Introduction to Abstract Interpretation

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

With some slides from Saman Amarasinghe. Used with permission.

November 4, 2015

Course Recap

What you have learned so far

Operational Semantics

- How will a given program behave on a given input?
- This is the ground truth for any analysis

Types

- Annotations describe properties of the data that can be refered by a variable.
- Easy to describe properties that are global to the execution, but only one variable at a time (at least with the machinery we have seen here)
- Properties are fixed a priori by the type system designer
- Actual analysis is cheap
- Annotations can often be inferred

Program Logics

- Annotations describe properties of the state at a given point in the program.
- Easy to describe complex properties of the overall program state, but messy to describe properties that hold over time
- Logic provides a rich language for properties
- Actual analysis can be expensive
- Annotations are hard to infer

?

Some motivation

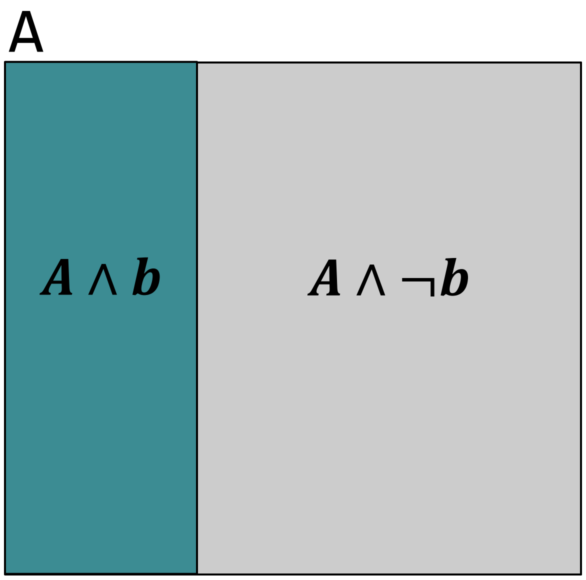
```
{true}
y=0;
while(x<10){
  x = x+1;
  y = y+2;
}
{even(y)}
```

$$\frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} while\ b\ do\ c\ \{A \wedge not\ b\}}$$

What is the loop invariant?

Intuition:

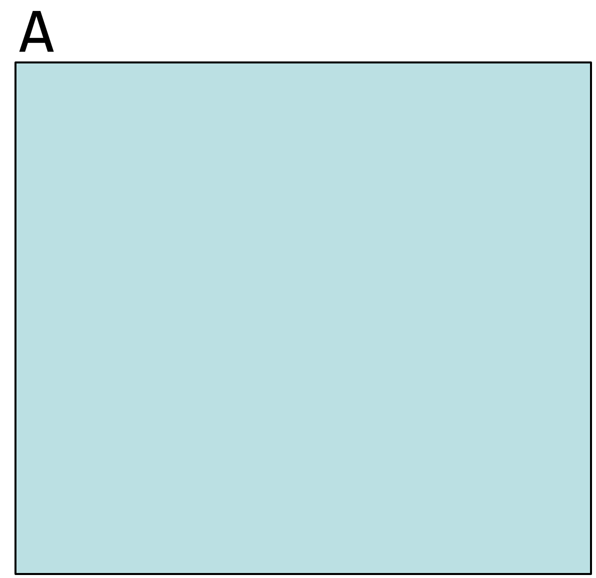
- The loop invariant is a set of states
- C transforms elements in $A \wedge b$ to other elements in A.



Simplifying the problem

```
{true}
y=0;
while(x<10){
  x = x+1;
  y = y+2;
}
{even(y)}
```

$$\frac{\vdash \{A \wedge b\} c \{A\}}{\vdash \{A\} while\ b\ do\ c\ \{A \wedge not\ b\}}$$



This rule is strictly weaker

- Many correct programs can't be proved with it

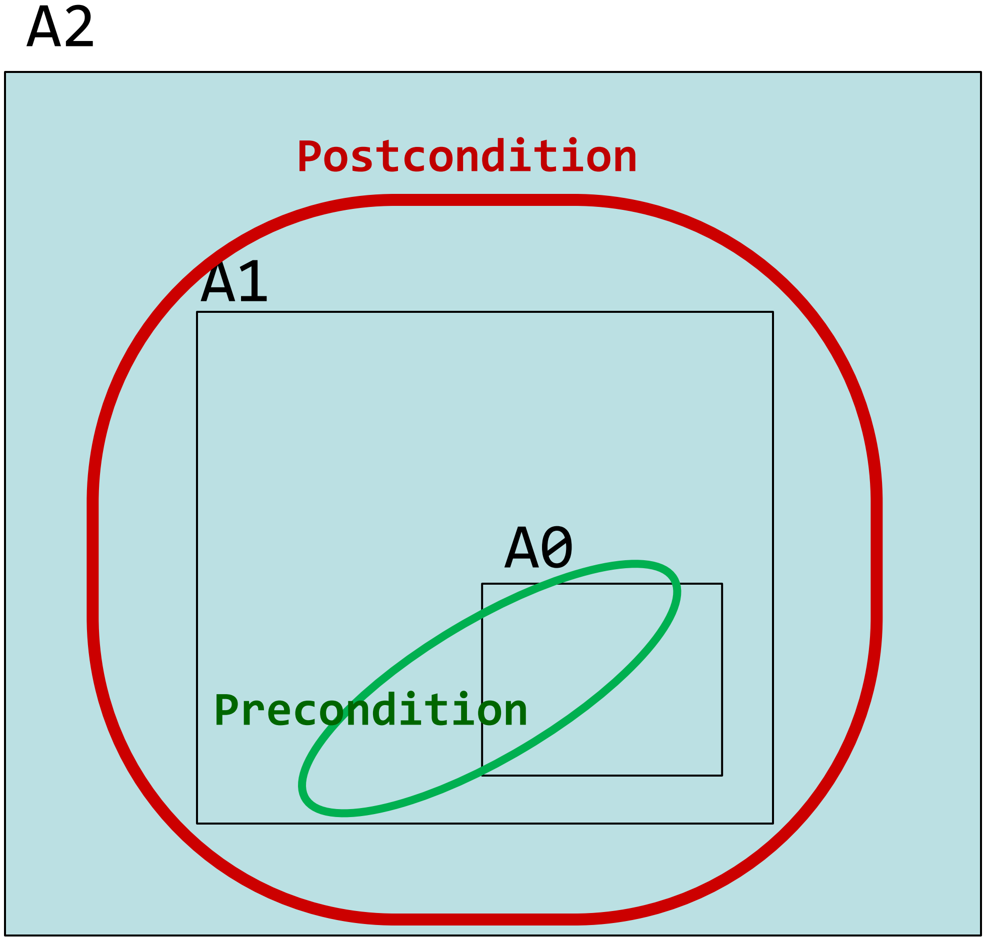
Simpler Intuition:

