Introduction to Program Analysis

Uday Khedker

(www.cse.iitb.ac.in/~uday)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

Dec 2017

Introduction to Program Analysis: An Outline

- Motivating example of improving garbage collection
- Soundness and precision of program analysis

What is Program Analysis?

Discovering information about a given program

What is Program Analysis?

Discovering information about a given program

- Representing the dynamic behaviour of the program

What is Program Analysis?

Discovering information about a given program

- Representing the dynamic behaviour of the program
- Most often obtained without executing the program
  - Static analysis Vs. Dynamic Analysis
  - Example of loop tiling for parallelization

What is Program Analysis?

Discovering information about a given program

- Representing the dynamic behaviour of the program
- Most often obtained without executing the program
  - Static analysis Vs. Dynamic Analysis
  - Example of loop tiling for parallelization
- Must represent all execution instances of the program

Why is it Useful?

- Code optimization
  - Improving time, space, energy, or power efficiency
  - Compilation for special architecture (eg. multi-core)

Why is it Useful?

- Code optimization
  - Improving time, space, energy, or power efficiency
  - Compilation for special architecture (eg. multi-core)
- Verification and validation

  Giving guarantees such as: The program will
  - never divide a number by zero
  - never dereference a NULL pointer
  - close all opened files, all opened socket connections
  - not allow buffer overflow security violation

Why is it Useful?

- Code optimization
  - Improving time, space, energy, or power efficiency
  - Compilation for special architecture (eg. multi-core)
- Verification and validation

  Giving guarantees such as: The program will
  - never divide a number by zero
  - never dereference a NULL pointer
  - close all opened files, all opened socket connections
  - not allow buffer overflow security violation
- Software engineering
  - Maintenance, bug fixes, enhancements, migration
  - Example: Y2K problem

Why is it Useful?

- Code optimization
  - Improving time, space, energy, or power efficiency
  - Compilation for special architecture (eg. multi-core)
- Verification and validation

  Giving guarantees such as: The program will
  - never divide a number by zero
  - never dereference a NULL pointer
  - close all opened files, all opened socket connections
  - not allow buffer overflow security violation
- Software engineering
  - Maintenance, bug fixes, enhancements, migration
  - Example: Y2K problem
- Reverse engineering

  To understand the program

Part 1

Program Analysis for Improving Garbage Collection

Garbage Collection ≡ Automatic Deallocation

- Retain active data structure.

  Deallocate inactive data structure.

- What is an Active Data Structure?

Garbage Collection ≡ Automatic Deallocation

- Retain active data structure.
  Deallocate inactive data structure.

- What is an Active Data Structure?

  If an object does not have an access path, (i.e. it is unreachable)
  then its memory can be reclaimed.

Garbage Collection ≡ Automatic Deallocation

- Retain active data structure.
  Deallocate inactive data structure.

- What is an Active Data Structure?

  If an object does not have an access path, (i.e. it is unreachable)
  then its memory can be reclaimed.

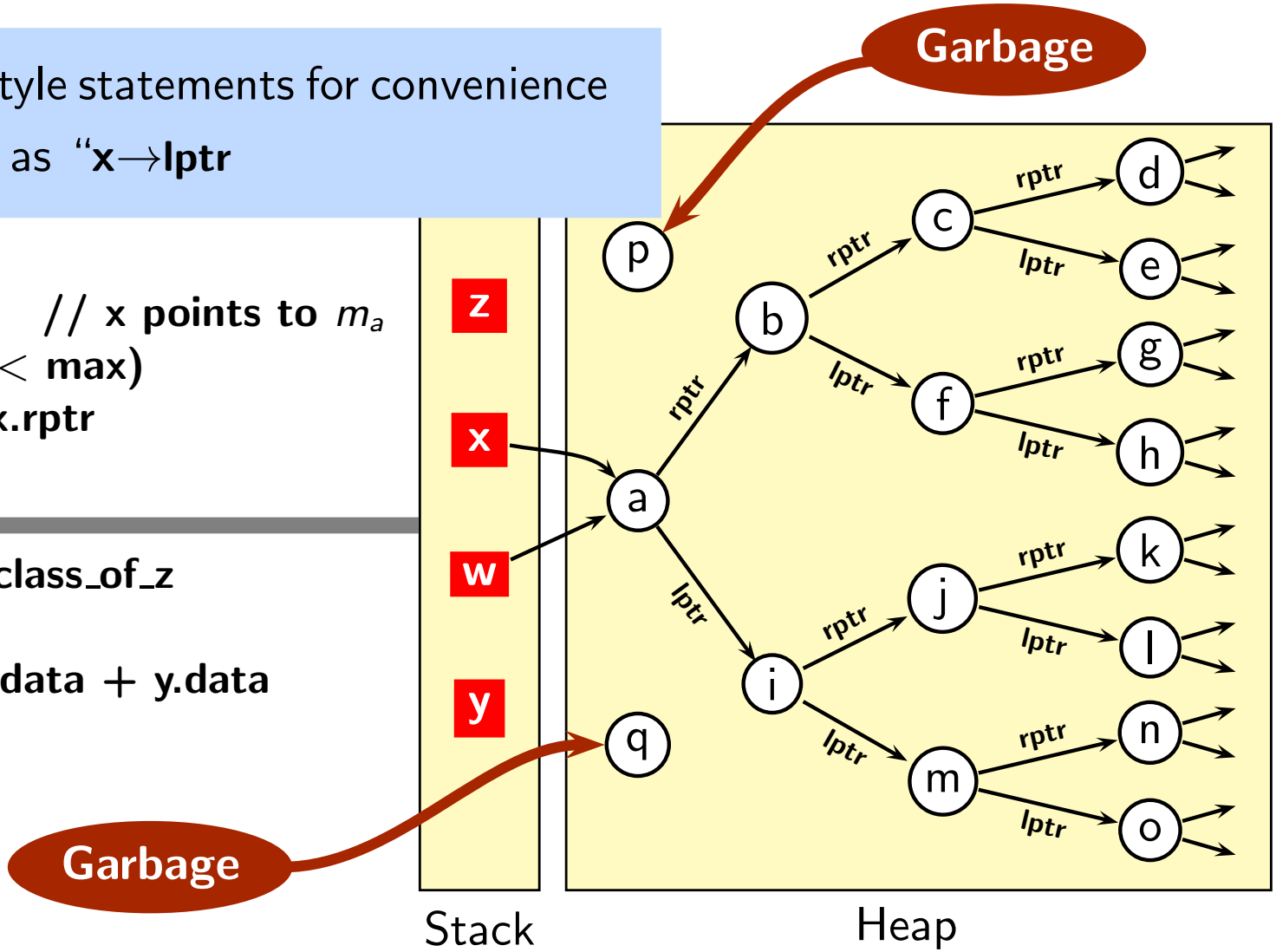
What if an object has an access path, but is not accessed after the given program point?

What is Garbage?

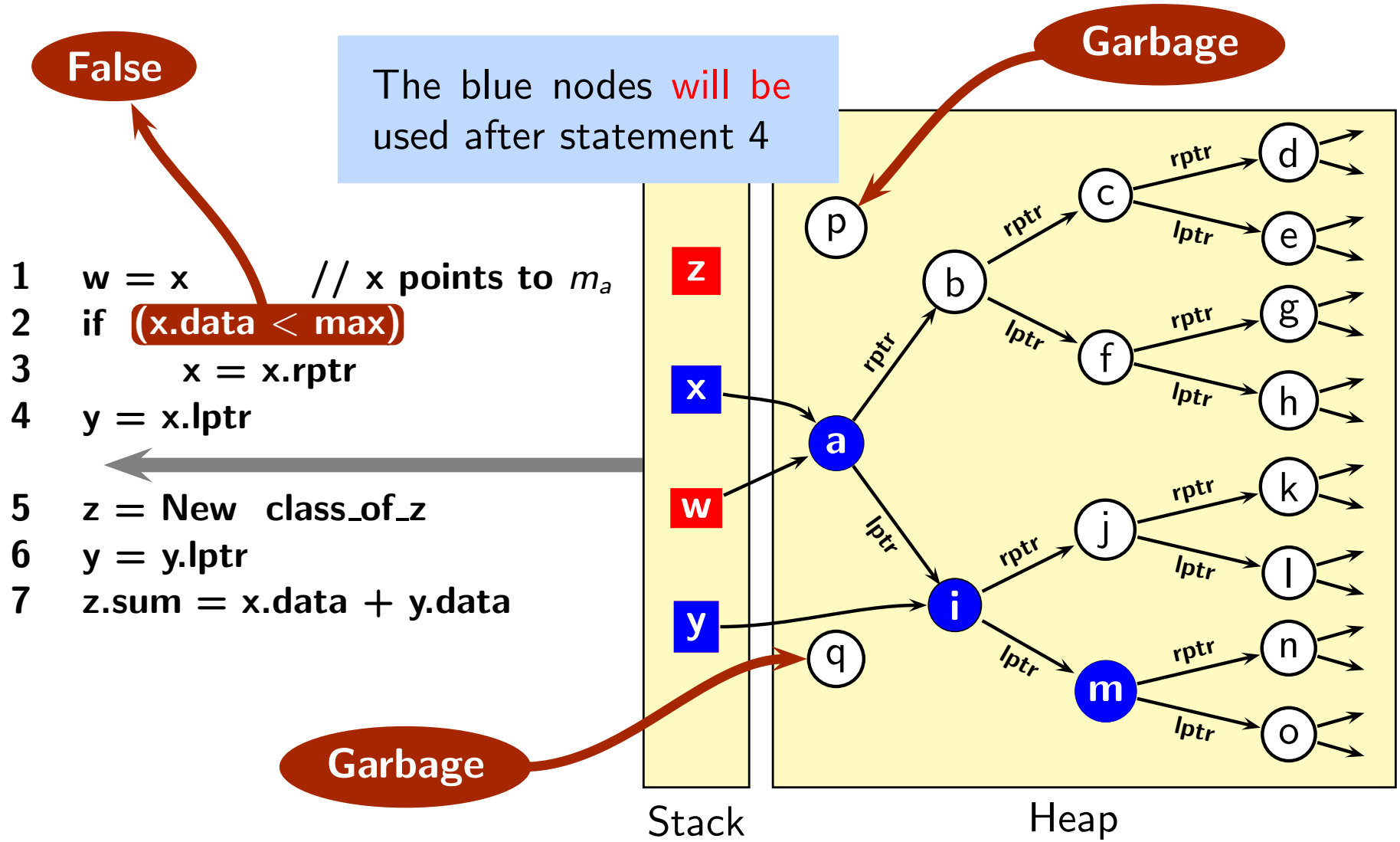
We use Java style statements for convenience

