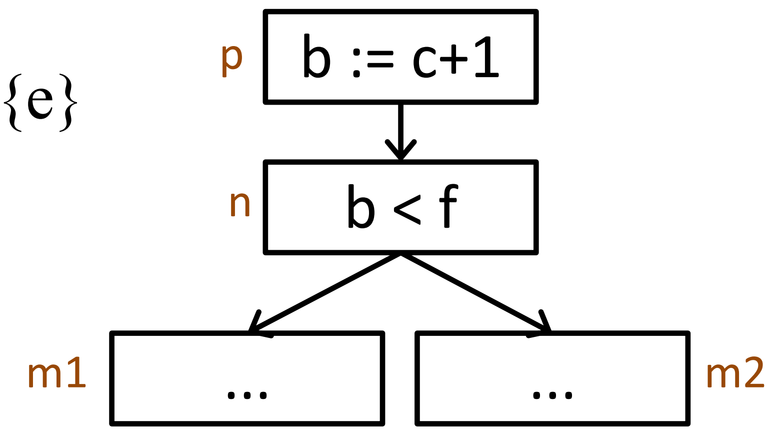


Dataflow Equations for Liveness

- **Rule 1:** If $a \in in[n]$, then for $\forall m \in pred[n]$, $a \in out[m]$
- **Rule 2:** If $a \in use[n]$, then $a \in in[n]$
  - If statement n uses variable a, then a is live on entry of n

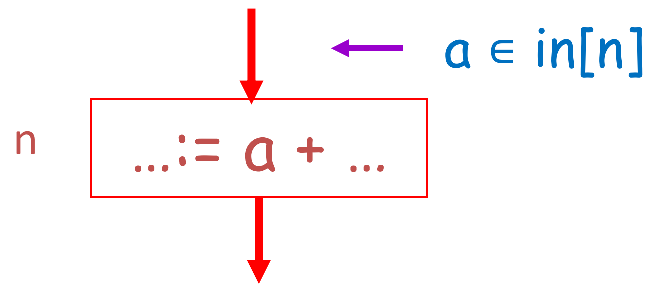


- Example
  - Suppose: in[m1] = {d}, in[m2] = {e}
  - out[n] = {d, e}
  - **in[n] = {b, f, …}**



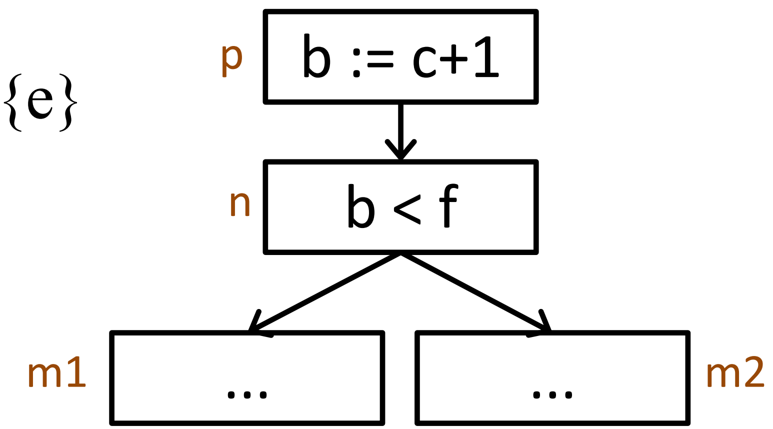
Dataflow Equations for Liveness

- **Rule 1:** If $a \in in[n]$, then for $\forall m \in pred[n]$, $a \in out[m]$
- **Rule 2:** If $a \in use[n]$, then $a \in in[n]$
  - If statement n uses variable a, then a is live on entry of n



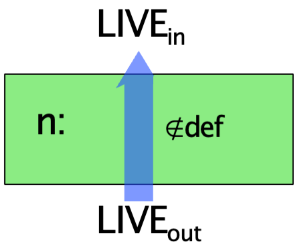
- Example
  - Suppose: $in[m1] = \{d\}$, $in[m2] = \{e\}$
  - $out[n] = \{d, e\}$
  - **in[n] = {b, f, …} (other vars?)**

  To update in[n], can we only look at the uses in node n?



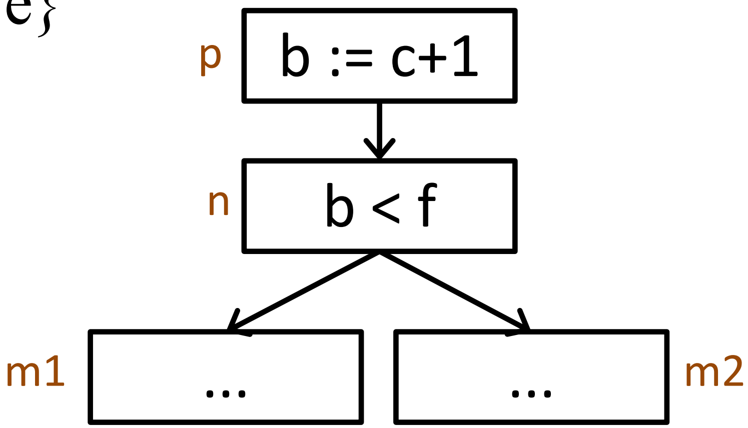
Dataflow Equations for Liveness

- **Rule 1:** If $a \in in[n]$, then for $\forall m \in pred[n]$, $a \in out[m]$
- **Rule 2:** If $a \in use[n]$, then $a \in in[n]$
- **Rule 3:** If $a \in out[n]$ and $a \notin def[n]$, then $a \in in[n]$
  - If a is live after n and not defined by n, then a is live on entry of n



- Example
  - Suppose: in[m1] = {d}, in[m2] = {e}
  - out[n] = {d, e}
  - **in[n] = {b, f, d, e}**
  - out[p] = **?**
  - in[p] = **?**

Rule 3

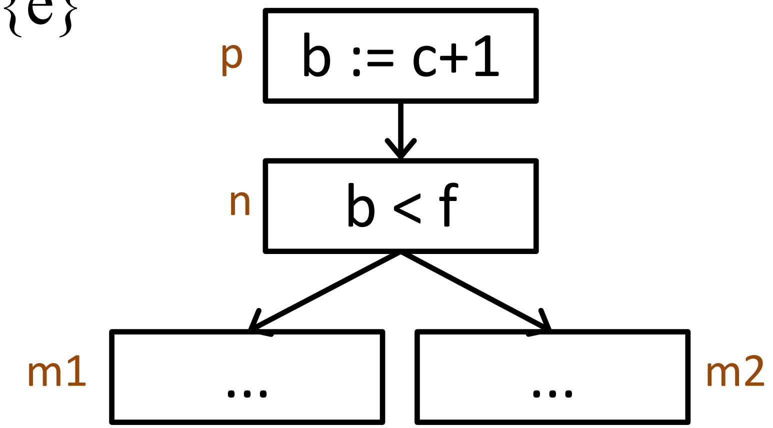


Dataflow Equations for Liveness

- **Rule 1:** If $a \in in[n]$, then for $\forall m \in pred[n]$, $a \in out[m]$
- **Rule 2:** If $a \in use[n]$, then $a \in in[n]$
- **Rule 3:** If $a \in out[n]$ and $a \notin def[n]$, then $a \in in[n]$

- Example
  - Suppose: in[m1] = {d}, in[m2] = {e}
  - out[n] = {d, e}
  - in[n] = {b, f, d, e}
  - **out[p] = {b, f, d, e}**
  - in[p] = ?