- The loop invariant is a set of states
- C transforms elements in A to other elements in A.

Discovering the invariant

There may be many candidates for A

- True is always an invariant



Big <=> Weak

Discovering the invariant

We want a set A such that $\dashv \{A\} c \, \{A\}$

- It should be small enough to prove the postcondition (strong)
- But big enough to prove the precondition (weak)

Let $F(P) = \ wpc(c, P) \wedge Post$

- Then what we want is a <u>greatest</u> fixpoint solution of A=F(A)

Convergence properties

- Can we always find such solutions?

Forward vs. Backward

- When is it better to use wpc vs. spc?

Precision

- How do we minimize the loss of precision?

Partial Orders

Set P

Partial order $\leq$ such that $\forall x,y,z \in P$

- $x \leq x$ (reflexive)
- $x \leq y$ and $y \leq x$ implies $x = y$ (asymmetric)
- $x \leq y$ and $y \leq z$ implies $x \leq z$ (transitive)

Can use partial order to define

- Upper and lower bounds
- Least upper bound
- Greatest lower bound

Upper Bounds

If $S \subseteq P$ then

- $x \in P$ is an upper bound of S if $\forall y \in S.\ y \leq x$
- $x \in P$ is the least upper bound of S if
  - x is an upper bound of S, and
  - $x \leq y$ for all upper bounds y of S
- $\vee$ - join, least upper bound, lub, supremum, sup
  - $\vee$ S is the least upper bound of S
  - $x \vee y$ is the least upper bound of $\{x,y\}$
- Often written as $\sqcup$ as well

Lower Bounds

If $S \subseteq P$ then

- $x \in P$ is a lower bound of S if $\forall y \in S.\ x \leq y$
- $x \in P$ is the greatest lower bound of S if
  - x is a lower bound of S, and
  - $y \leq x$ for all lower bounds y of S
- $\wedge$ - meet, greatest lower bound, glb, infimum, inf
  - $\wedge S$ is the greatest lower bound of S
  - $x \wedge y$ is the greatest lower bound of $\{x,y\}$
- Often written as $\sqcap$ as well

Covering

$x < y$ if $x \leq y$ and $x \neq y$

x is covered by y (y covers x) if

- $x < y$, and
- $x \leq z < y$ implies $x = z$

Conceptually,

- y covers x if there are no elements between x and y

Lattices

If $x \wedge y$ and $x \vee y$ exist for all $x,y \in P$

then P is a lattice

If $\wedge S$ and $\vee S$ exist for all $S \subseteq P$

then P is a complete lattice

All finite lattices are complete

Example of a lattice that is not complete

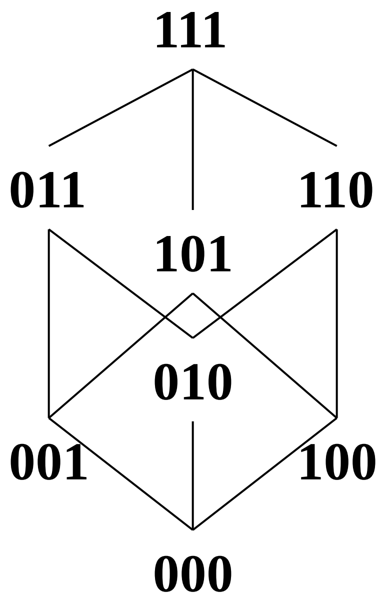
- Integers I
- For any $x, y \in I$, $x \vee y = \max(x,y)$, $x \wedge y = \min(x,y)$
- But $\vee I$ and $\wedge I$ do not exist
- $I \cup \{+\infty, -\infty\}$ is a complete lattice

Example

P = { 000, 001, 010, 011, 100, 101, 110, 111}

(standard boolean lattice, also called hypercube)

x ≤ y if (x bitwise and y) = x



Hasse Diagram

- If y covers x
  - Line from y to x
  - y above x in diagram

Top and Bottom

Greatest element of P (if it exists) is top ($\top$)

Least element of P (if it exists) is bottom ($\perp$)

Connection Between $\leq$, $\wedge$, and $\vee$

The following 3 properties are equivalent:

- $x \leq y$
- $x \vee y = y$
- $x \wedge y = x$

Chains

A set S is a chain if $\forall x,y \in S.\ y \leq x$ or $x \leq y$

P has no infinite chains if every chain in P is finite

Product Latices

Given two latices L and Q, the product can easily be made a latice

$$(l_1, q_1) \sqsubseteq (l_2, q_2) \Leftrightarrow l_1 \sqsubseteq l_2 \; and \; q_1 \sqsubseteq q_2$$

For vectors of L, defining a latice is also easy

$$\langle l_1, l_2, \dots, l_k \rangle \sqsubseteq \langle t_1, t_2, \dots, t_k \rangle \Leftrightarrow \forall_{i \in [1,k]} \, l_i \sqsubseteq t_i$$

Back to our problem

```
{true}
y=0;
while(x<10){
  x = x+1;
  y = y+2;
}
{even(y)}
```

$$x = \begin{cases} \top & \text{Could be odd or even} \\ odd & \text{definitely odd} \\ even & \text{definitely even} \\ \bot & \text{who cares} \end{cases}$$

A latice of predicates:

- <($\mathrm{x} = \bot, even, odd, \top$)>
  - Ex: $\langle x = even, y = odd \rangle \sqsubseteq \langle x = \top, y = odd \rangle$

$$\begin{tikzcd} & \top \arrow[dl] \arrow[dr] & \\ odd \arrow[dr] & & even \arrow[dl] \\ & \bot & \end{tikzcd}$$

What does this have to do with our problem?