Read "**x.lptr**" as "**x→lptr**

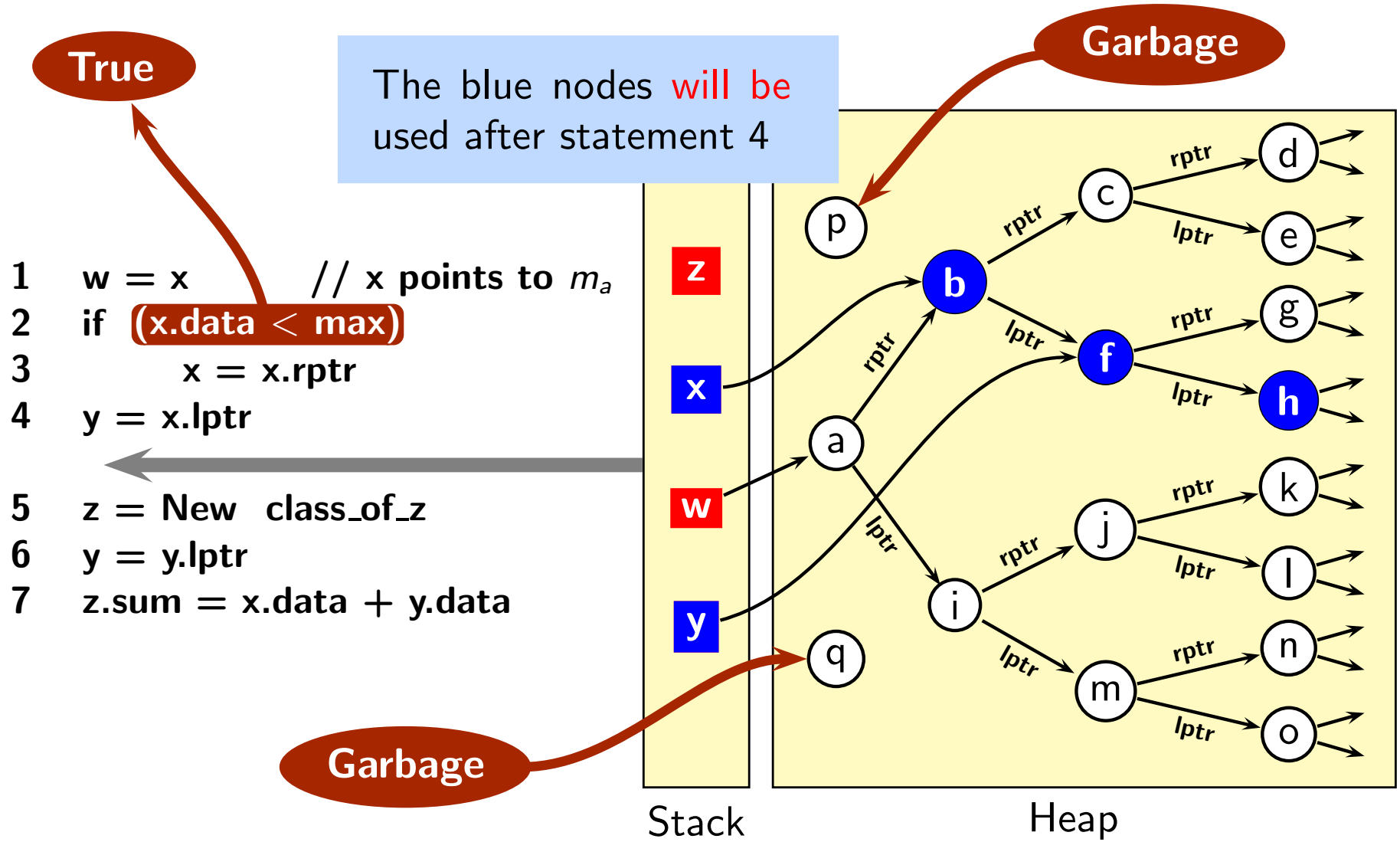
```
1    w = x          // x points to m_a
2    if  (x.data < max)
3          x = x.rptr
4    y = x.lptr

5    z = New  class_of_z
6    y = y.lptr
7    z.sum = x.data + y.data
```



What is Garbage?



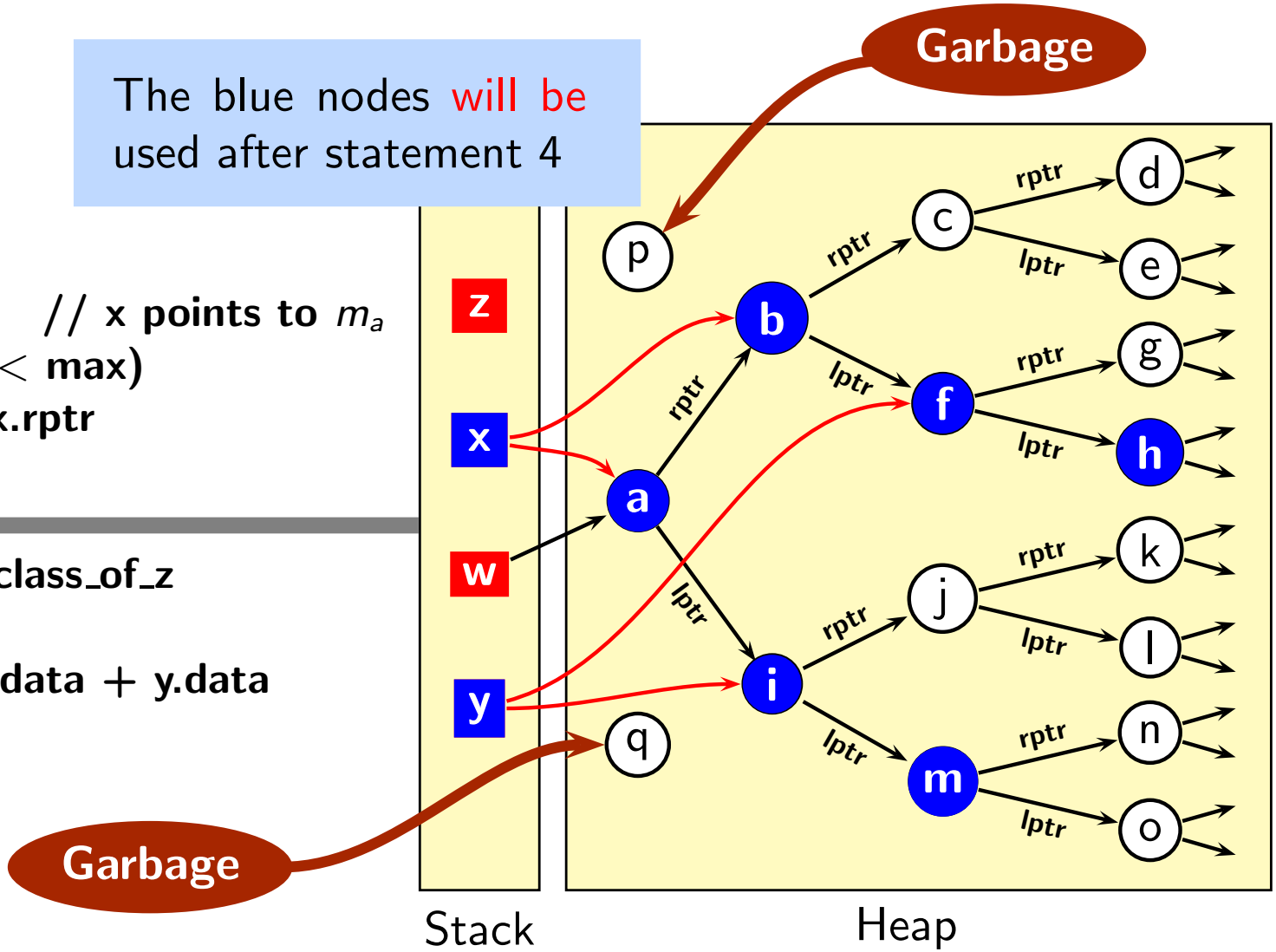
What is Garbage?



What is Garbage?

The blue nodes will be used after statement 4

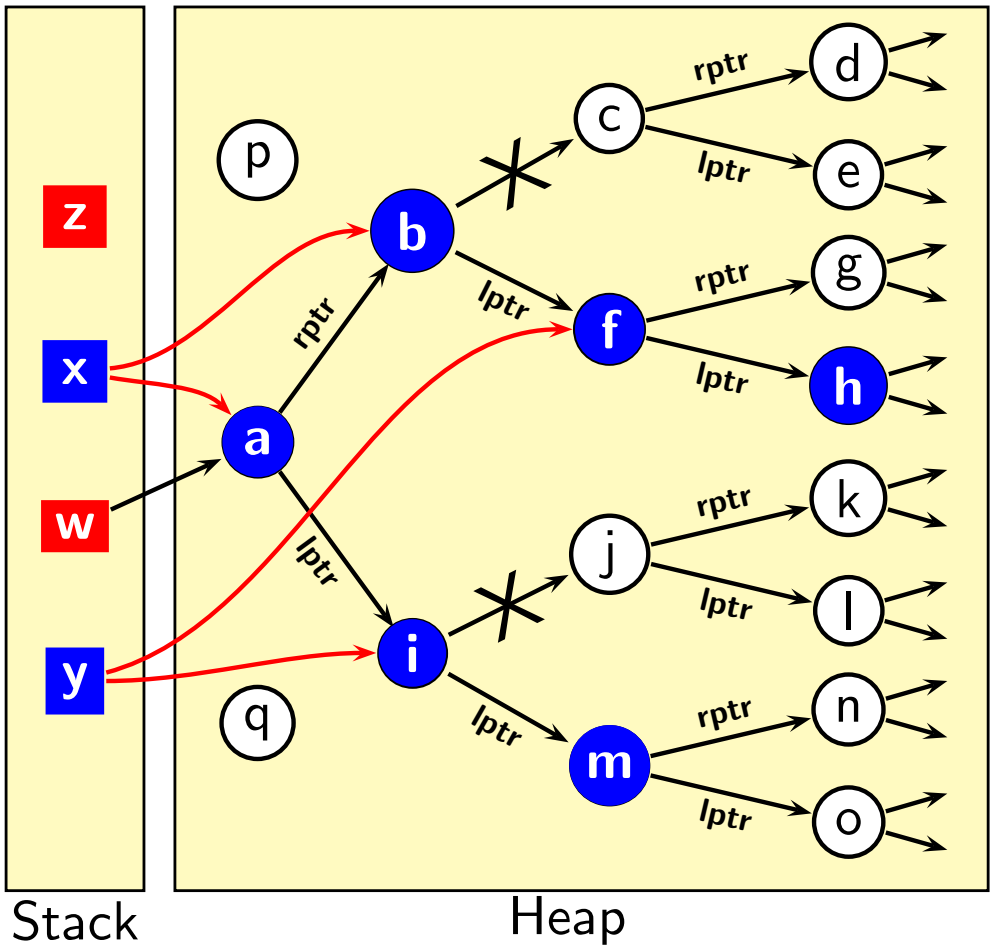
```
1    w = x          // x points to m_a
2    if  (x.data < max)
3          x = x.rptr
4    y = x.lptr

5    z = New  class_of_z
6    y = y.lptr
7    z.sum = x.data + y.data
```