Rule 1

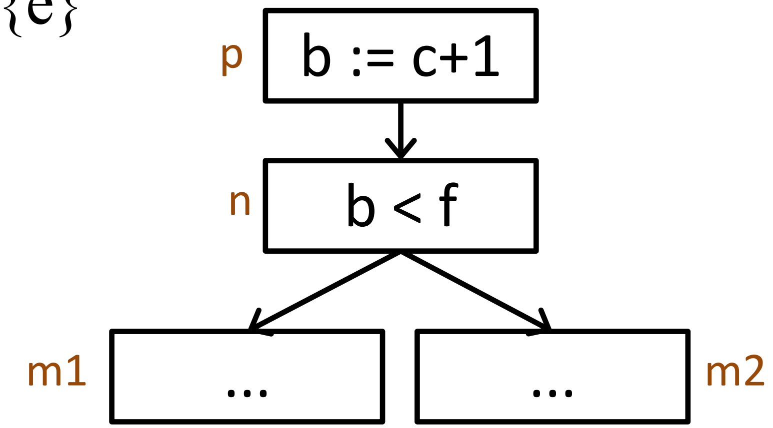


Dataflow Equations for Liveness

- **Rule 1:** If $a \in in[n]$, then for $\forall m \in pred[n]$, $a \in out[m]$
- **Rule 2:** If $a \in use[n]$, then $a \in in[n]$
- **Rule 3:** If $a \in out[n]$ and $a \notin def[n]$, then $a \in in[n]$

- Example
  - Suppose: in[m1] = {d}, in[m2] = {e}
  - out[n] = {d, e}
  - in[n] = {b, f, d, e}
  - out[p] = {b, f, d, e}
  - **in[p] = {c, f, d, e}**

Rule 3



Dataflow Equations for Liveness

- **Rule 1:** If $a \in in[n]$, then for $\forall m \in pred[n]$, $a \in out[m]$
- **Rule 2:** If $a \in use[n]$, then $a \in in[n]$
- **Rule 3:** If $a \in out[n]$ and $a \notin def[n]$, then $a \in in[n]$

Based on the above rules, we can define the dataflow equations for liveness analysis

$$\boxed{\begin{gathered}\mathbf{in}[\mathbf{n}] = \mathbf{use}[\mathbf{n}] \cup (\mathbf{out}[\mathbf{n}] - \mathbf{def}[\mathbf{n}]) \\ \mathbf{out}[\mathbf{n}] = \bigcup_{\mathbf{s} \in \mathbf{succ}[\mathbf{n}]} \mathbf{in}[\mathbf{s}]\end{gathered}}$$

3. Liveness Analysis

- Liveness Variables
- Dataflow Equations for Liveness
- Solving the Equations

Solving Dataflow Equations

Declarative Specification

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

Runnable Algorithm

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

use_B 和 def_B 的值可以直接从控制流图计算出来，因此在方程中作为已知量

Solving Dataflow Equations

for each n
  in[n] ←{}; out[n] ←{}
repeat
  for each n
    in′[n] ← in[n]; out′[n] ← out[n]
    in[n] ← use[n] ∪ (out[n] − def[n])
    out[n] ← $\bigcup_{s \in succ[n]}$ in[s]
until in′[n] = in[n] and out′[n] = out[n] for all n

- Start with a "rough" approximation to the answer

Solving Dataflow Equations

for each n
  in[n] ←{}; out[n] ←{}
repeat
  for each n
    in′[n] ← in[n]; out′[n] ← out[n]
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
until in′[n] = in[n] and out′[n] = out[n] for all n

- Start with a "rough" approximation to the answer
- Iteratively re-compute in[n] and out[n]
  - Each iteration will add variables to in[n] and out[n]
  - i.e. the live variable sets will increase monotonically
  - The sizes of in[n] and out[n] are bounded

Solving Dataflow Equations

for each n
  in[n] ← {}; out[n] ← {}
repeat
  for each n
    in′[n] ← in[n]; out′[n] ← out[n]
    in[n] ← use[n] ∪ (out[n] − def[n])
    out[n] ← $\bigcup_{s \in succ[n]}$ in[s]
until in′[n] = in[n] and out′[n] = out[n] for all n

- Start with a "rough" approximation to the answer
- Iteratively re-compute in[n] and out[n]
  - Each iteration will add variables to in[n] and out[n]
  - i.e. the live variable sets will increase monotonically
  - The sizes of in[n] and out[n] are bounded
- Keep going until a *fixed point* has been reached

Solving Dataflow Equations

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

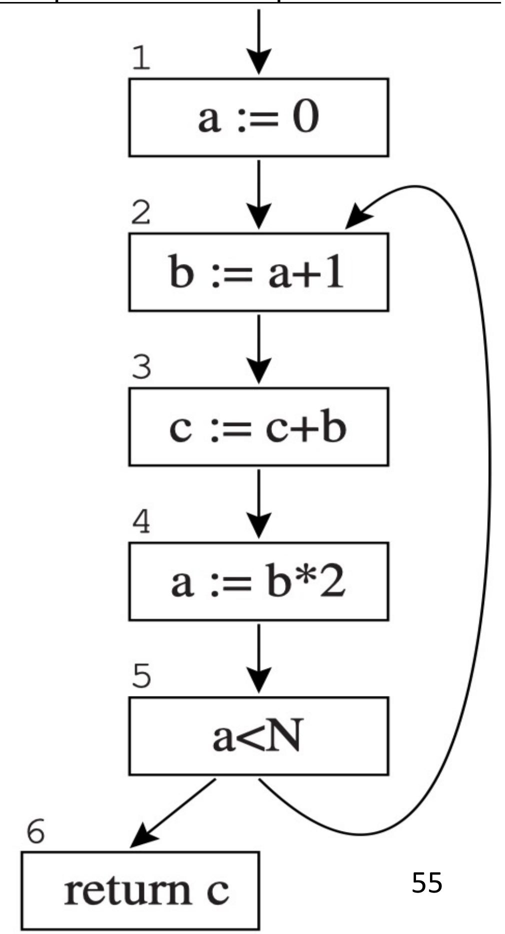
- Start with a "rough" approximation to the answer
- Iteratively re-compute in[n] and out[n]
  - Each iteration will add variables to in[n] and out[n]
  - i.e. the live variable sets will increase monotonically
  - The sizes of in[n] and out[n] are bounded
- Keep going until a *fixed point* has been reached

通过什么迭代策略可以加速算法收敛?