Latices and fixpoints

Order Preserving (Monotonic) Function:

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

Now, let $x_\perp$ be the least fixed point of $f: L \to L$

- so $f(x_\perp) = x_\perp$

Now, let $x_0 = \perp$ and $x_i = f(x_{i-1})$

- By induction, $x_i \sqsubseteq x_\perp$
- Also, the chain x_i is an ascending chain
- If L has no infinite ascending chains, sooner or later $x_i = x_{i+1} = x_\perp$

Same trick works for greatest fixed point!

- But then you have to start with $x_0 = \top$

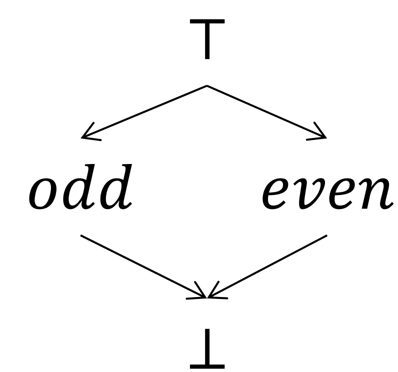
Back to our problem

```
{true}
y=0;
while(x<10){
  x = x+1;
  y = y+2;
}
{even(y)}
```

$$x = \begin{cases} \top & \text{Could be odd or even} \\ odd & \text{definitely odd} \\ even & \text{definitely even} \\ \bot & \text{who cares} \end{cases}$$

A latice of predicates:

- $<(x = \bot, even, odd, \top)>$
  - Ex: $\langle x = even, y = odd \rangle \sqsubseteq \langle x = \top, y = odd \rangle$



We now have a recipe to find a greatest fixpoint solution

- As long as $F(P) = wpc(c, P) \wedge Post$ is monotonic in our latice

Finding a fixpoint

```
{x = ⊤, y = ⊤}
y=0;
while(x<10){
  x = x+1;
  y = y+2;
}
{x = ⊤, y = even}
```

$$x = \begin{cases} \top & \text{Could be odd or even} \\ odd & \text{definitely odd} \\ even & \text{definitely even} \\ \bot & \text{who cares} \end{cases}$$

$$F(P) = wpc(c, P) \wedge Post$$

- $P_0 = \{x = \top, y = \top\}$
- $P_1 = \{x = \top, y = even\}$
- $P_2 = \{x = \top, y = even\}$
- Success!

Complicating things a bit

$\{\boldsymbol{x} = \top, \boldsymbol{y} = \top\}$

```
y=0; t=1;            } C0
while(x<10){
  x = x+1;           } C1
  y = y+2;           }
  if(x=5){
     t=t+2;          } C2
  }else{
     y = t+1;        } C3
  }
}
```

$\{\boldsymbol{x} = \top, \boldsymbol{y} = \boldsymbol{even}\}$

$$\frac{\vdash \{A \wedge b\} c_1 \{B\} \qquad \vdash \{A \wedge not\ b\} c_2 \{B\}}{\vdash \{A\} if\ b\ then\ c_1 else\ c_2\ \{B\}}$$

Relaxed Rule

$$\frac{\vdash \{A \wedge b\} c_1 \{B\} \qquad \vdash \{A \wedge not\ b\} c_2 \{B\}}{\vdash \{A\} if\ b\ then\ c_1 else\ c_2\ \{B\}}$$

$$\begin{aligned} F(P) &= wpc(c, P) \wedge Post \\ &= wpc\big(c1, wpc(c2, P) \wedge wpc(c3, P)\big) \wedge Post \end{aligned}$$

Dataflow equations

```
Big <=> Weak
```
So $A \Rightarrow B$
is equivalent to
$A \sqsubseteq B$

$\{x = \top, y = \top\}$ <-P1
y=0; t=1; } C0
while(x<10){ <-P2
  x = x+1;
  y = y+2; } C1
  if(x=5){ <-P3
    t=t+2; } C2
  }else{
    y = t+1; } C3
  }
<-P2
}
$\{x = \top, y = even\}$ <-P5

$$F(P) = wpc(c, P) \wedge Post$$
$$= wpc\big(c1, wpc(c2, P) \wedge wpc(c3, P)\big) \wedge Post$$

$$p1 \sqsubseteq wpc(c0, p2)$$

$$\boxed{p2 \sqsubseteq wpc(c1, p3)}$$

$$p3 \sqsubseteq wpc(c2, p2) \wedge wpc(c3, p2)$$

$$\boxed{p2 \sqsubseteq p5}$$

$$p2 \sqsubseteq wpc(c1, p3) \wedge p5$$

Dataflow equations

$\{x = \top, y = \top\}$ <-P1

```
y=0; t=1;              C0
while(x<10){           <-P2
  x = x+1;
  y = y+2;             C1
  if(x=5){             <-P3
     t=t+2;            C2
  }else{
     y = t+1;          C3
  }
                       <-P2
}
```