All white nodes are unused and should be considered garbage

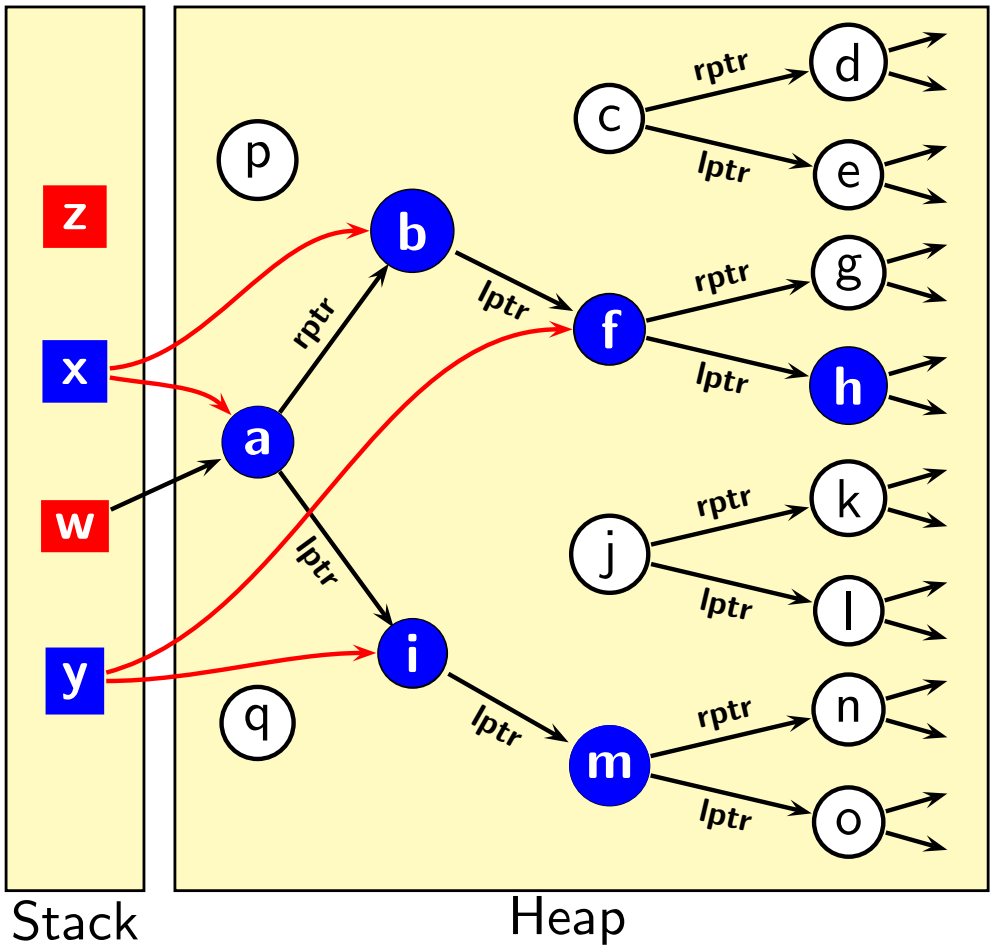
Cedar Mesa Folk Wisdom

Make the unused memory unreachable by setting references to NULL. (GC FAQ: `http://www.iecc.com/gclist/GC-harder.html`)



Cedar Mesa Folk Wisdom

Make the unused memory unreachable by setting references to NULL. (GC FAQ: `http://www.iecc.com/gclist/GC-harder.html`)

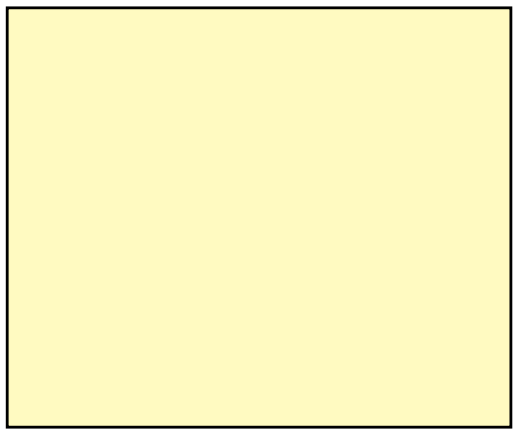


Liveness of Stack Data: An Informal Introduction

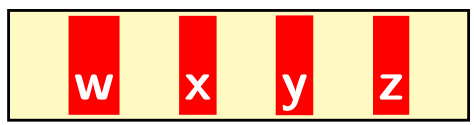
We use Java style statements for convenience

Read "**x.lptr**" as "**x→lptr**

```
1   w = x          // x points to m_a
2   while  (x.data < max)
3          x = x.rptr
4   y = x.lptr
5   z = New  class_of_z
6   y = y.lptr
7   z.sum = x.data + y.data
```



Heap

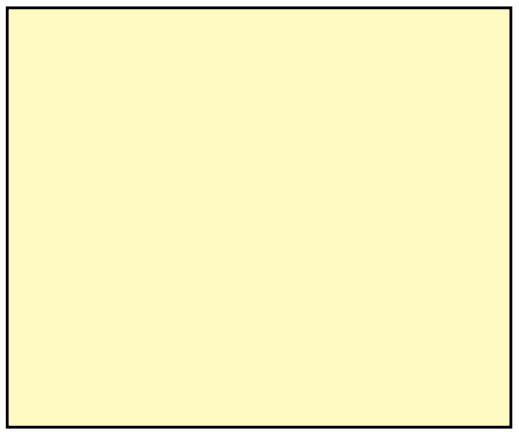


Stack

if changed to while

Liveness of Stack Data: An Informal Introduction

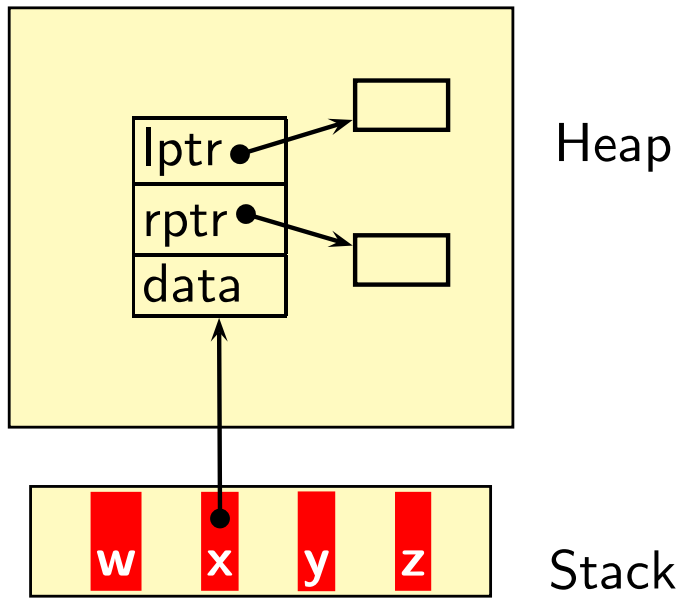
```
1   w = x          // x points to m_a
2   while  (x.data < max)
3          x = x.rptr
4   y = x.lptr
5   z = New  class_of_z
6   y = y.lptr
7   z.sum = x.data + y.data
```



What is the meaning of the use of data?

Liveness of Stack Data: An Informal Introduction

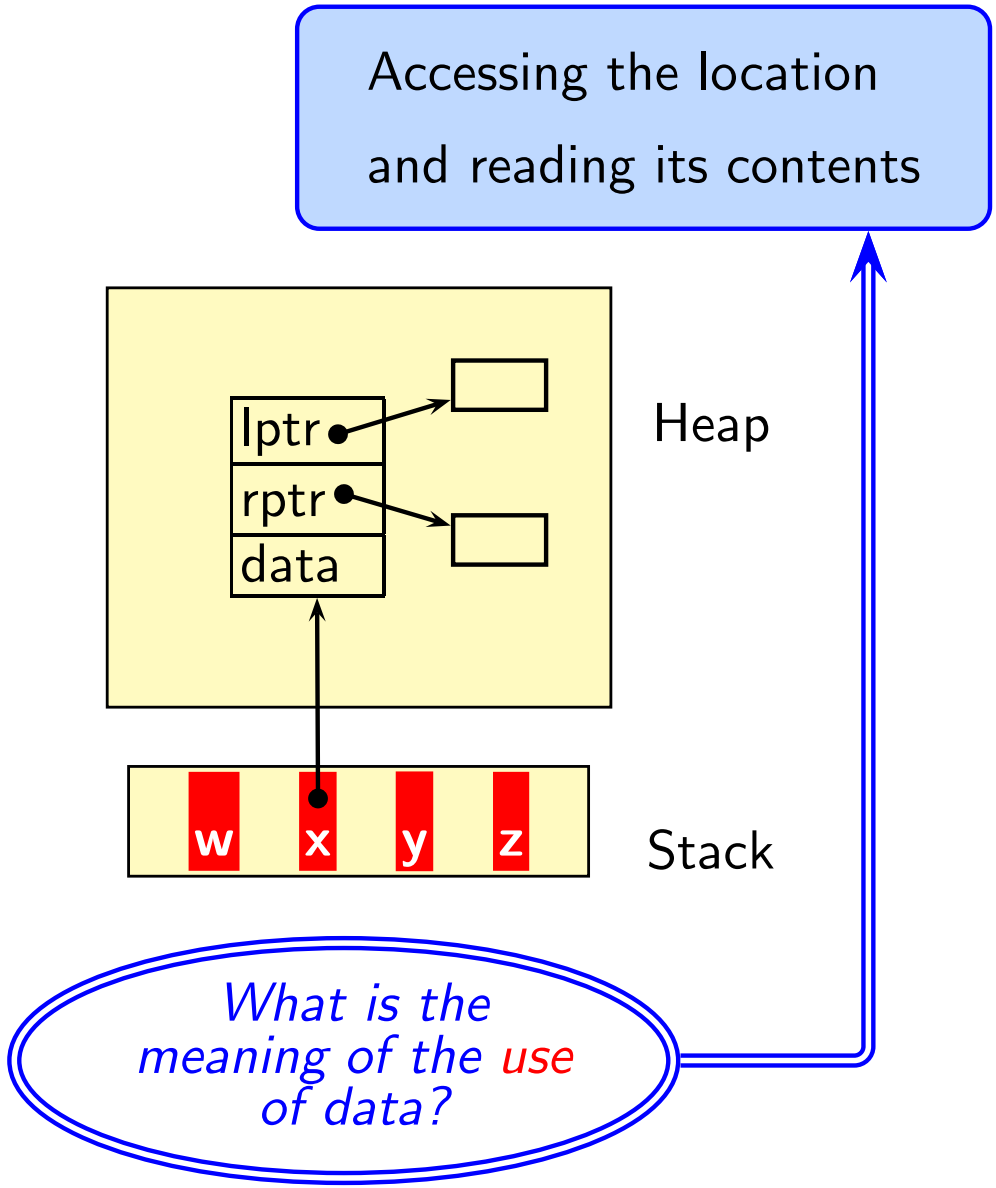
```
1   w = x          // x points to m_a
2   while  (x.data < max)
3          x = x.rptr
4   y = x.lptr
5   z = New  class_of_z
6   y = y.lptr
7   z.sum = x.data + y.data
```



What is the meaning of the use of data?

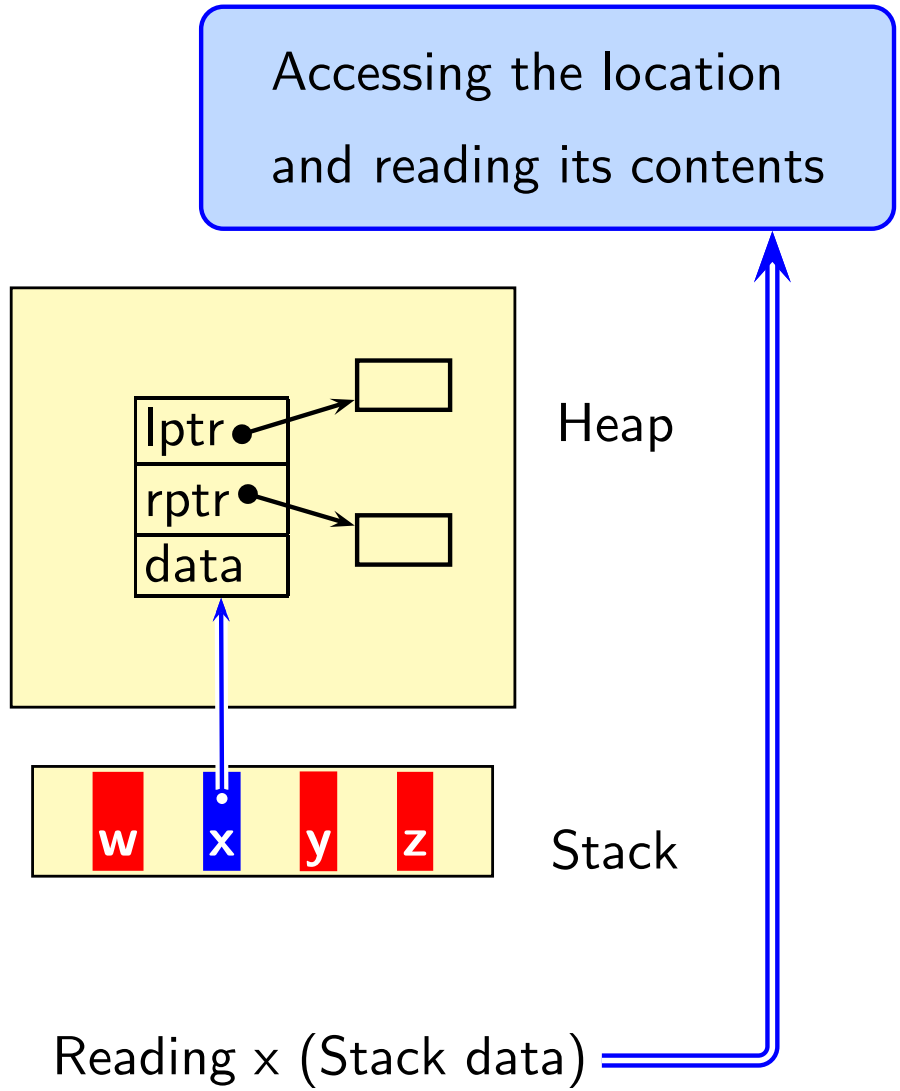
Liveness of Stack Data: An Informal Introduction

```
1   w = x          // x points to m_a
2   while (x.data < max)
3         x = x.rptr
4   y = x.lptr
5   z = New class_of_z
6   y = y.lptr
7   z.sum = x.data + y.data
```