Example: Calculation of Liveness

- **Strategy**: following forward control-flow edges

	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														
2	a	b														
3	bc	c														
4	b	a														
5	a															
6	c															



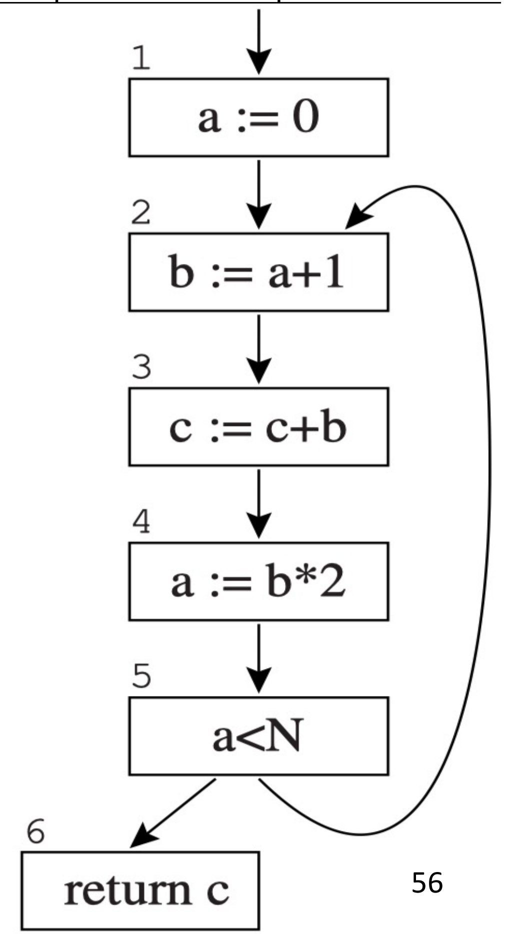
$$in[n] = use[n] \cup (out[n] - def[n])$$

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

Example: Calculation of Liveness

- **Strategy**: following forward control-flow edges

	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														
2	a	b	a													
3	bc	c														
4	b	a														
5	a															
6	c															



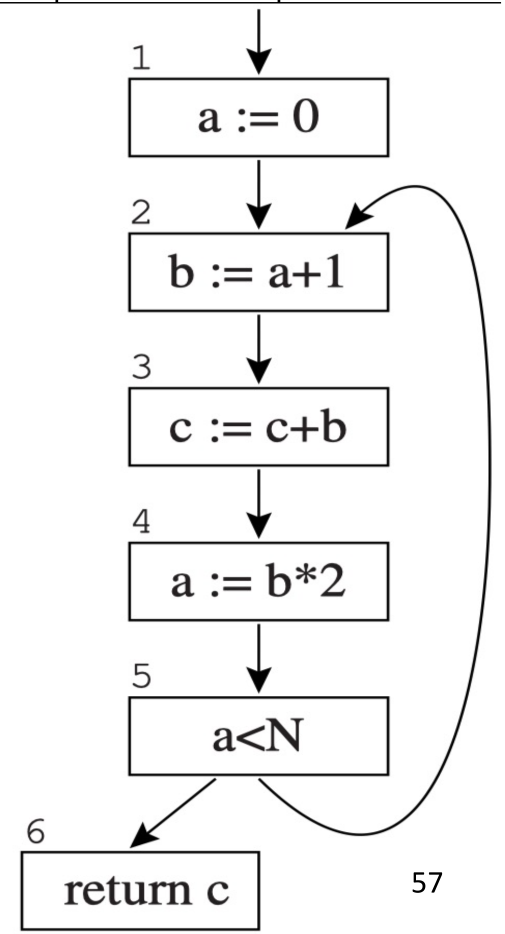
$$\boxed{in[n] = use[n] \cup (out[n] - def[n])}$$

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

Example: Calculation of Liveness

- **Strategy**: following forward control-flow edges

	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														
2	a	b	a													
3	bc	c	bc													
4	b	a														
5	a															
6	c															



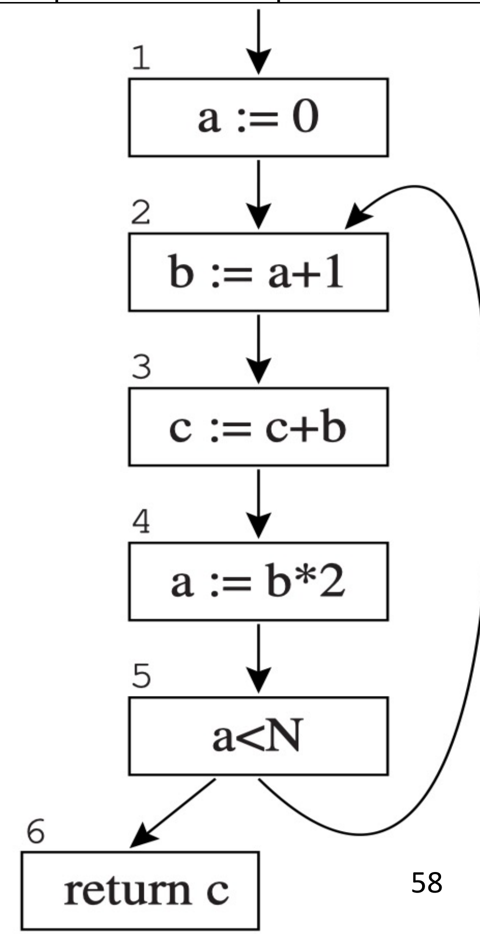
$$\boxed{in[n] = use[n] \cup (out[n] - def[n])}$$

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

Example: Calculation of Liveness

- **Strategy**: following forward control-flow edges

	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														
2	a	b	a													
3	bc	c	bc													
4	b	a	b													
5	a															
6	c															



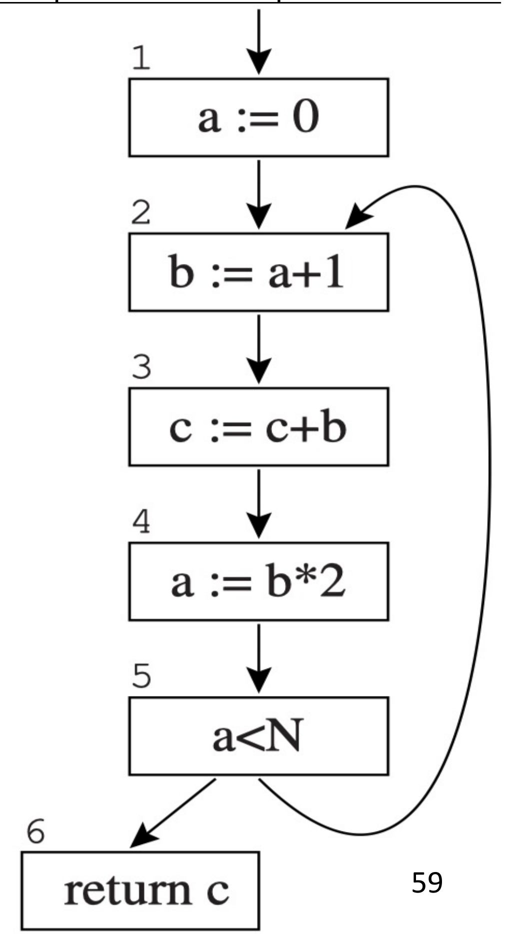
$$\boxed{in[n] = use[n] \cup (out[n] - def[n])}$$