$\{x = \top, y = even\}$ <-P5

$$p1 \sqsubseteq wpc(c0, p2)$$

$$p2 \sqsubseteq wpc(c1, p3) \wedge p5$$

$$p3 \sqsubseteq wpc(c2, p2) \wedge wpc(c3, p2)$$

Dataflow Analysis

General Analysis Framework

- Developed by Kildall in 1973
- Traditionally used for compiler optimization

Frame analysis question as a set of equations on a CFG

Control Flow Graph

$\{x = \top, y = \top\}$ <-P1

y=0; t=1;

while(x<10){ <-P2

  x = x+1;

  y = y+2;

  if(x=5){ <-P3

    t=t+2;

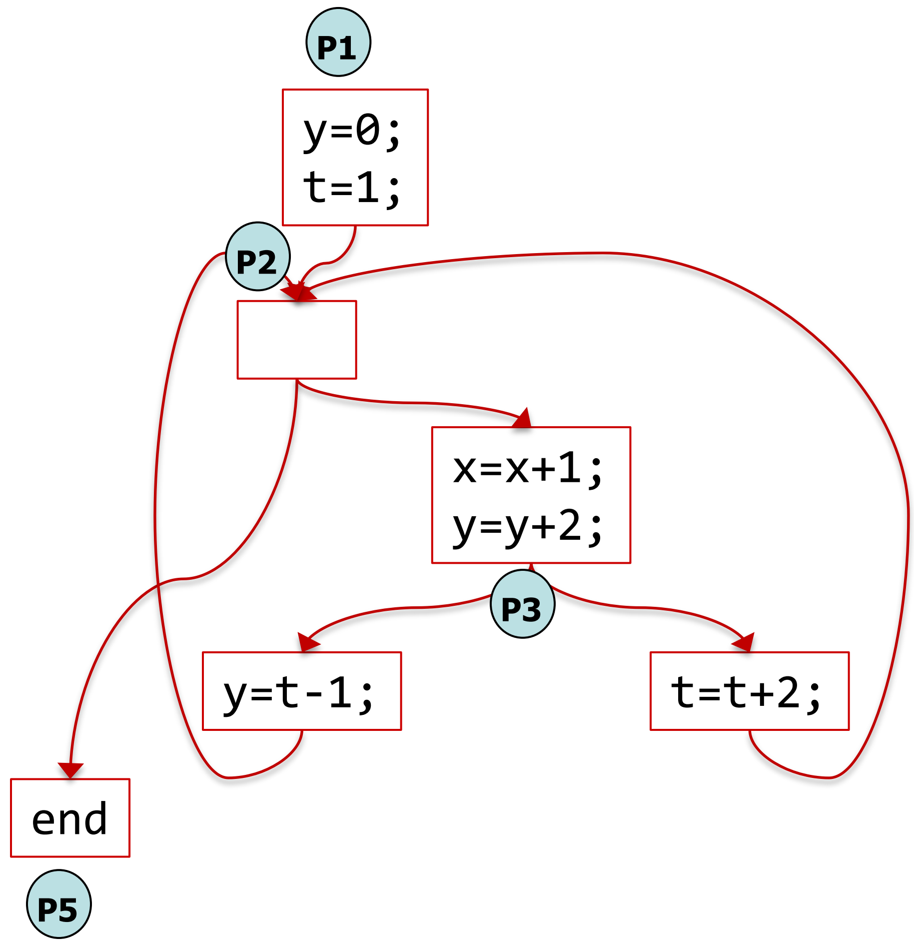
  }else{

    y = t+1;

  } <-P2

}

$\{x = \top, y = even\}$ <-P5



Control Flow Graph

Very general program representation

- Easy to represent unstructured control flow
- Widely used by most program analysis tools for imperative languages

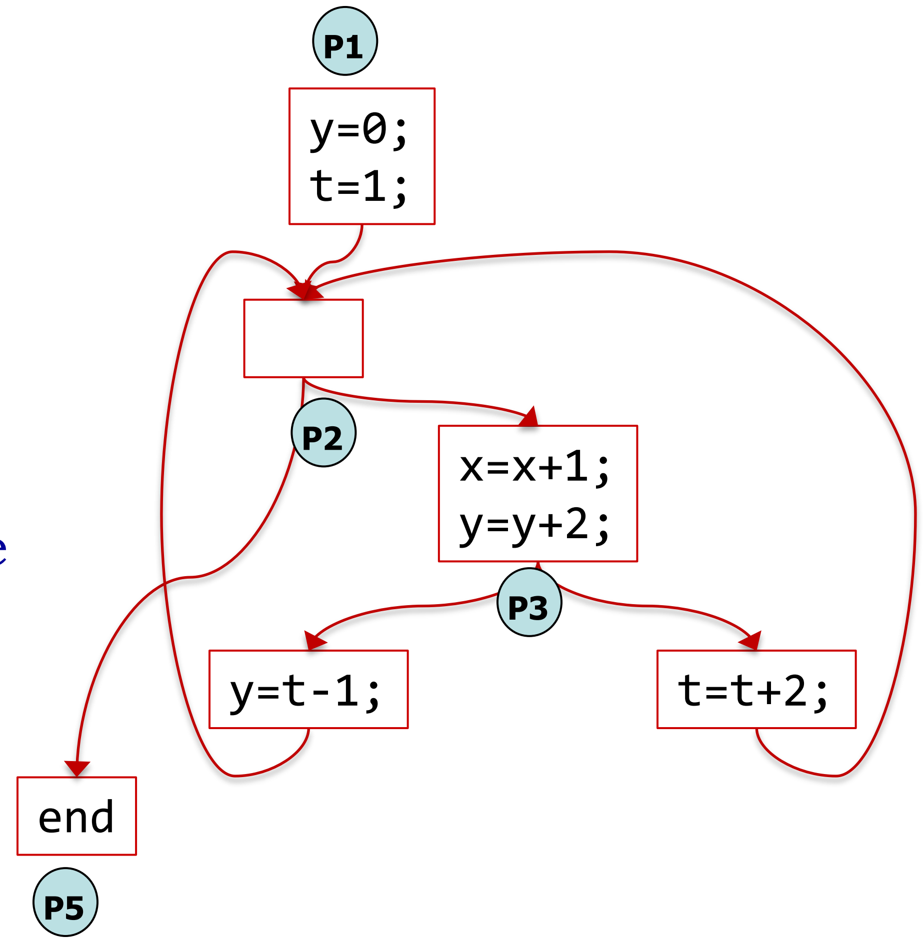
Solution strategy

For every basic block we have an equation of the form

- $Out \sqsubseteq F(in)$
- Use meet (∧) when many edges meet together

We can solve through "Chaotic Iteration"

- Keep a list of nodes to update
- Pick one CFG node at a time
- Update *out* from new *in*
- If out changed, add its children to the list



Computing transfer function

So far we defined it in terms of weakest precondition.