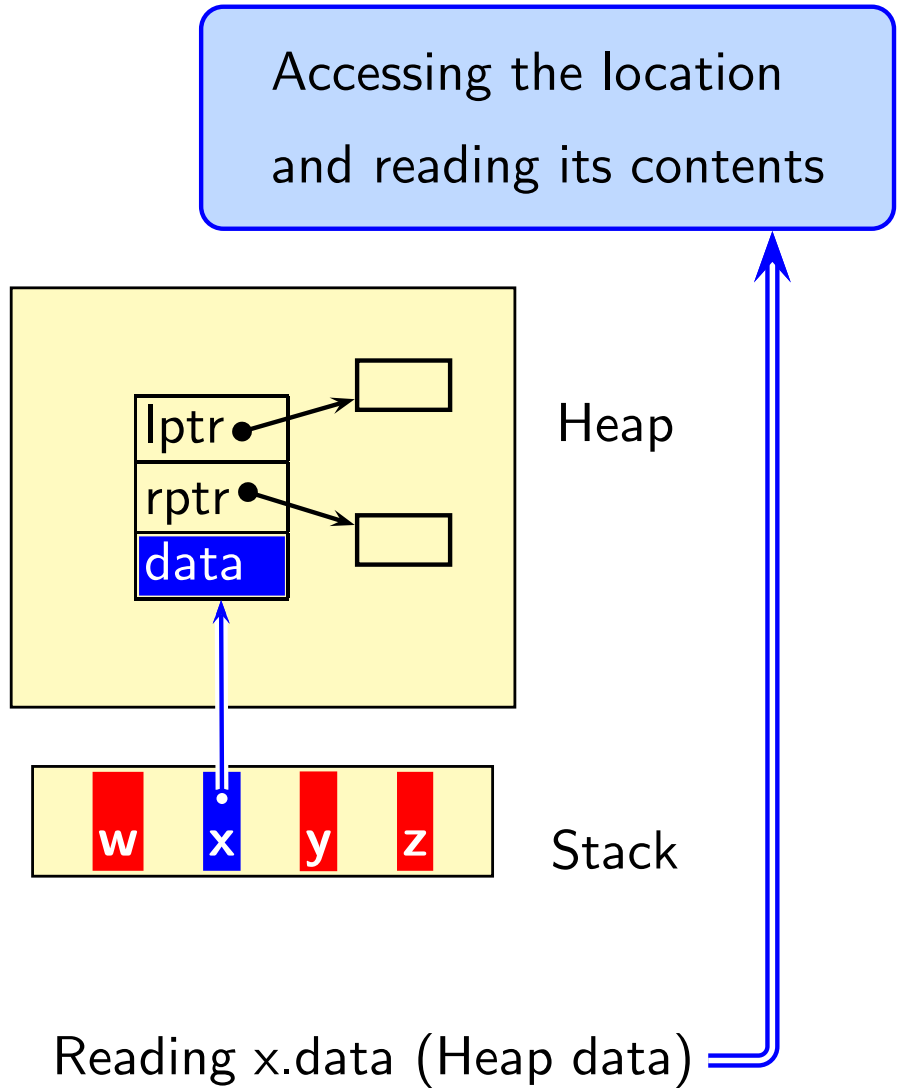
Liveness of Stack Data: An Informal Introduction

```
1   w = x          // x points to m_a
2   while  (x.data < max)
3          x = x.rptr
4   y = x.lptr
5   z = New  class_of_z
6   y = y.lptr
7   z.sum = x.data + y.data
```



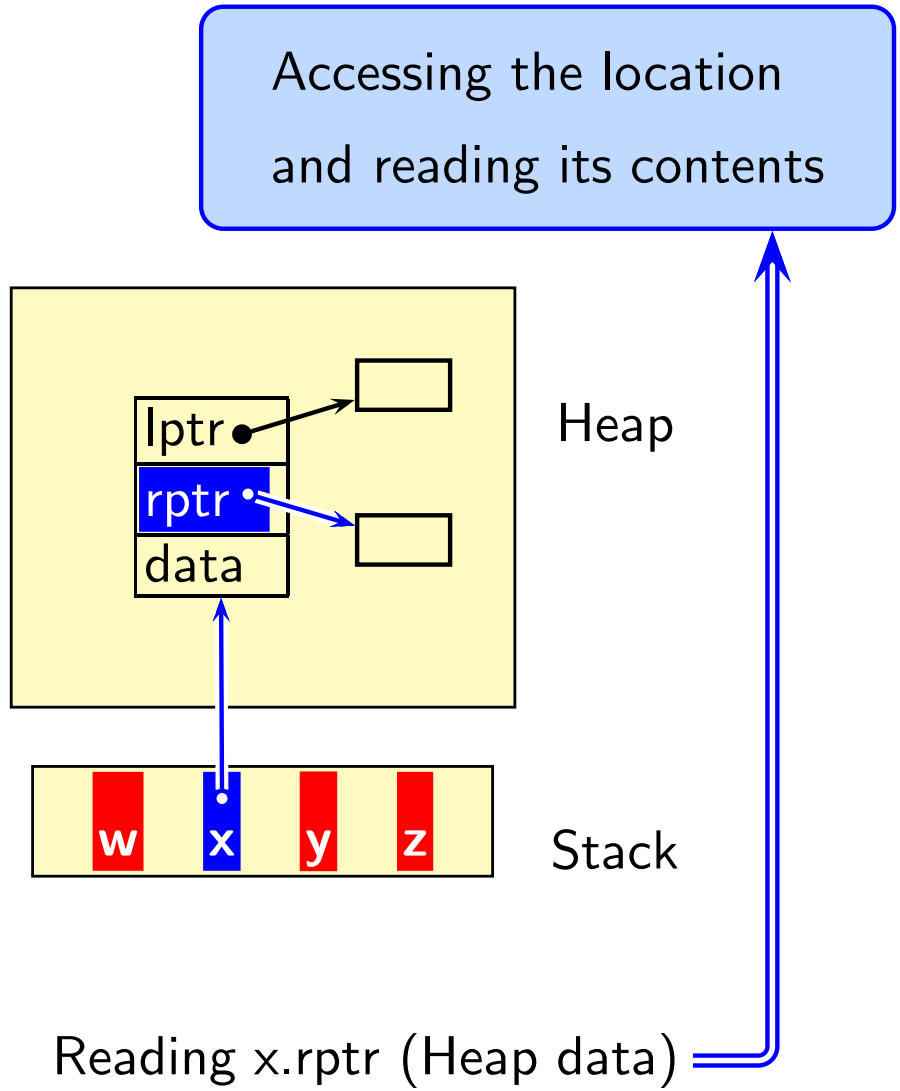
Liveness of Stack Data: An Informal Introduction

```
1   w = x          // x points to m_a
2   while  (x.data < max)
3          x = x.rptr
4   y = x.lptr
5   z = New  class_of_z
6   y = y.lptr
7   z.sum = x.data + y.data
```

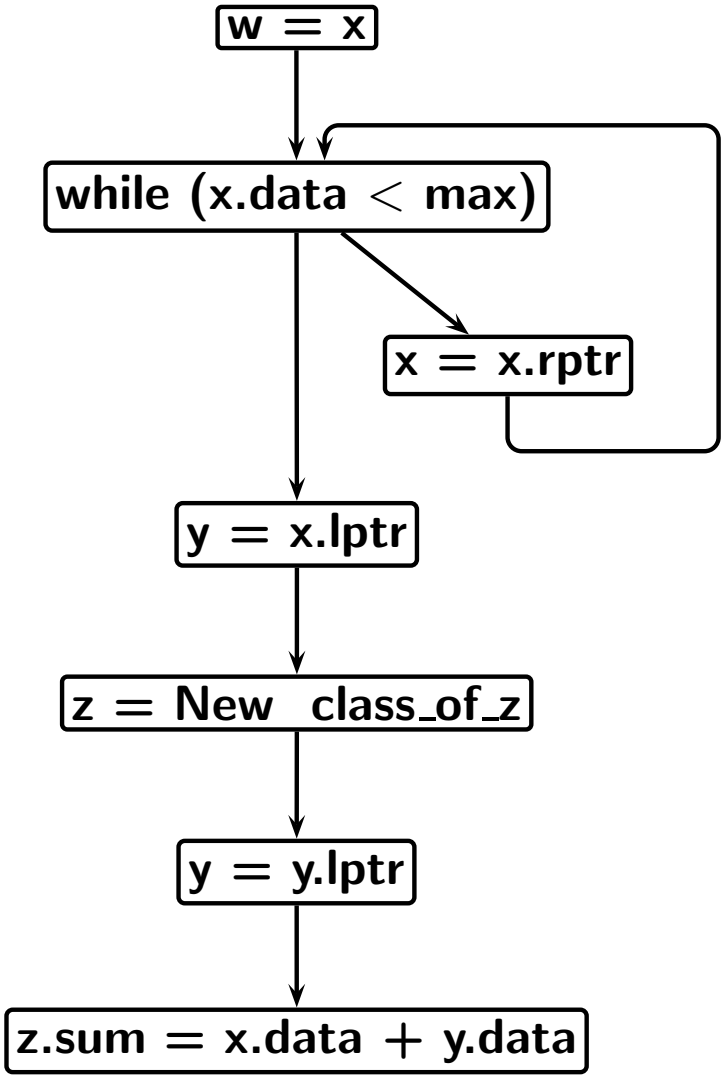


Liveness of Stack Data: An Informal Introduction

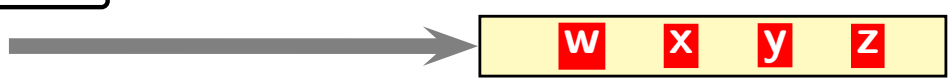
```
1   w = x          // x points to m_a
2   while  (x.data < max)
3          x = x.rptr
4   y = x.lptr
5   z = New  class_of_z
6   y = y.lptr
7   z.sum = x.data + y.data
```