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

Example: Calculation of Liveness

- **Strategy**: following forward control-flow edges

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



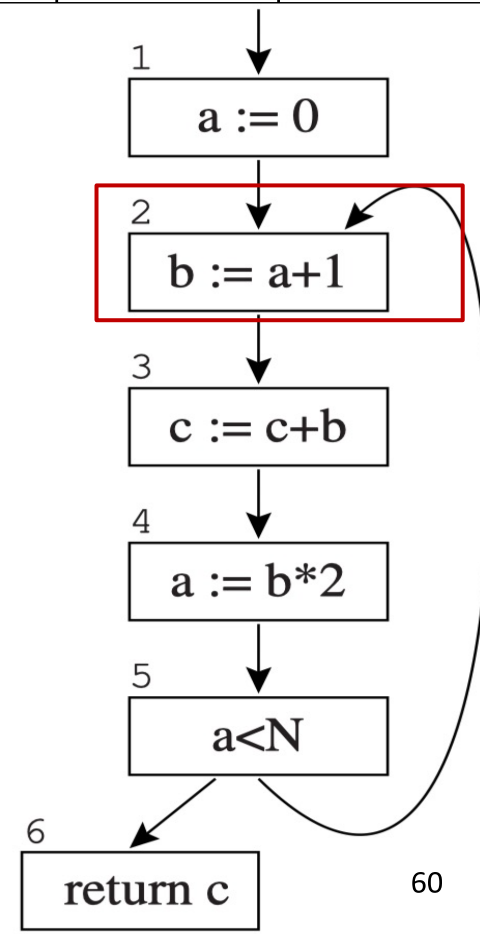
$$\boxed{in[n] = use[n] \cup (out[n] - def[n])}$$

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

Example: Calculation of Liveness

- **Strategy**: following forward control-flow edges

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



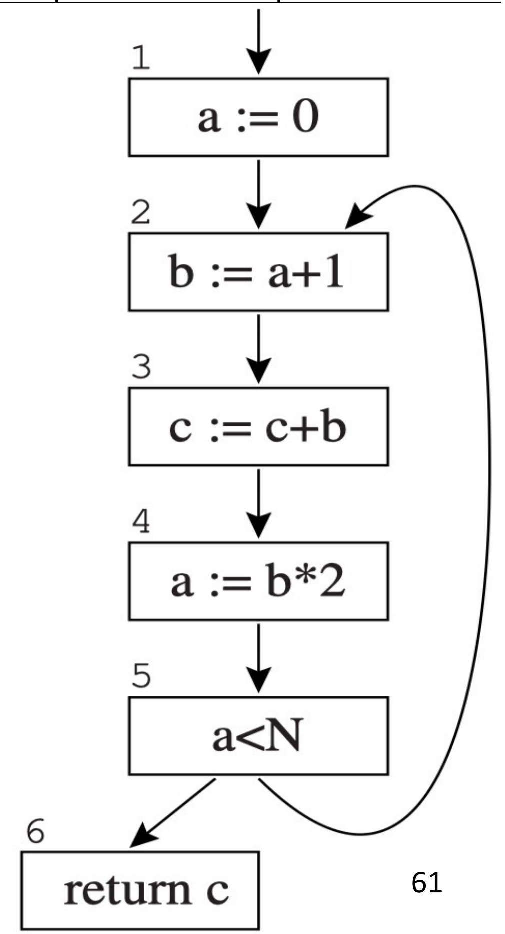
$$in[n] = use[n] \cup (out[n] - def[n])$$

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

Example: Calculation of Liveness

- **Strategy**: following forward control-flow edges

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



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

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

Example: Calculation of Liveness

- **Strategy**: following forward control-flow edges

			1st		2nd		3rd		4th		5th		6th		7th	
	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	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	

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

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

How to Speed Up the Iteration?

- **Observation**: the only way information propagates from one node to another is using: $out[n] := \bigcup_{s \in succ[n]} in[s]$
  - This is the only rule that involves more than one node
  - Liveness analysis is a "backward analysis"

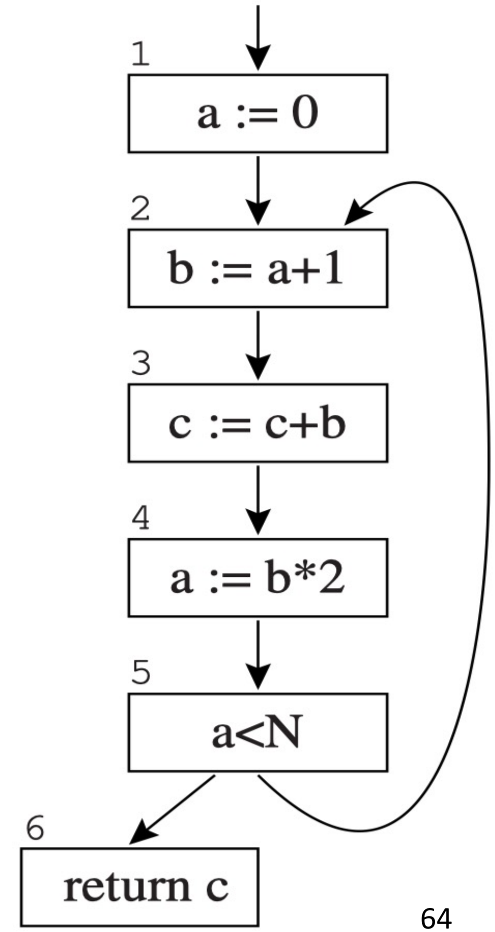
$$in[n] = use[n] \cup (out[n] - def[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

- Idea for an improved version of the algorithm:
  - Keep track of which node's successors have changed
  - Compute in the opposite order of control flows

Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



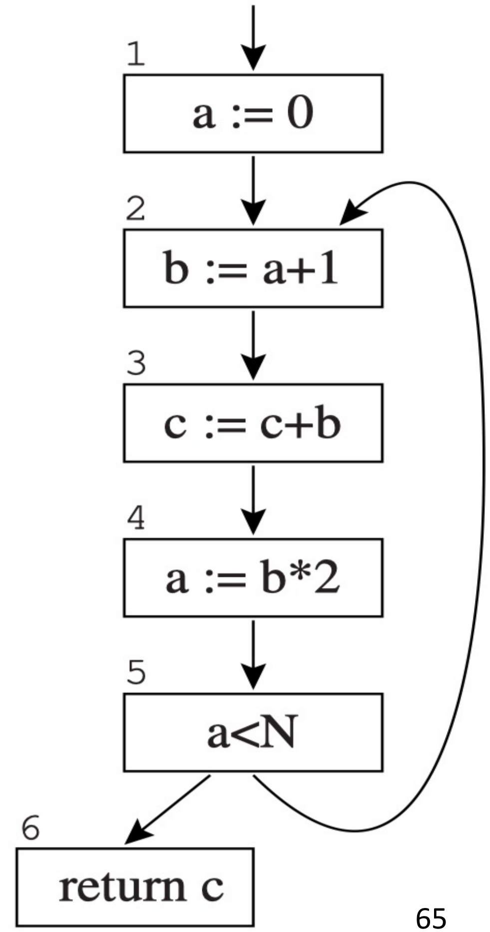
$$in[n] = use[n] \cup (out[n] - def[n])$$