- Or alternatively, strongest postcondition
- Too general and expensive!

We can hard-code a transfer function specific to the lattice

- For finite lattices they can be implemented cheaply in terms of bitvector operations

We can build lattices for arbitrary facts about the program

- Need to make sure our transfer functions are monotonic

Example: Reaching Definitions

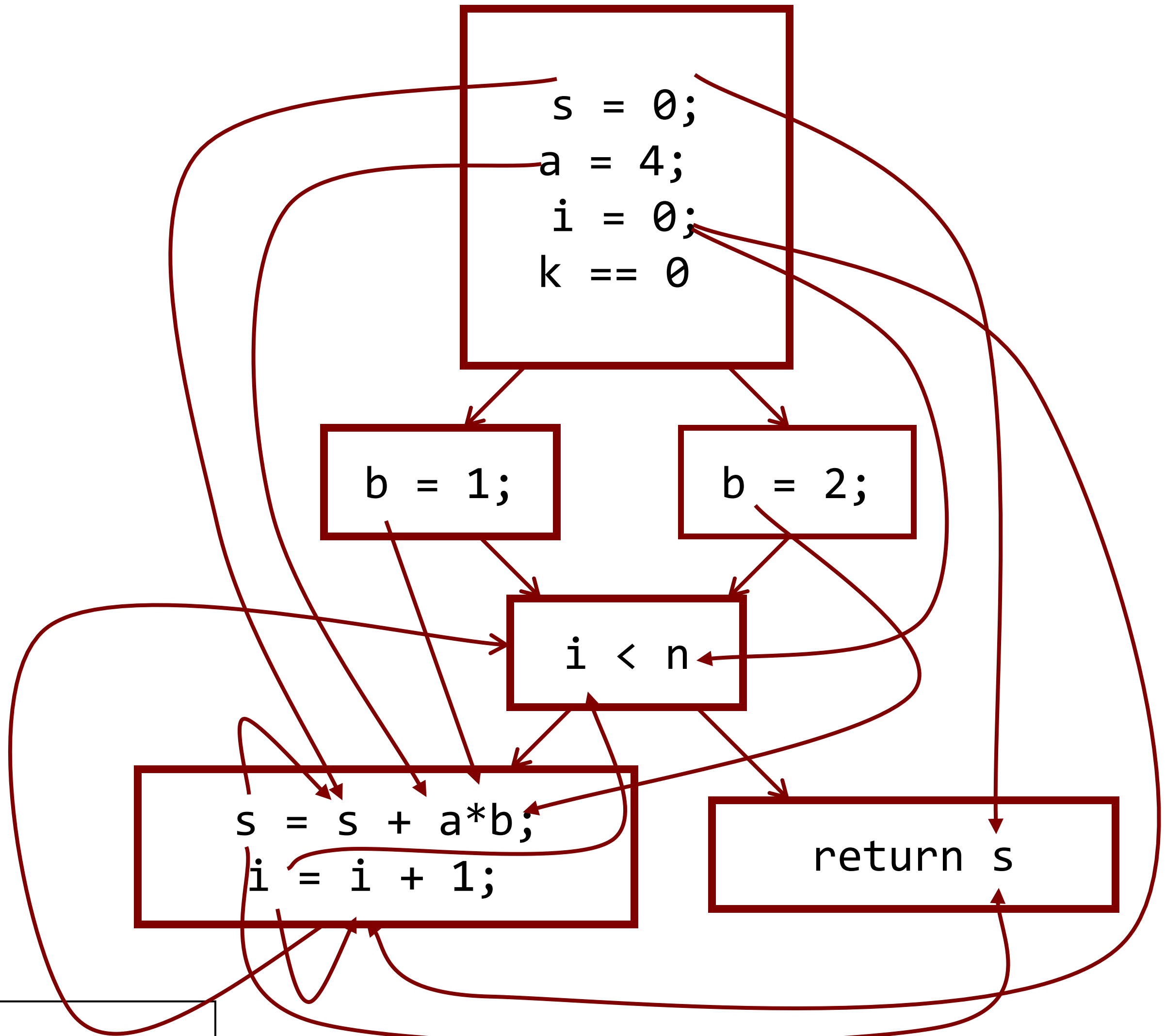
Concept of definition and use

- a = x+y
- is a definition of a