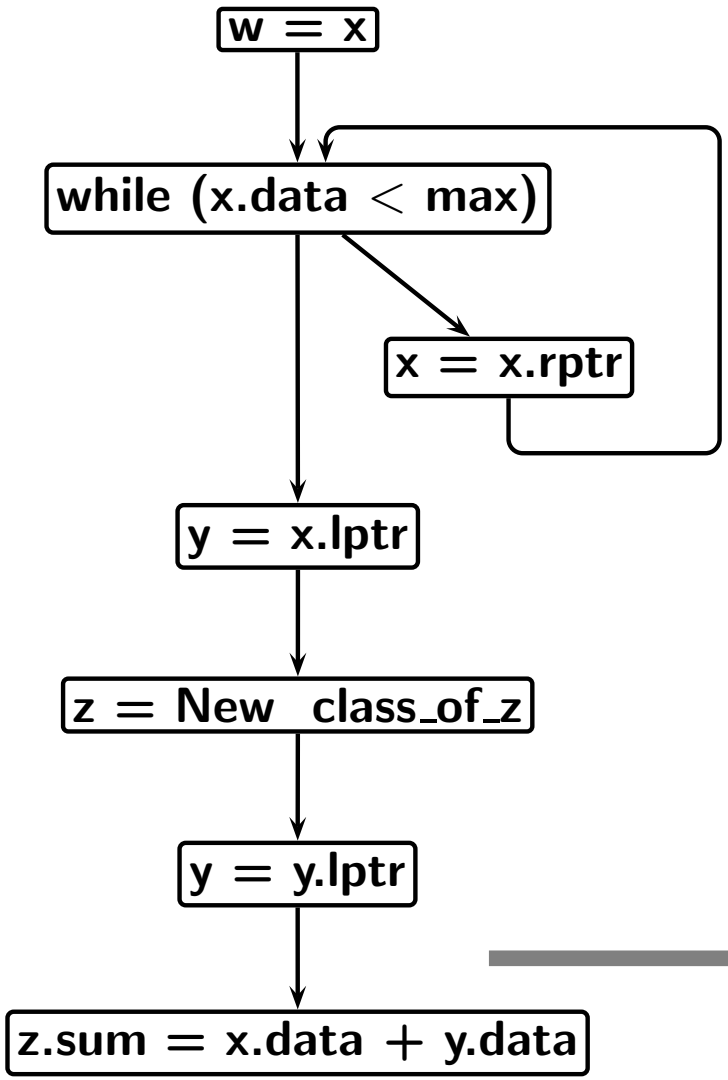
Liveness of Stack Data: An Informal Introduction



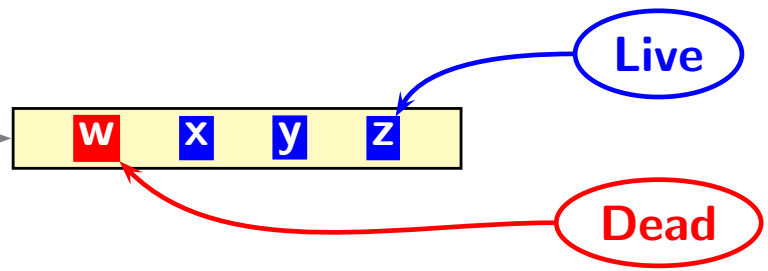
No variable is used beyond this program point



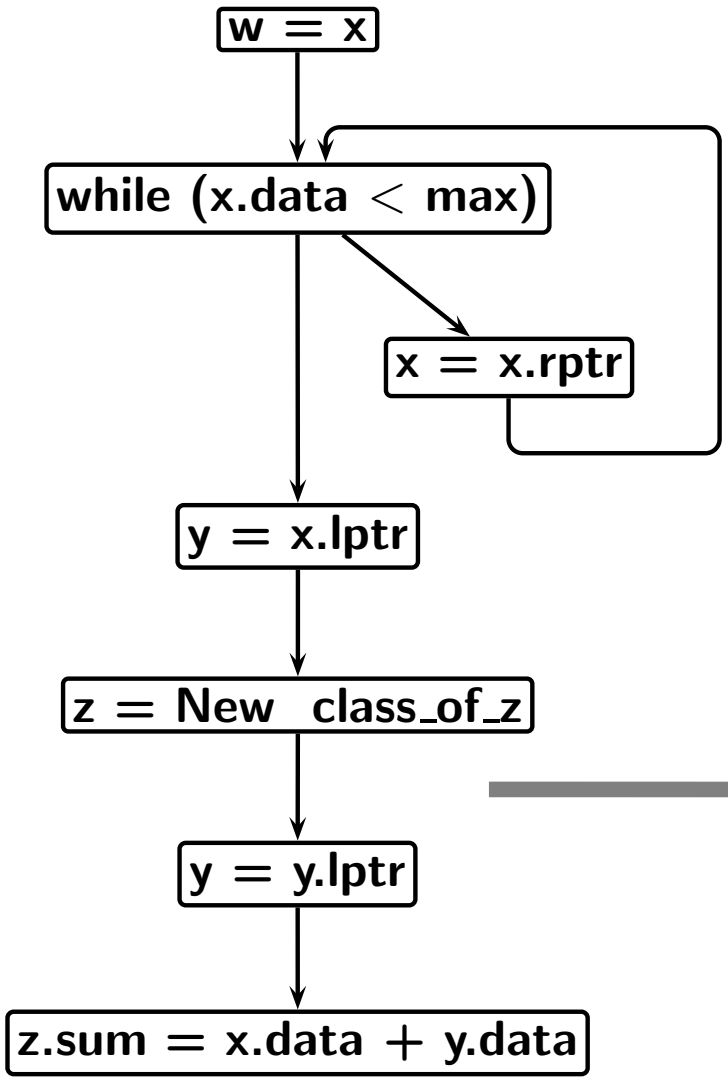
Liveness of Stack Data: An Informal Introduction

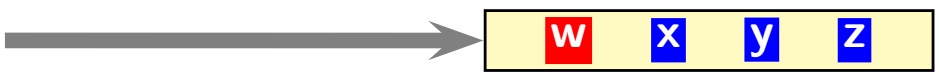


Current values of x, y, and z are used beyond this program point



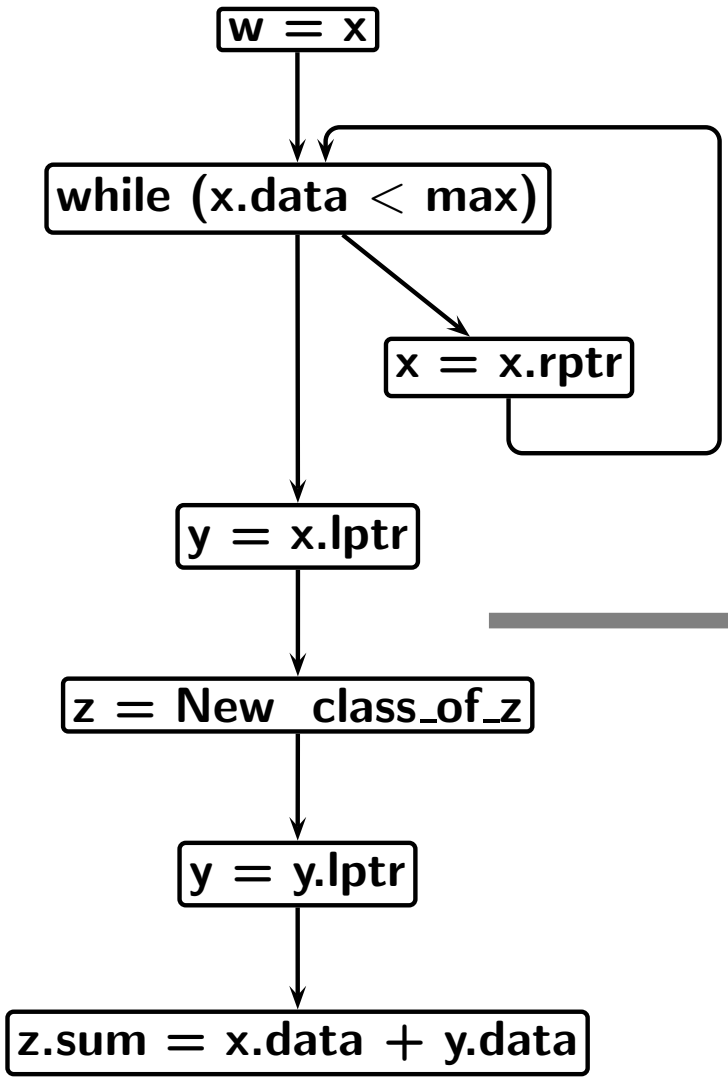
Liveness of Stack Data: An Informal Introduction

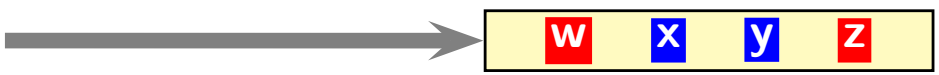




- Current values of x, y, and z are used beyond this program point
- The value of y is different before and after the assignment to y

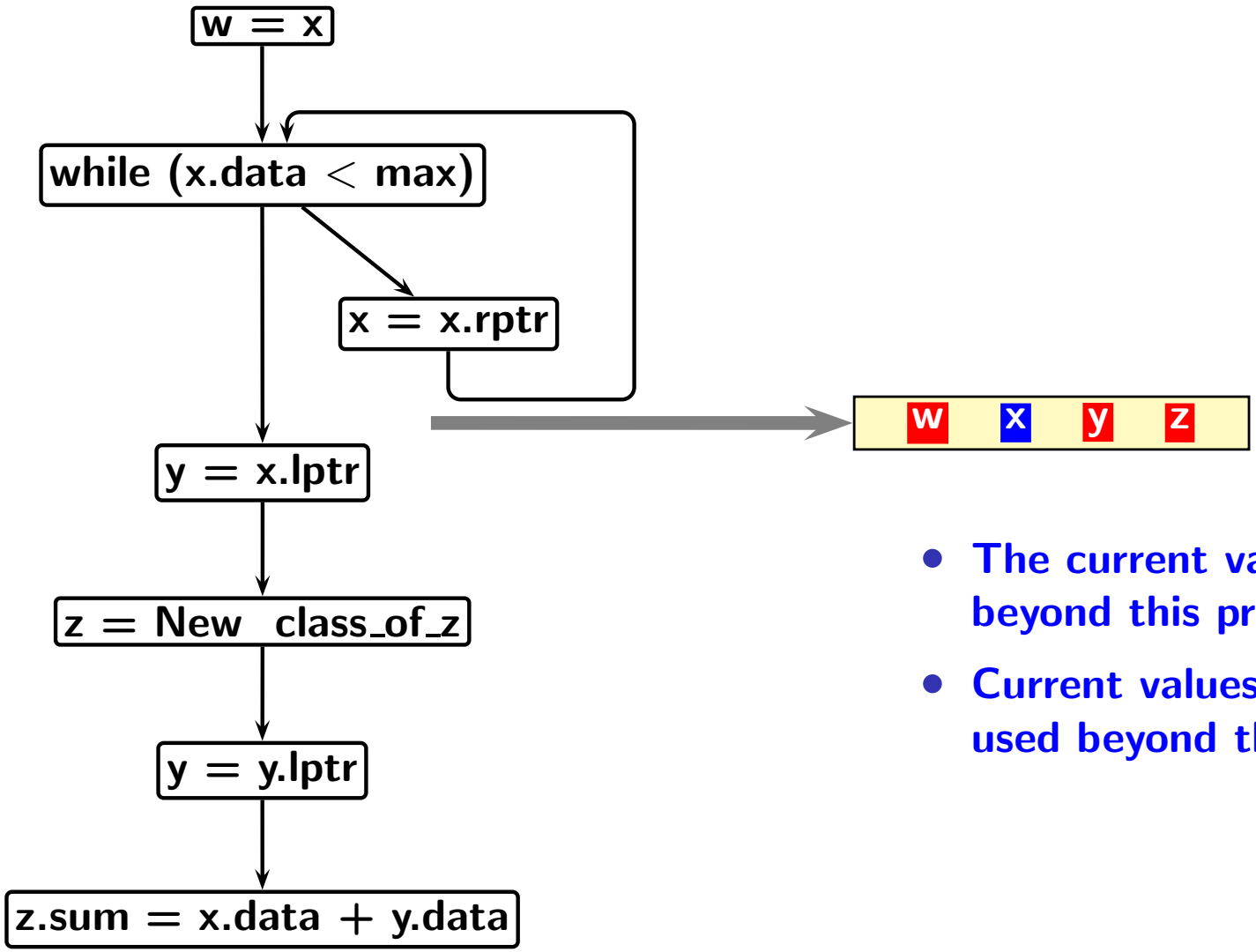
Liveness of Stack Data: An Informal Introduction