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

Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

			1^{st}		2^{nd}		3^{rd}	
	use	def	out	in	out	in	out	in
6	c							
5	a							
4	b	a						
3	bc	c						
2	a	b						
1		a						



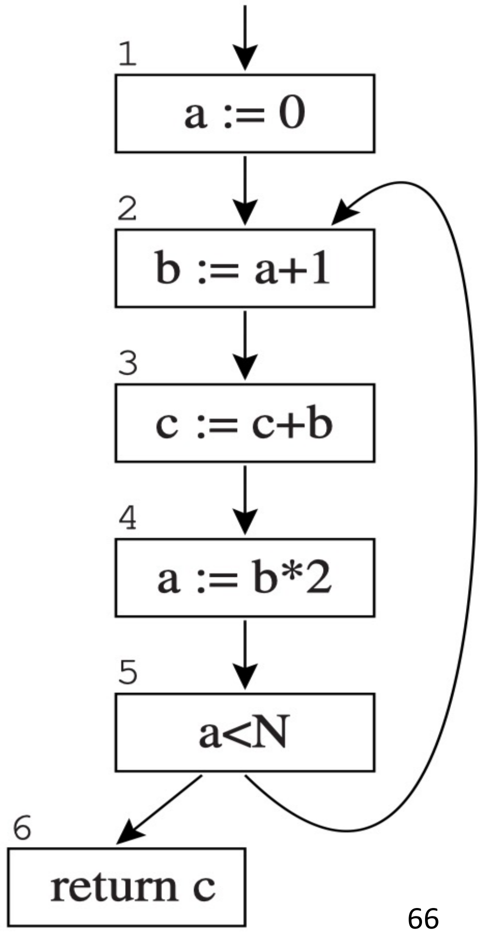
$$in[n] = use[n] \cup (out[n] - def[n])$$

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

Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



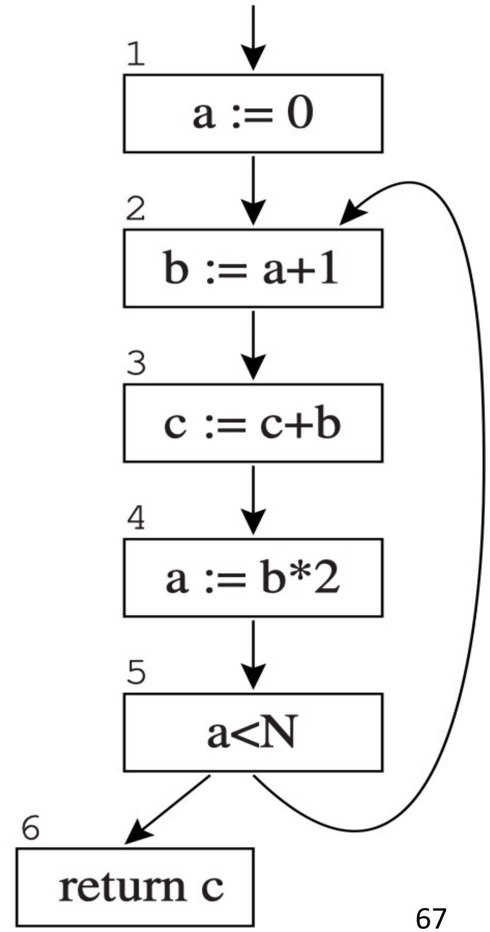
$$\boxed{in[n] = use[n] \cup (out[n] - def[n])}$$

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

Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



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

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

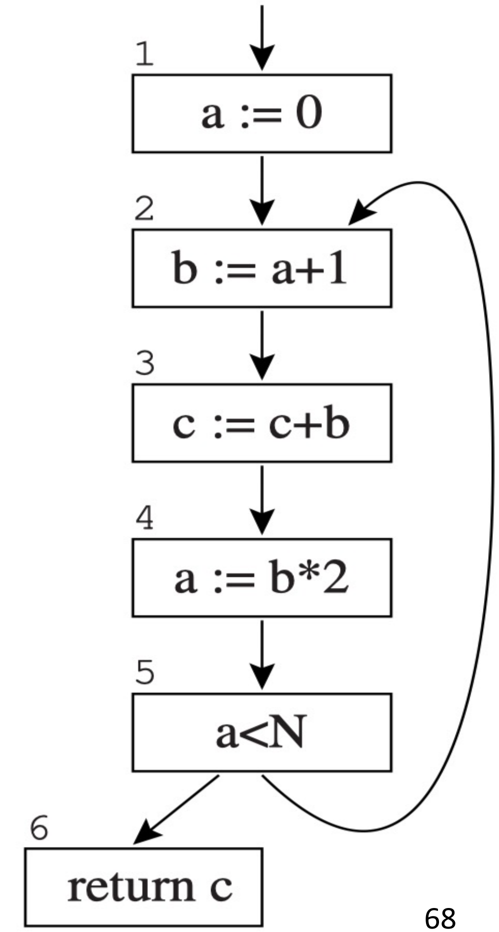
Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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

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

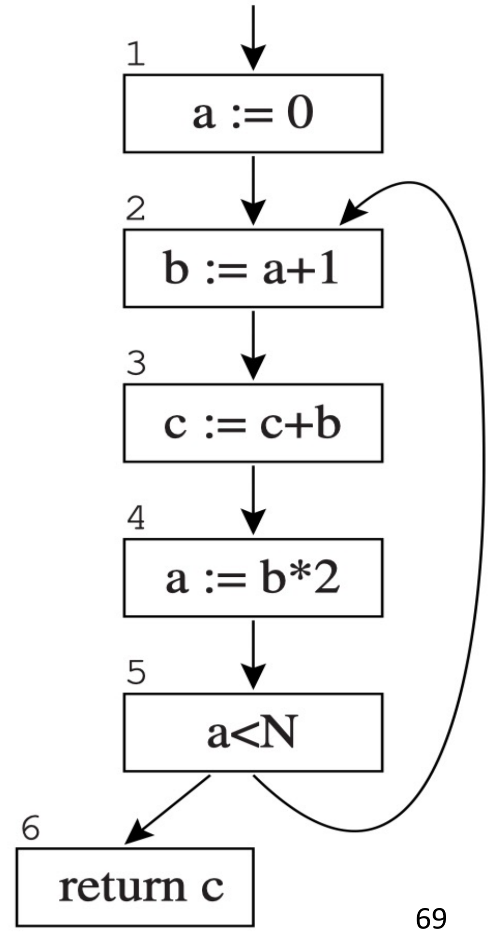
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$



Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



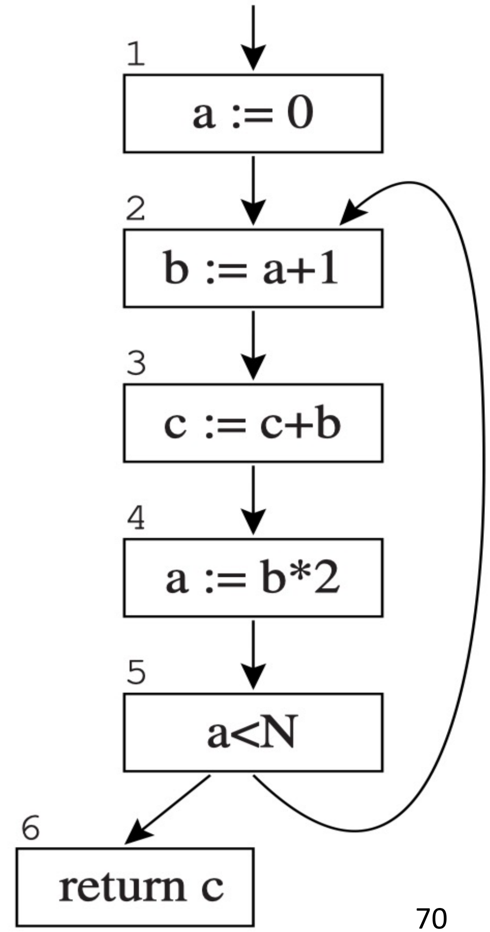
$$in[n] = use[n] \cup (out[n] - def[n])$$

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

Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



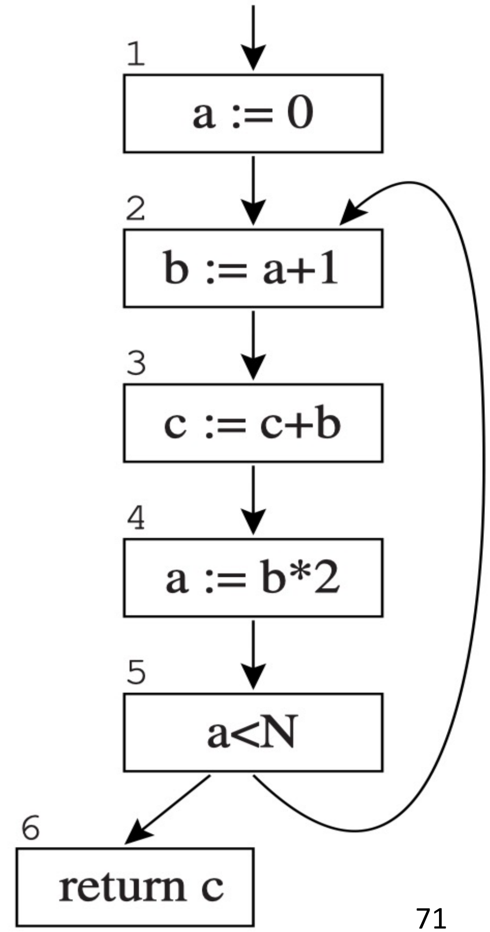
$$\boxed{in[n] = use[n] \cup (out[n] - def[n])}$$

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

Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



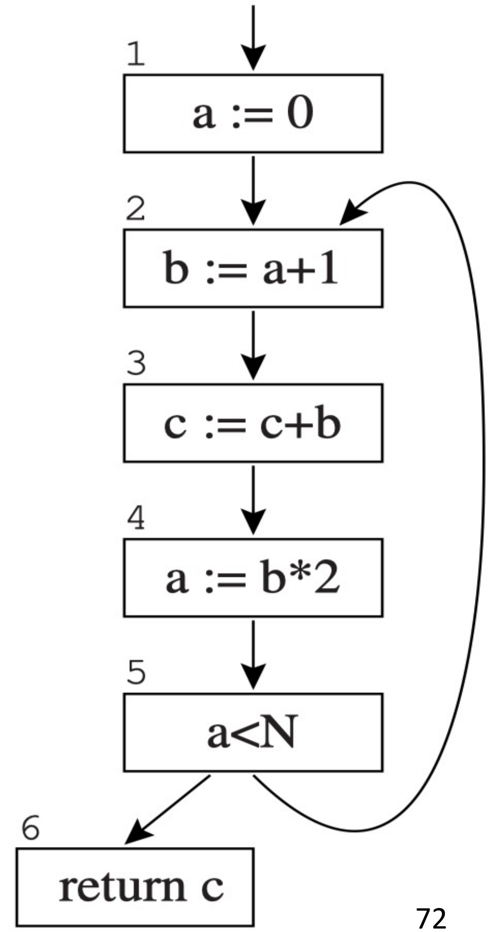
$$in[n] = use[n] \cup (out[n] - def[n])$$

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

Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



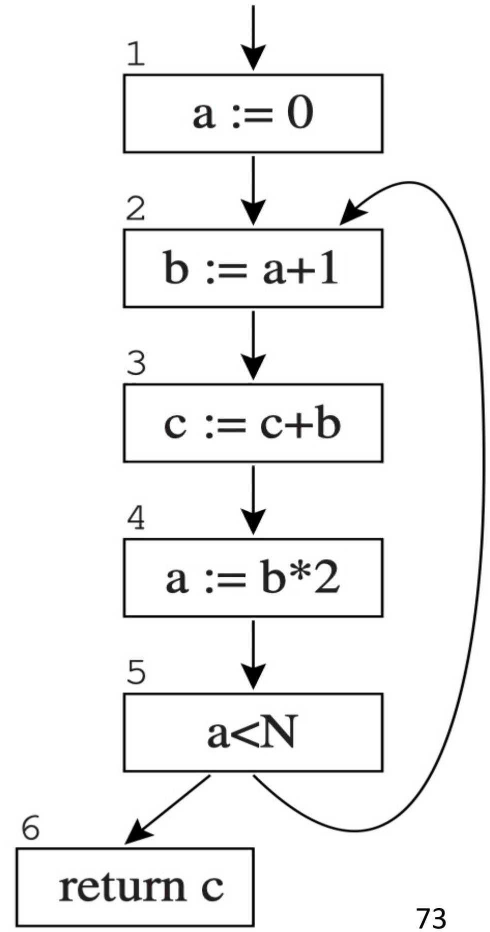
$$\boxed{in[n] = use[n] \cup (out[n] - def[n])}$$