- is a use of x and y

A definition reaches a use if

- value written by definition
- may be read by use

Reaching Definitions



Example by
Saman Amarasinghe

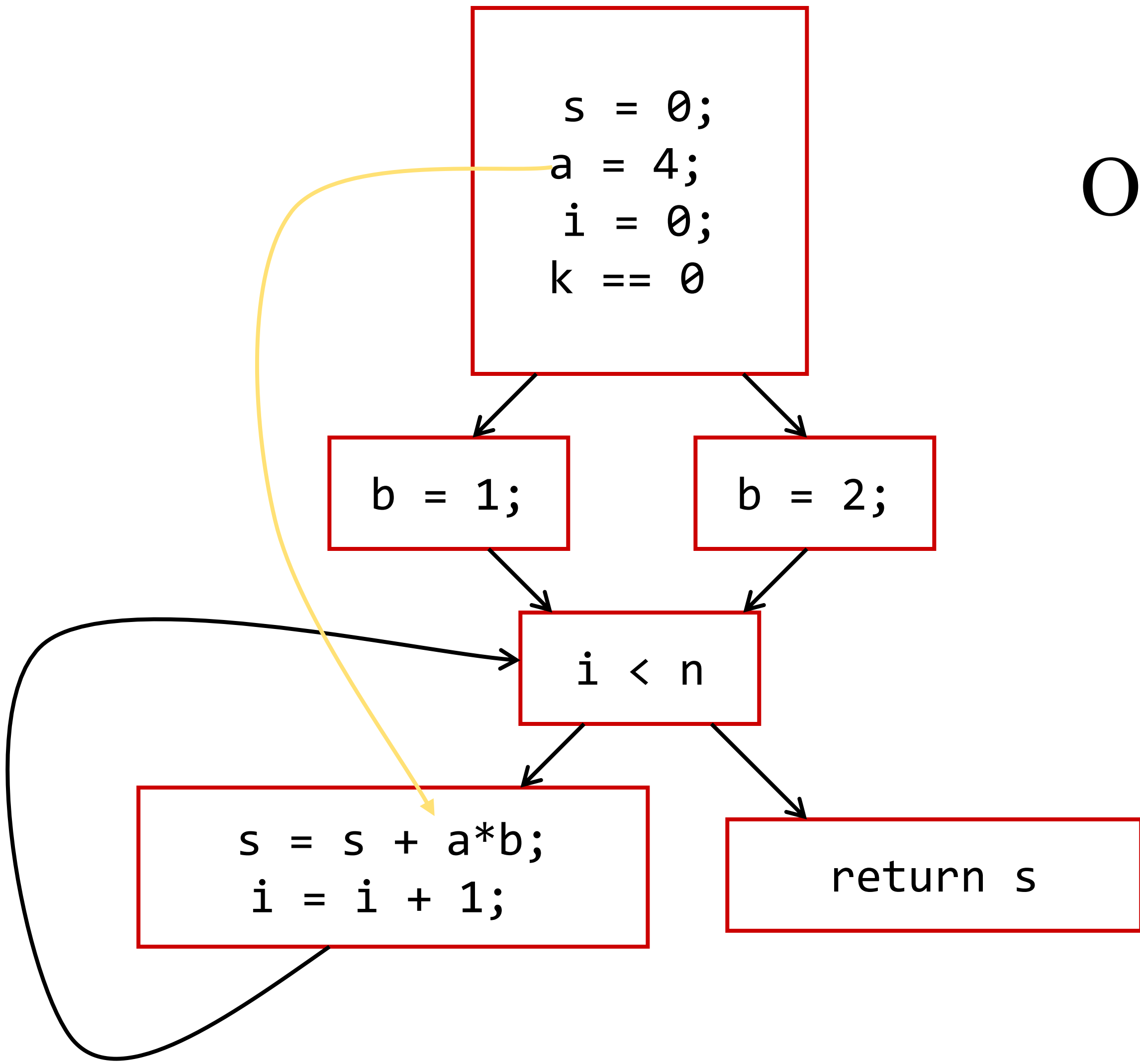
Reaching Definitions and Constant Propagation

Is a use of a variable a constant?

- Check all reaching definitions
- If all assign variable to same constant
- Then use is in fact a constant

Can replace variable with constant

Is a Constant in s = s+a*b?