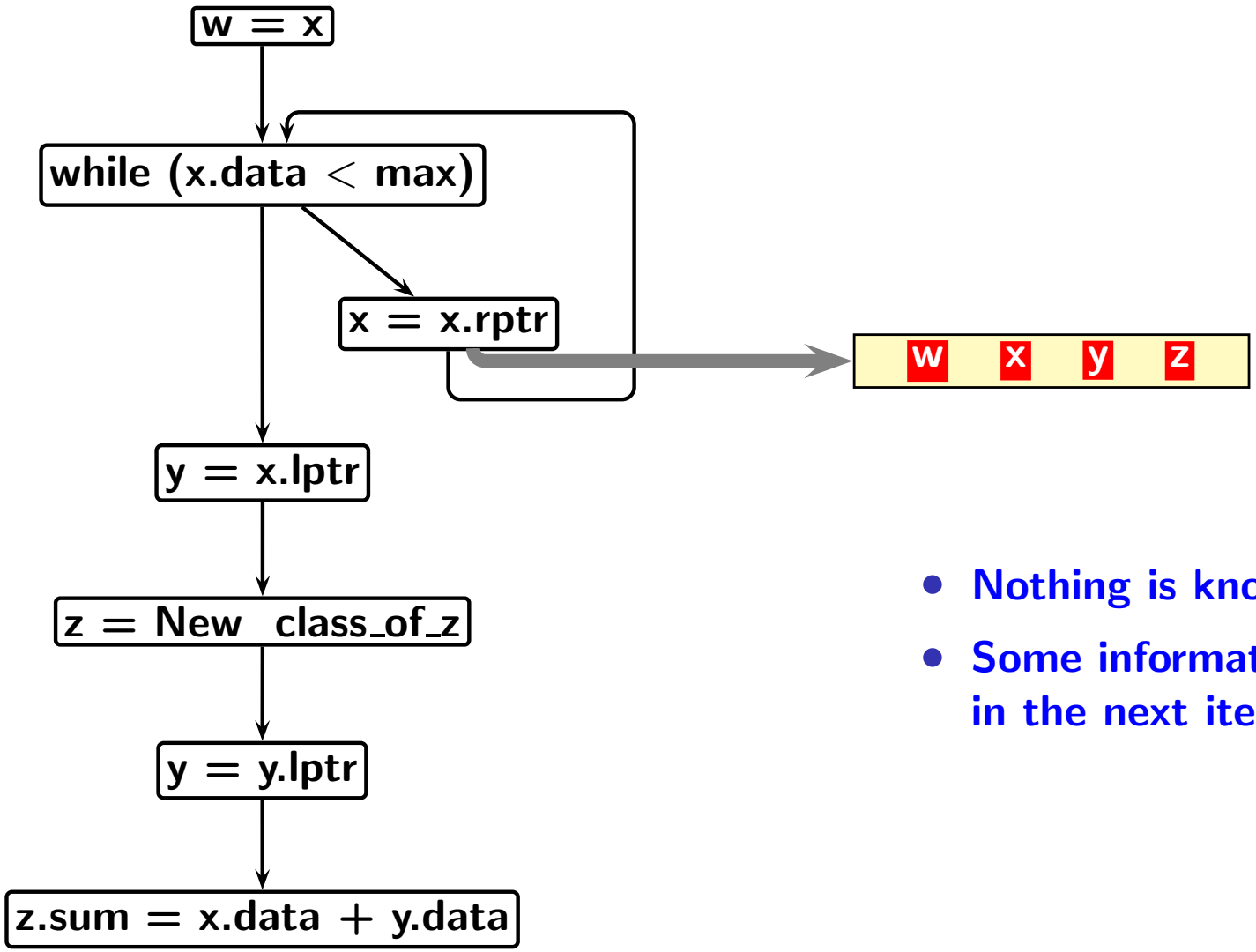
- The current values of x and y are used beyond this program point
- The current value of z is not used beyond this program point

Liveness of Stack Data: An Informal Introduction



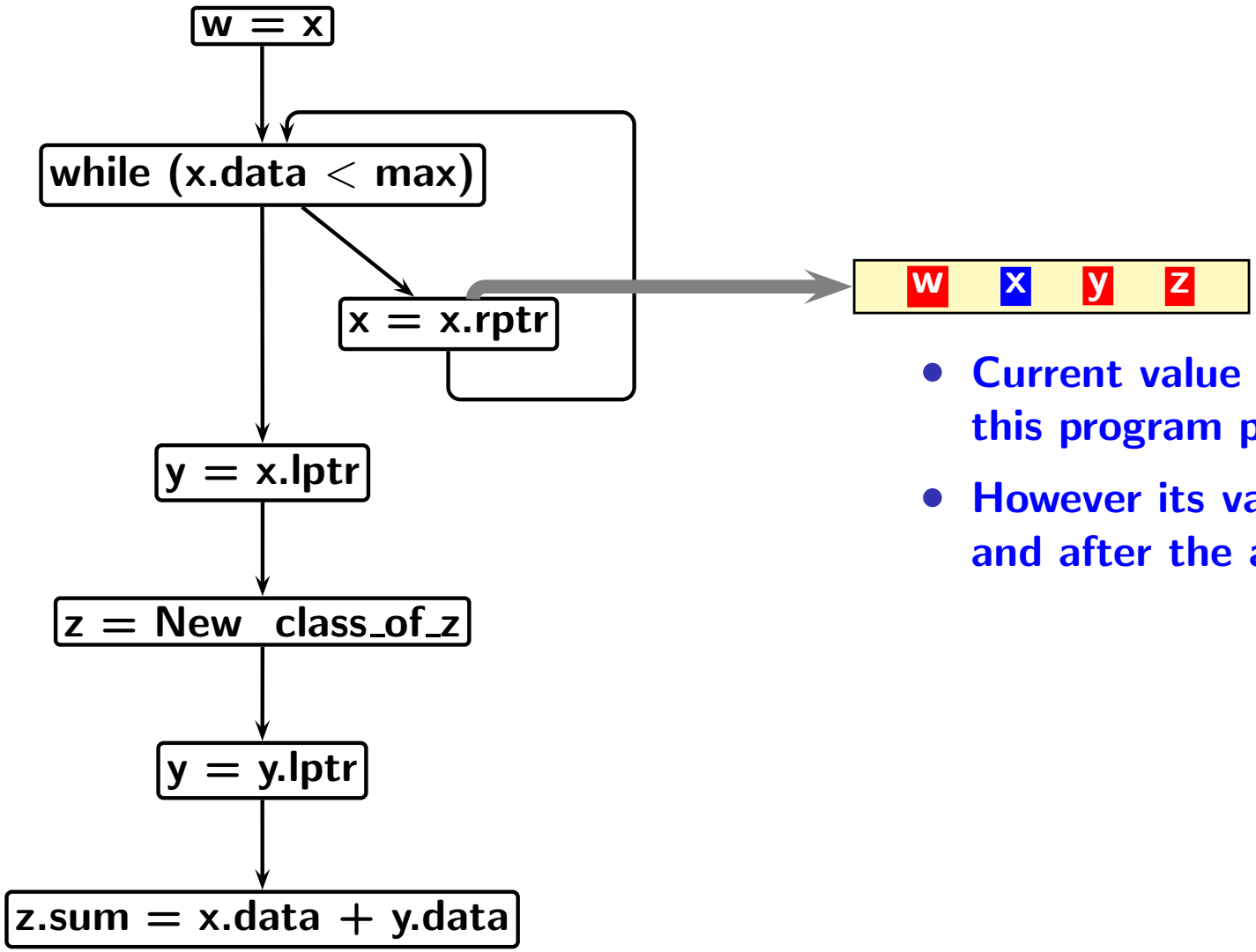
- The current values of x is used beyond this program point
- Current values of y and z are not used beyond this program point

Liveness of Stack Data: An Informal Introduction



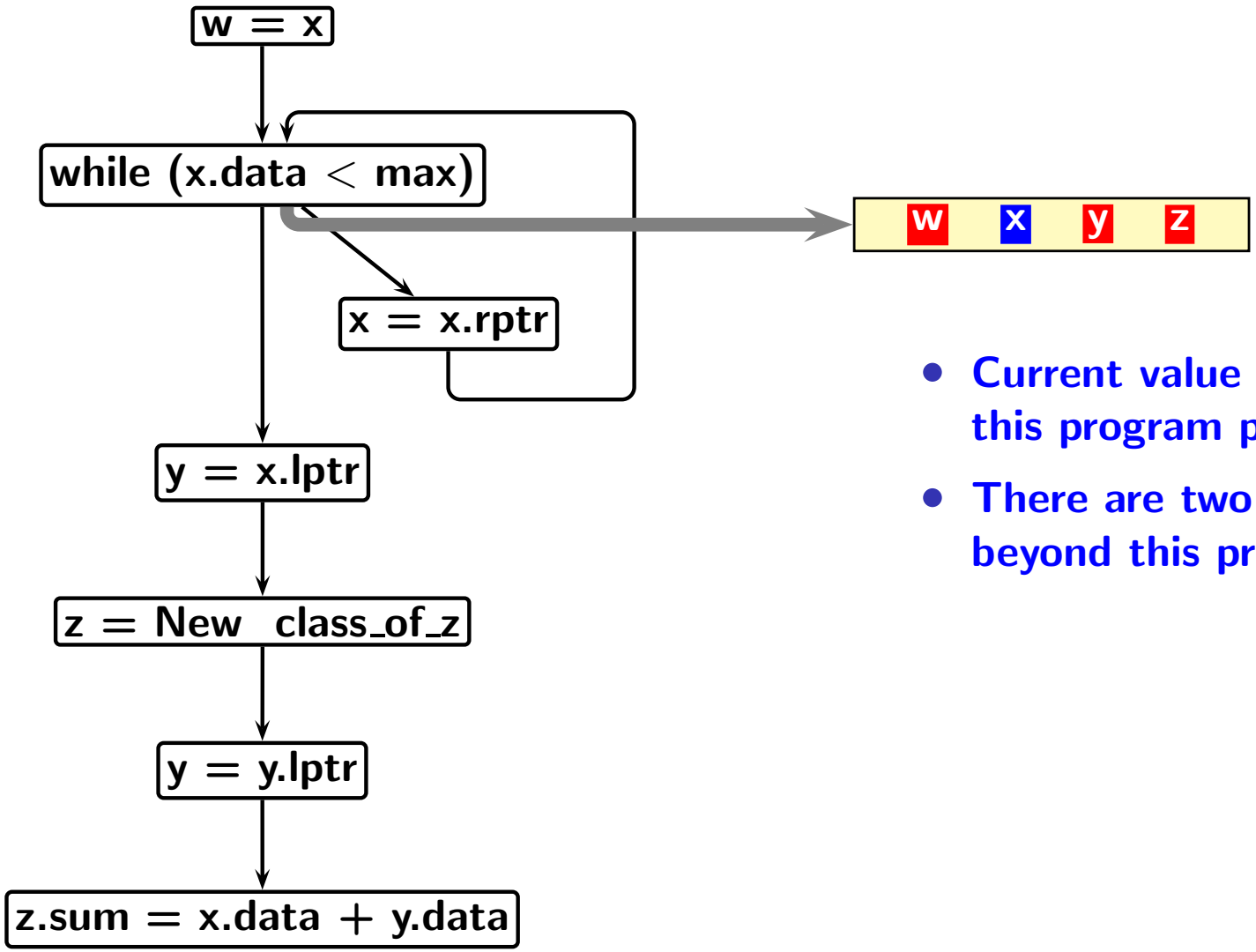
- Nothing is known as of now
- Some information will be available in the next iteration point