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

Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



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

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

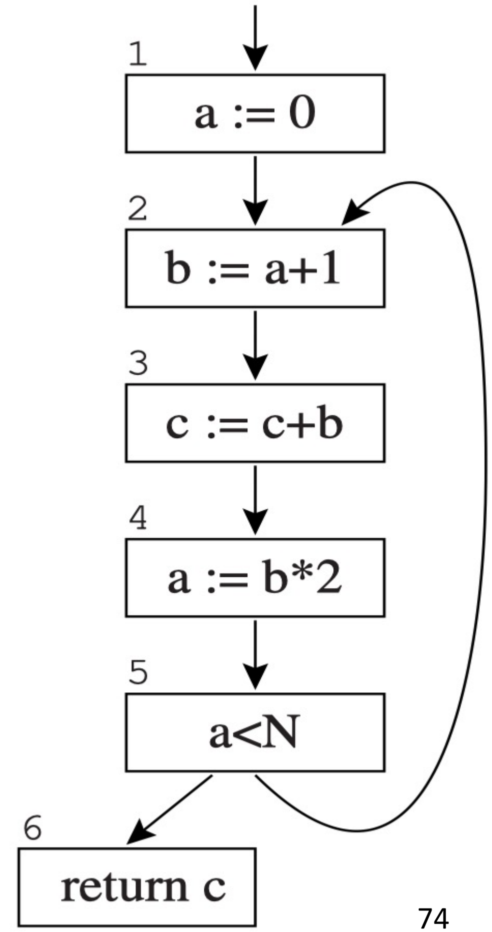
Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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

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

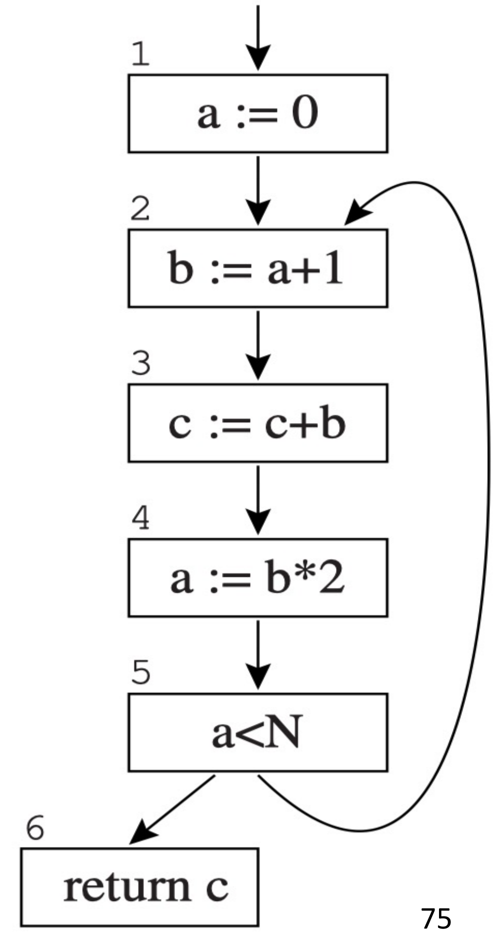
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$



Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



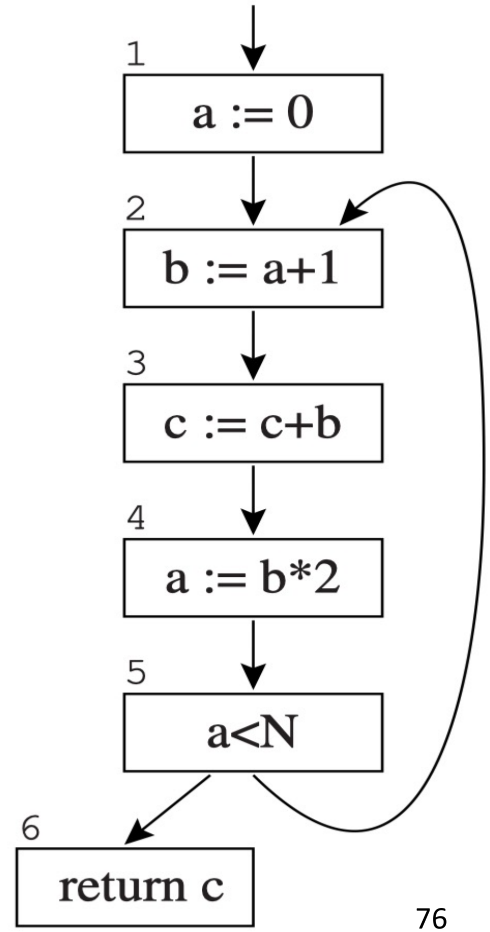
$$in[n] = use[n] \cup (out[n] - def[n])$$

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

Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

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



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

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

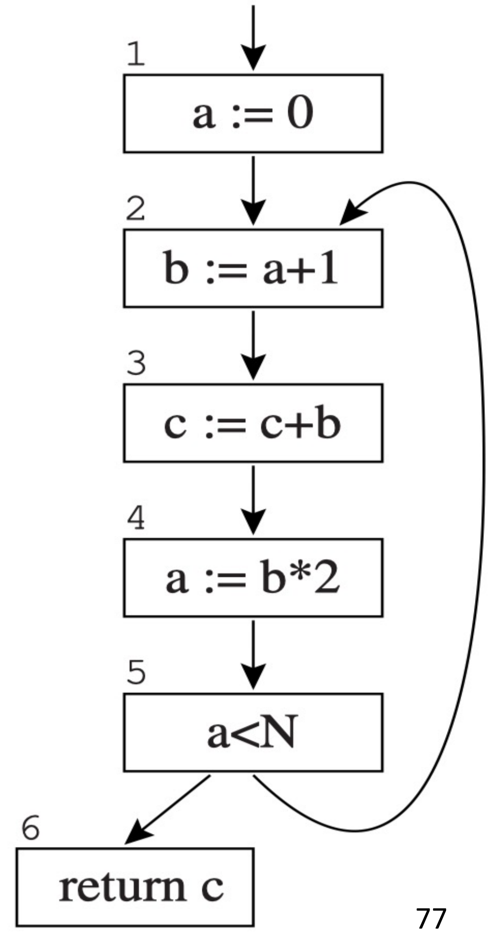
Example: Calculation of Liveness, Revisited

- out[n] is computed from in[s], in[n] is computed from out[n]
- **Strategy**: speed the convergence by computing in the opposite order (from 6 to 1, from out to in)

			1st		2nd		3rd	
	use	def	out	in	out	in	out	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

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

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



Summary: Calculation of Liveness

- Following forward control-flow edges VS. Computing in the opposite order

- When solving dataflow equations by iteration, the order of computation should follow the "**flow**" of dataflow facts

- Liveness flows backward along control-flow arrows, and from "out" to "in", so should the computation.