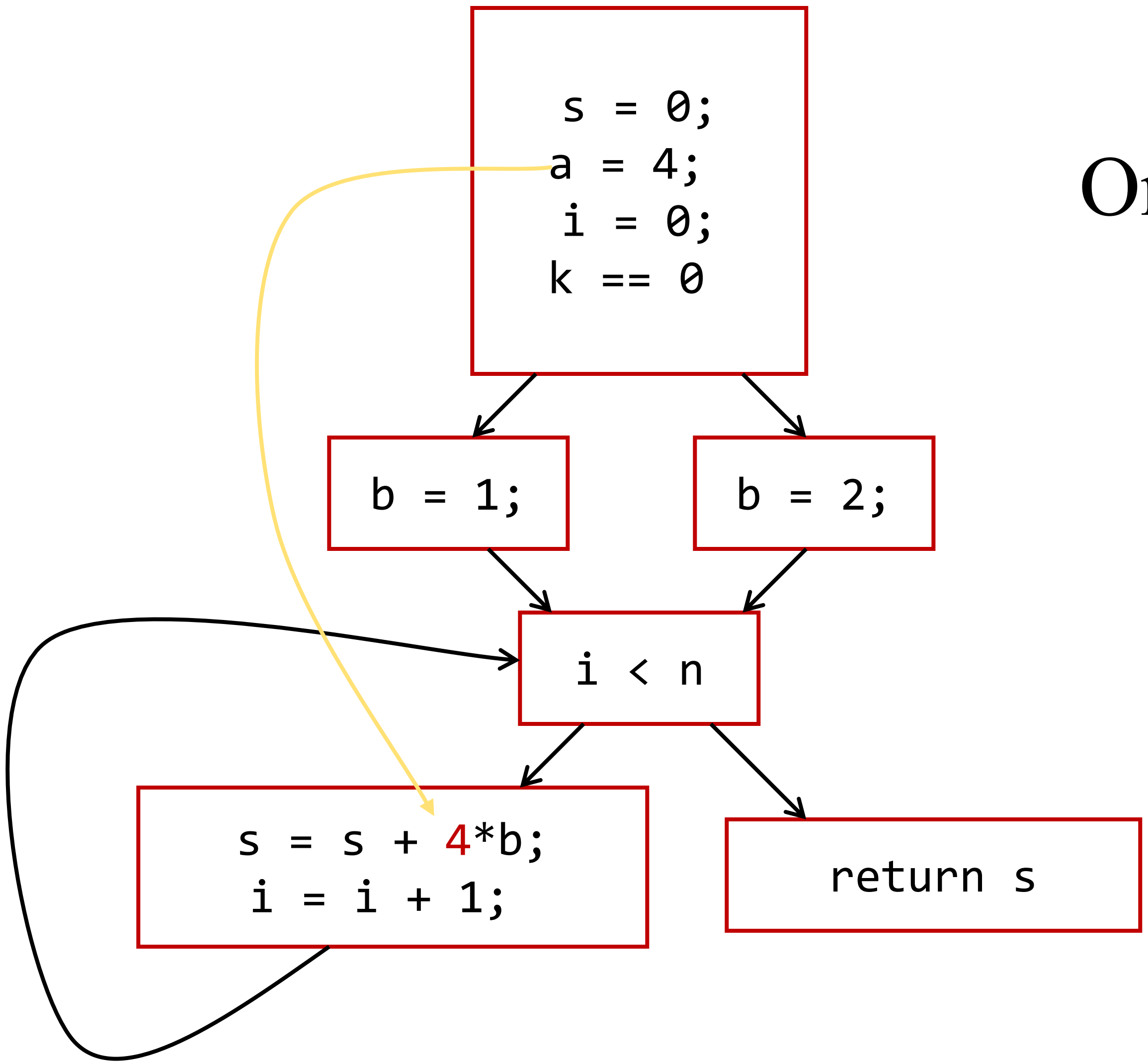


Yes!

On all reaching definitions

a = 4

Constant Propagation Transform

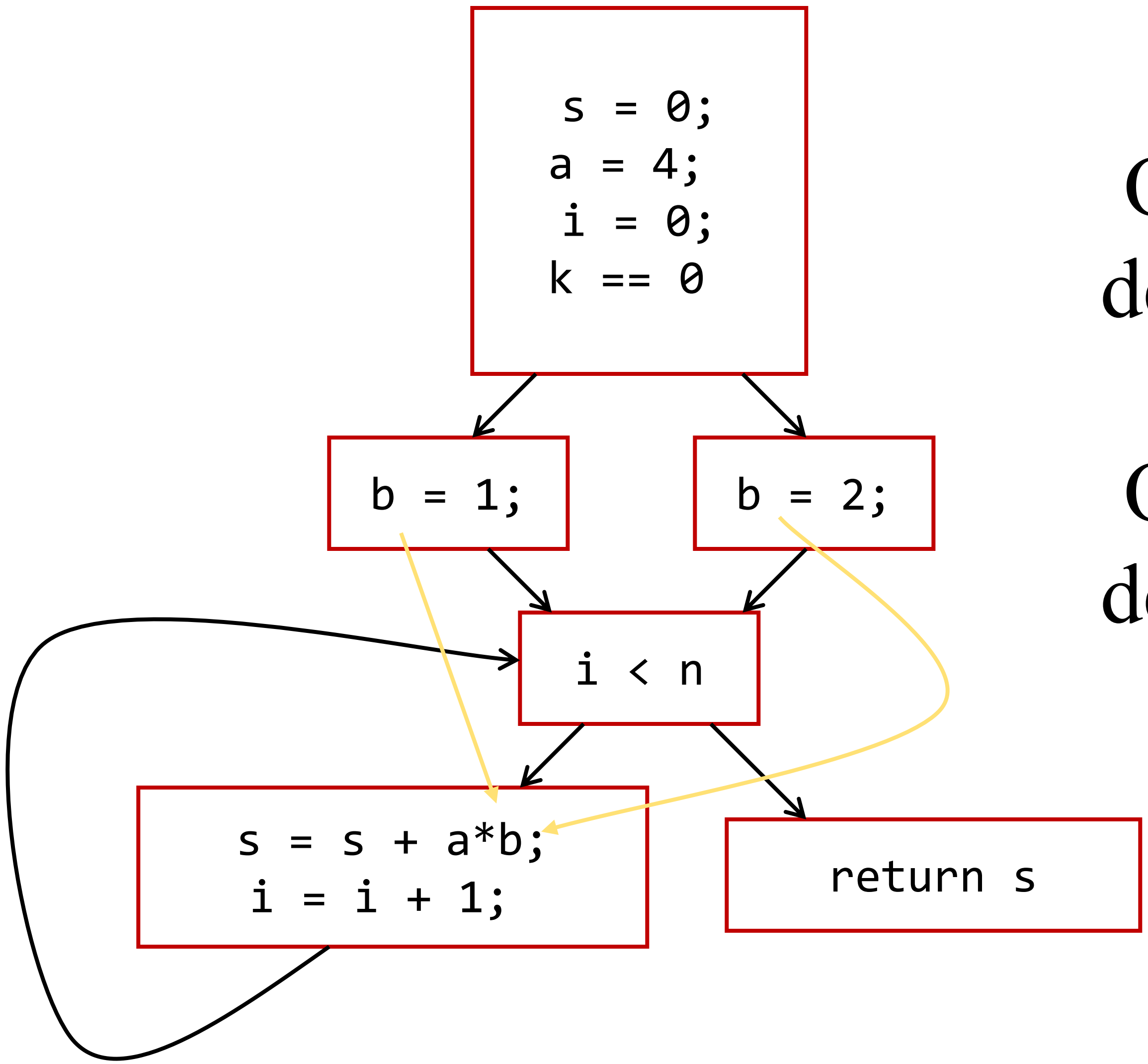


Yes!

On all reaching definitions

a = 4

Is b Constant in s = s+a*b?



No!