Liveness of Stack Data: An Informal Introduction



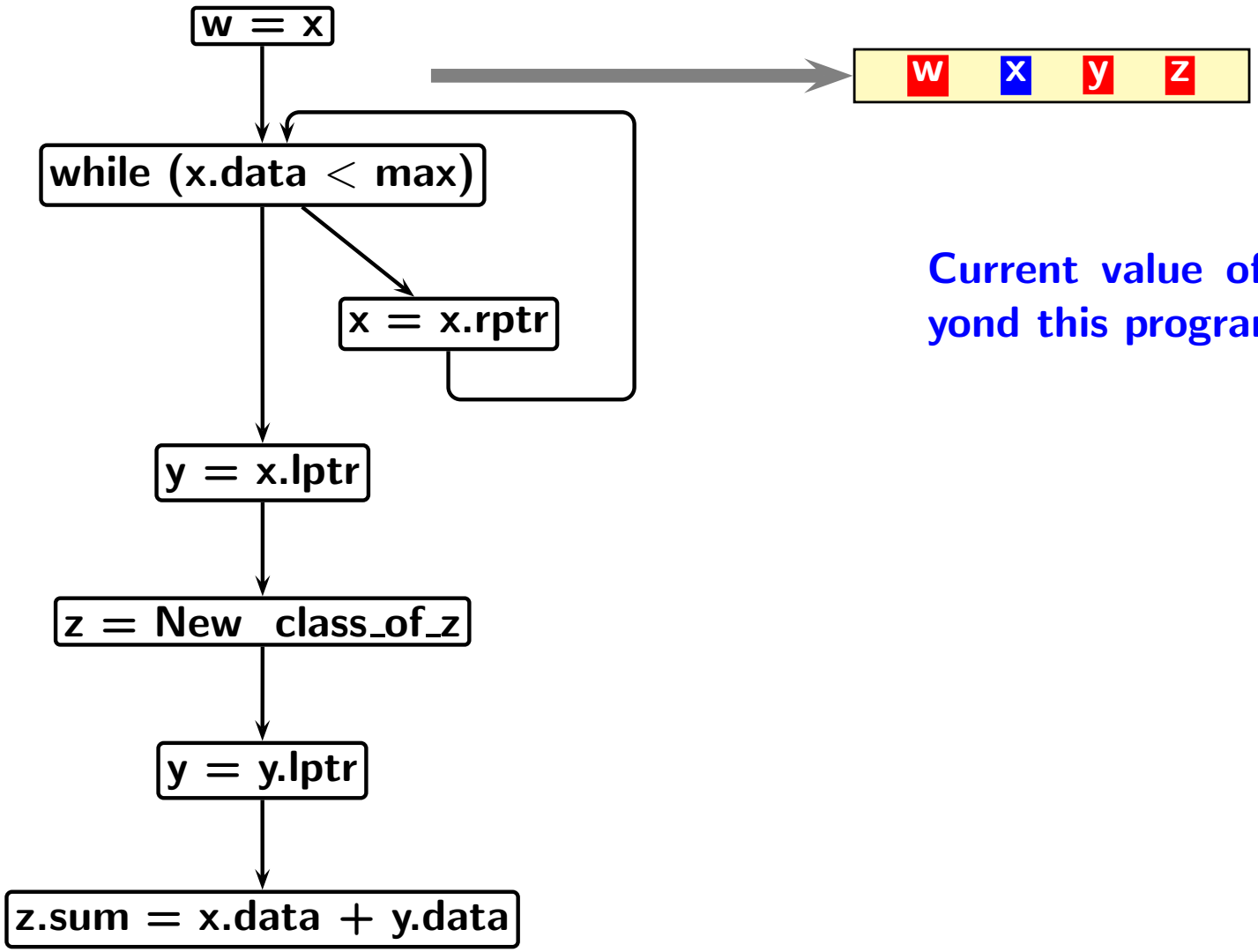
- Current value of x is used beyond this program point
- However its value is different before and after the assignment

Liveness of Stack Data: An Informal Introduction



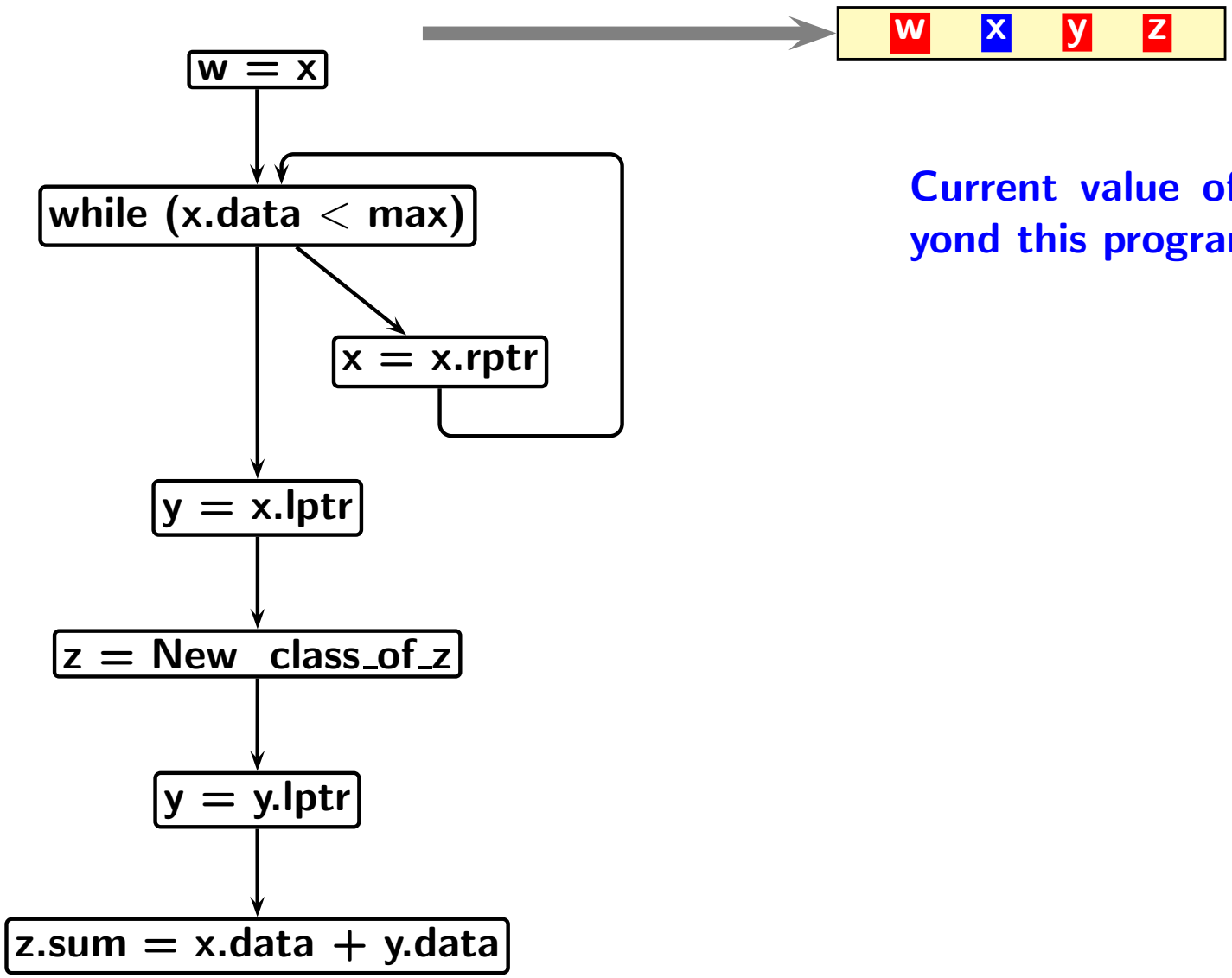
- Current value of x is used beyond this program point
- There are two control flow paths beyond this program point

Liveness of Stack Data: An Informal Introduction



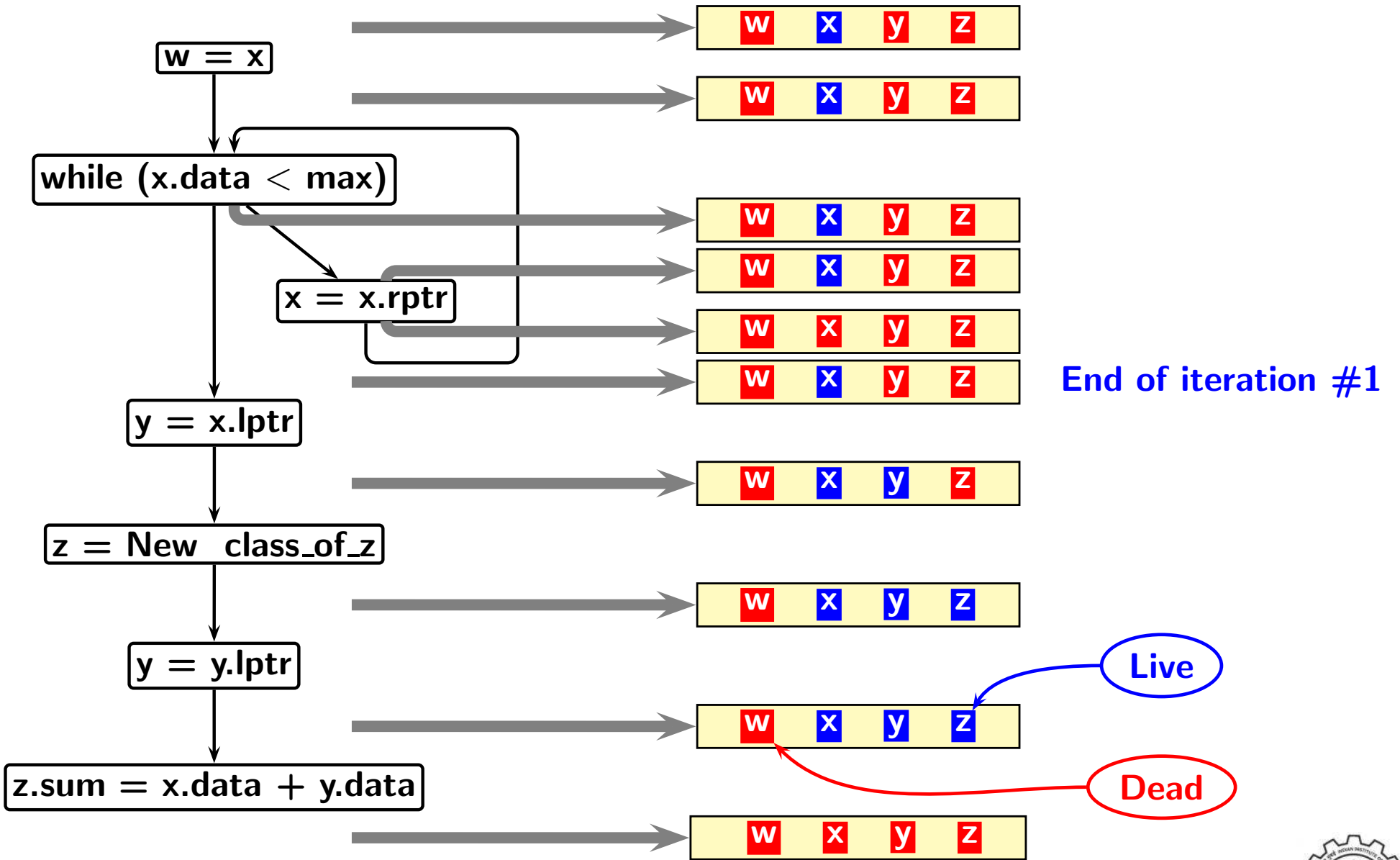
Current value of x is used beyond this program point