4. More Discussions

- Improvements
- Theoretical Results
- Static vs. Dynamic Liveness

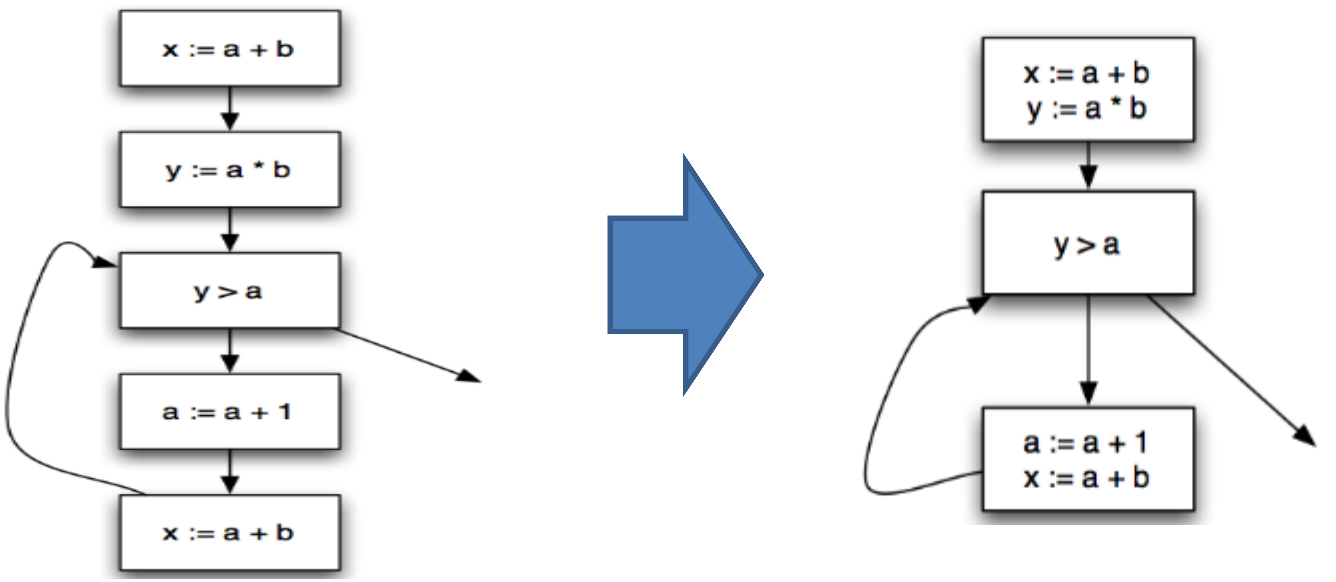
Optimizing the Iterative Solving Process

$$\mathrm{in}[n] = \mathrm{use}[n] \cup (\mathrm{out}[n] - \mathrm{def}[n])$$
$$\mathrm{out}[n] = \bigcup_{s \in \mathrm{succ}[n]} \mathrm{in}[s]$$

- **Ordering the nodes**
- Use variants of Control-flow graph (CFG)
- Once a variable
- Careful selection of set representation
- Use other intermediate representation (IR)
- ….?

Improvements: Use different CFGs

- **Basic blocks**: Flow-graph nodes that have only one predecessor and one successor are not very interesting.
  - Merging them with their predecessors and successors
  - Obtaining a graph with fewer nodes, where each node represents a basic block



Improvements: Variants of the Calculation

- **One variable at a time**: compute dataflow for one variable at a time as information about that variable is needed.

- This is also practical, since many temporaries have very short live ranges.

Improvements: Representations of Sets

- **How to represent in[n] and out[n] for implementation?**
- Two methods: as arrays of bits or as sorted lists of variables
- **Bit Arrays** (for dense set)
  - Suppose: N variables in the program, K bits per word
  - N bits for each set
  - The union of two sets: or-ing the corresponding bits at each position.
  - One set-union operation takes N/K operations.
- **Sorted Lists** (for sparse set)
  - sorted by any totally ordered key (such as variable name)
  - The union operation: merging the lists
- When the sets are sparse (fewer than N/K elements, on the average), the sorted-list representation is faster.

4. More Discussions

- Improvements
- Theoretical Results
- Static vs. Dynamic Liveness

Theorical Results: Decidability

- No compiler can ever fully understand how all the control flow in every program will work.
  - Prove through the halting problem

- **Theorem.** There is no program *H* that takes as input any program *P* and input *X* and (without infinite-looping) returns true if *P(X)* halts and false if *P(X)* infinite-loops.

- **Corollary.** No program *H'(P, L)* can tell, for any program *P* and label L in *P*, whether the label L is ever reached on an execution of *P*.
  - prove by showing that if *H'* exists, then *H* exists.
  - let L be the end of the program, halt => goto L

Theorical Results: Decidability

- This theorem does not mean that we can never tell if a given label is reached or not, just that there is not a **general** algorithm that can **always** tell (precisely)

```
x = y; // is x live here?
f();  // does f halt??
return x;
```

- We could improve out liveness analysis with some special-case algorithms.
- But, no compiler can really tell if a variable's value is truly needed – whether the variable is truly live.
- We have to make do with a conservative approximation.
  – Assume that any conditional branch goes both ways.

Theorical Results: Time Complexity

- **How fast is the iterative dataflow analysis?**
- A program of size N: at most N nodes and at most N variables.
- Each set-union operation takes O(N) time.
- For loop: computes a constant number of set operations per node; there are O(N) nodes => O(N^2) time
- Repeat loop: The sum of the sizes of all in and out sets is 2N^2, which is the most that the repeat loop can iterate
  - **Why?** The in and out sets are monotonic, and cannot keep growing infinitely.
- Worst-case run time:
  - **O(N^4)**
- In practice :
  - **Between O(N) and O(N^2)**
  - Proper computation order

for each n
  in[n] ←{}; out[n] ←{}
repeat
  for each n
    in′[n] ← in[n]; out′[n] ← out[n]
    in[n] ← use[n] ∪ (out[n] − def[n])
    out[n] ← $\bigcup_{s \in succ[n]}$ in[s]
until in′[n] = in[n] and out′[n] = out[n] for all n

Theorical Results: Least Fixed Points

			X		Y		Z	
	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	

TABLE 10.7. X and Y are solutions to the liveness equations; Z is not a solution.

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

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

- Assume there is another program variable d not used in this fragment.
- Any solution to the dataflow equations is a conservation approximation.
  - If a is needed after node n in some execution, we can be assured that a ∊ out[n] in any solution of the equations.
  - But the converse is not true. a ∊ out[n] does not mean its value will really be used.

Acceptable?

Theorical Results: Least Fixed Points

- **Theorem.** Equation 10.3 have more than one solutions
  - X and Y
- If X is the solution of Equation 10.3 and all solutions to Equation 10.3 contains Solution X, we say that X is the least solution (least fixed point) to Equation 10.3.
- Equation 10.3 have a least fixed point and the iteration algorithm described above always computes the least fixed point.

4. More Discussions

- Improvements
- Theoretical Results
- Static vs. Dynamic Liveness

Static and Dynamic Liveness

- **Static liveness (over-approximation)**
  - A variable *a* is statically live at node *n* if there is **some path of control-flow edges** from *n* to some use of *a* that does not go though a definition of *a*.

- **Dynamic liveness (under-approximation)**
  - A variable *a* is dynamically live at node *n* if **some execution** of the program goes from *n* to a use of *a* without going through any definition of *a*.

If a is dynamically live, it is also statically live.

Thank you all for your attention

Worklist Agorithm: Use a FIFO Queue of Nodes that Might Need to be Updated

```
for all n, in[n] := Ø, out[n] := Ø
w = new queue with all nodes
repeat until w is empty:
  let n = w.pop()                            // pull a node off the queue
    old_in = in[n]                           // remember old in[n]
    out[n] := U_{n'∈succ[n]} in[n']
    in[n] := use[n] U (out[n] - def[n])
    if (old_in != in[n]):                    // if in[n] has changed
      for all m in pred[n]: w.push(m)        // add pred to worklist
end
```