One reaching definition with

b = 1

One reaching definition with

b = 2

Computing Reaching Definitions

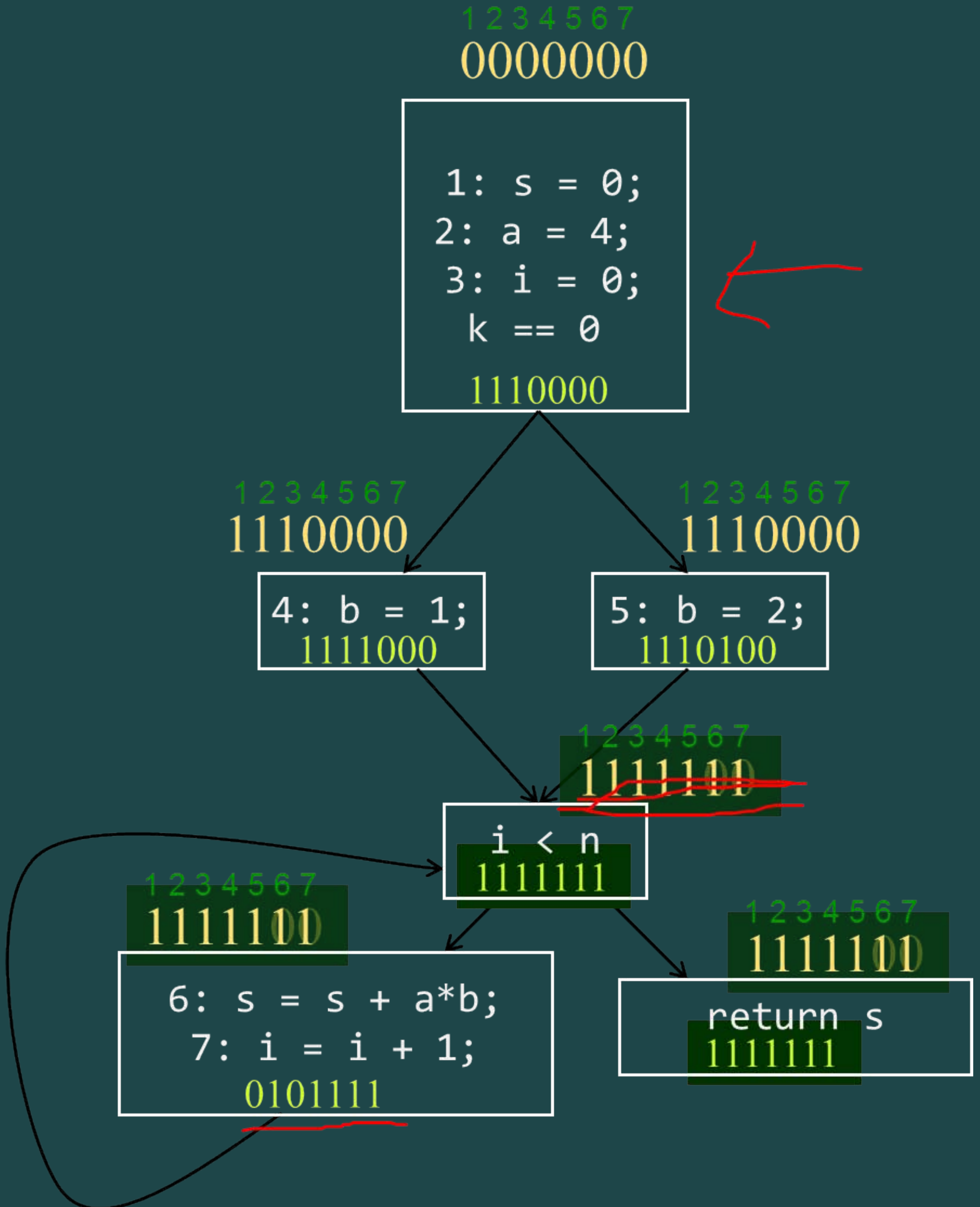
Compute with sets of definitions

- represent sets using bit vectors
- each definition has a position in bit vector

At each basic block, compute

- definitions that reach start of block
- definitions that reach end of block

Do computation by simulating execution of program until reach fixed point


1 2 3 4 5 6 7
0000000
1: s = 0;
2: a = 4;
3: i = 0;
k == 0
1110000
1 2 3 4 5 6 7
1110000
4: b = 1;
1111000
1 2 3 4 5 6 7
1110000
5: b = 2;
1110100
1 2 3 4 5 6 7
1111111
i < n
1111111
1 2 3 4 5 6 7
1111111
6: s = s + a*b;
7: i = i + 1;
0101111
1 2 3 4 5 6 7
1111111
return s
1111111

Transfer functions

Each basic block has

- IN - set of definitions that reach beginning of block
- OUT - set of definitions that reach end of block
- GEN - set of definitions generated in block
- KILL - set of definitions killed in block

GEN[s = s + a*b; i = i + 1;] = 0000011

KILL[s = s + a*b; i = i + 1;] = 1010000

Analyzer scans each basic block to derive GEN and KILL sets for each function

Dataflow Equations

IN[b] = OUT[b1] U ... U OUT[bn]

- where b1, ..., bn are predecessors of b in CFG

OUT[b] = (IN[b] - KILL[b]) U GEN[b]

IN[entry] = 0000000

Result: system of equations

Solving Equations

Use fixed point algorithm

Initialize with solution of OUT[b] = 0000000

Repeatedly apply equations

- IN[b] = OUT[b1] U ... U OUT[bn]
- OUT[b] = (IN[b] - KILL[b]) U GEN[b]

Until reach fixed point

Until equation application has no further effect

Use a worklist to track which equation applications may have a further effect

Questions

Does the algorithm halt?

- yes, because transfer function is monotonic
- if increase IN, increase OUT
- in limit, all bits are 1

If bit is 0, does the corresponding definition ever reach basic block?

If bit is 1, is does the corresponding definition always reach the basic block?

6.820 Fundamentals of Program Analysis
Fall 2015