Liveness of Stack Data: An Informal Introduction

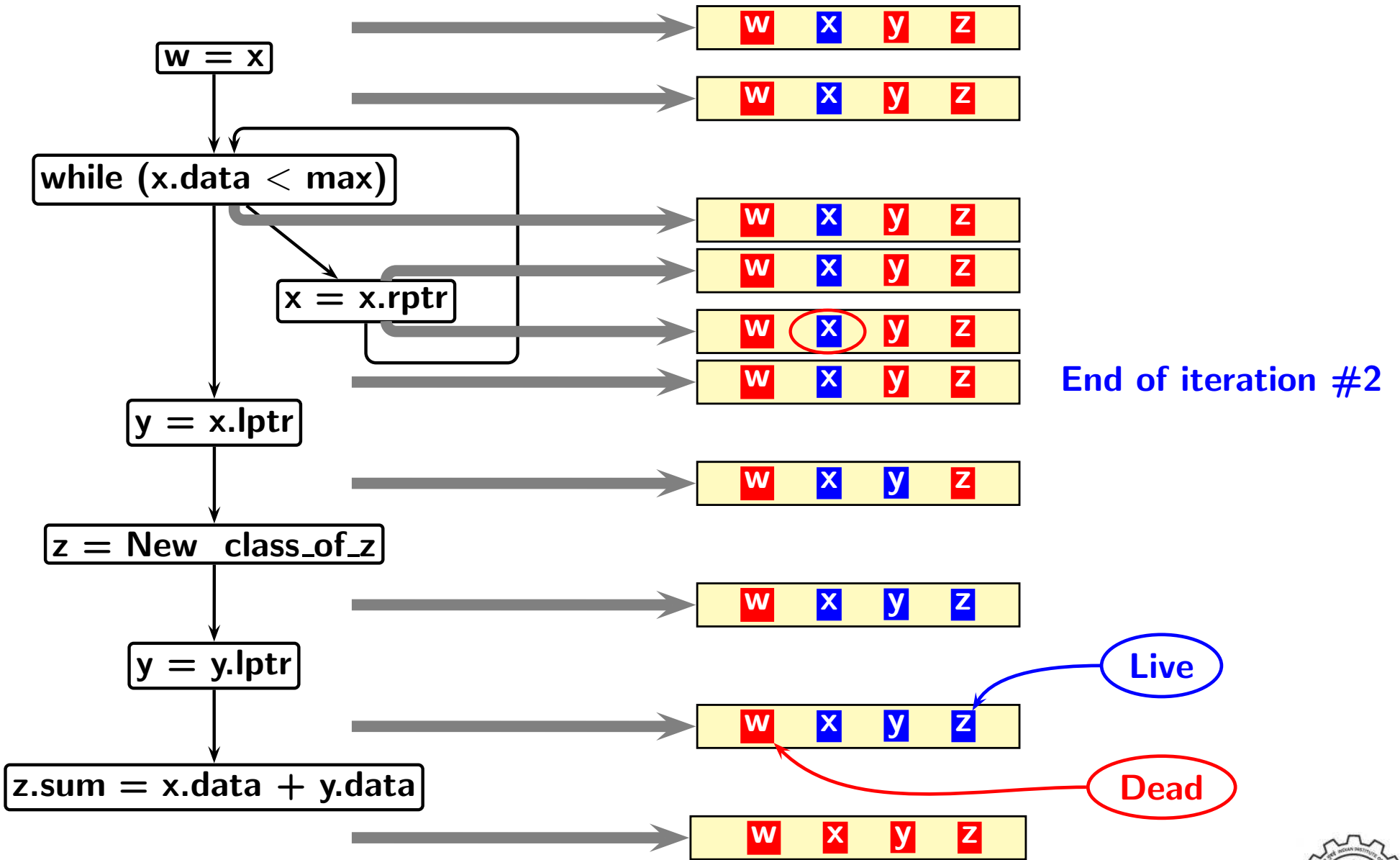


Current value of x is used beyond this program point

Liveness of Stack Data: An Informal Introduction



Liveness of Stack Data: An Informal Introduction

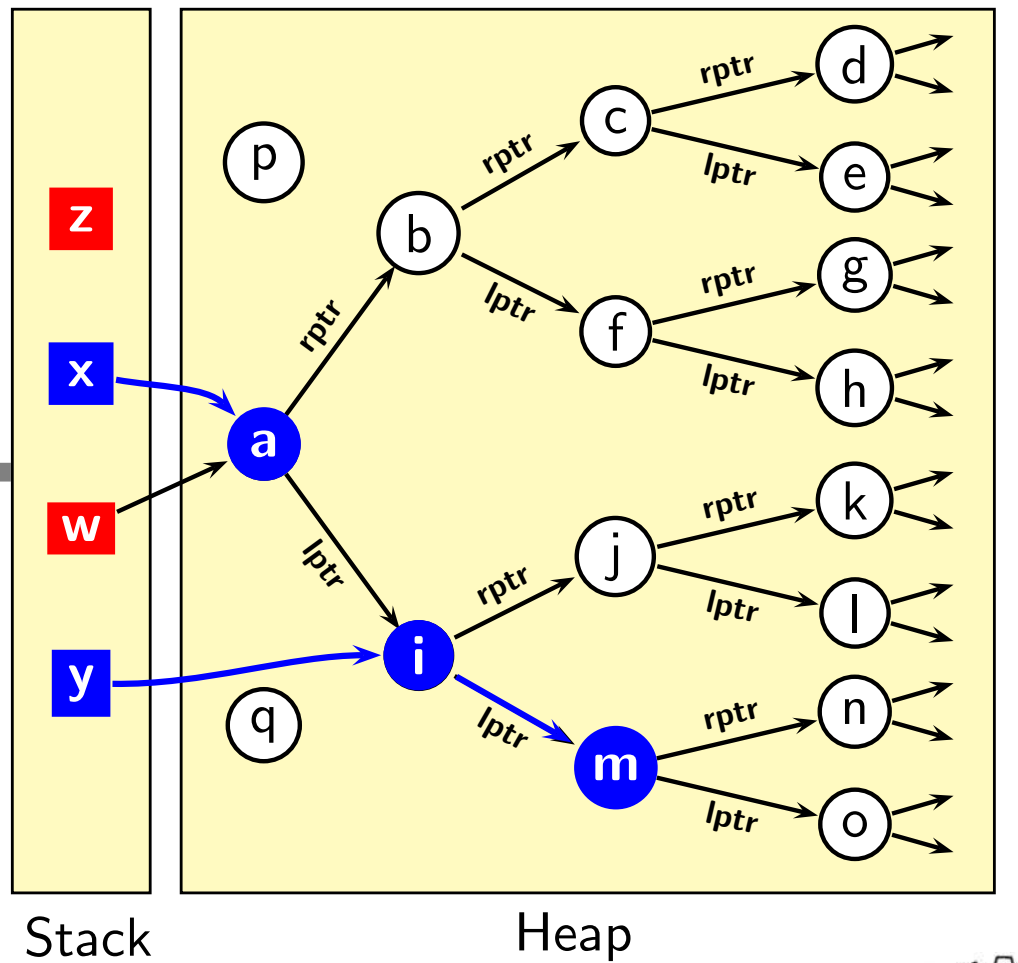


Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the while loop is not executed even once.

```
1    w = x          // x points to m_a
2    while  (x.data < max)
3           x = x.rptr
4    y = x.lptr

5    z = New  class_of_z
6    y = y.lptr
7    z.sum = x.data + y.data
```

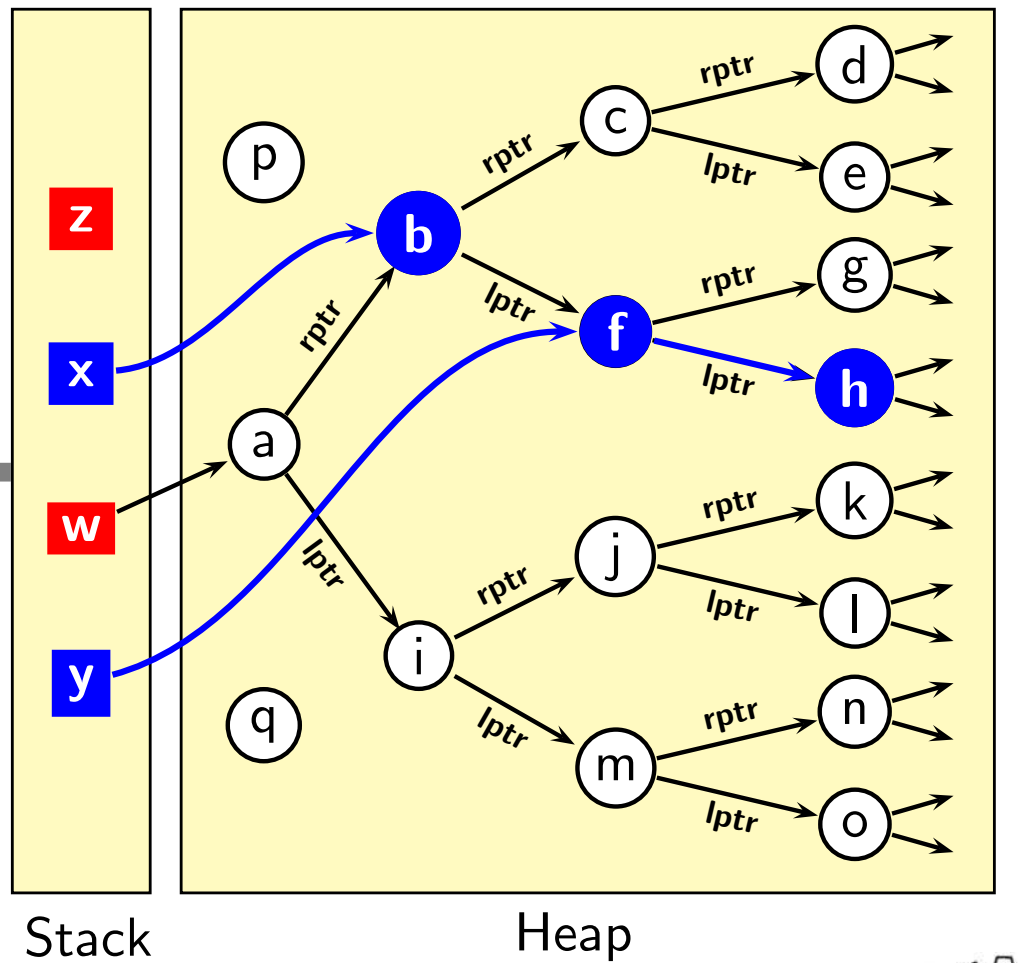


Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the while loop is executed once.

```
1    w = x          // x points to m_a
2    while  (x.data < max)
3           x = x.rptr
4    y = x.lptr
5    z = New  class_of_z
6    y = y.lptr
7    z.sum = x.data + y.data
```

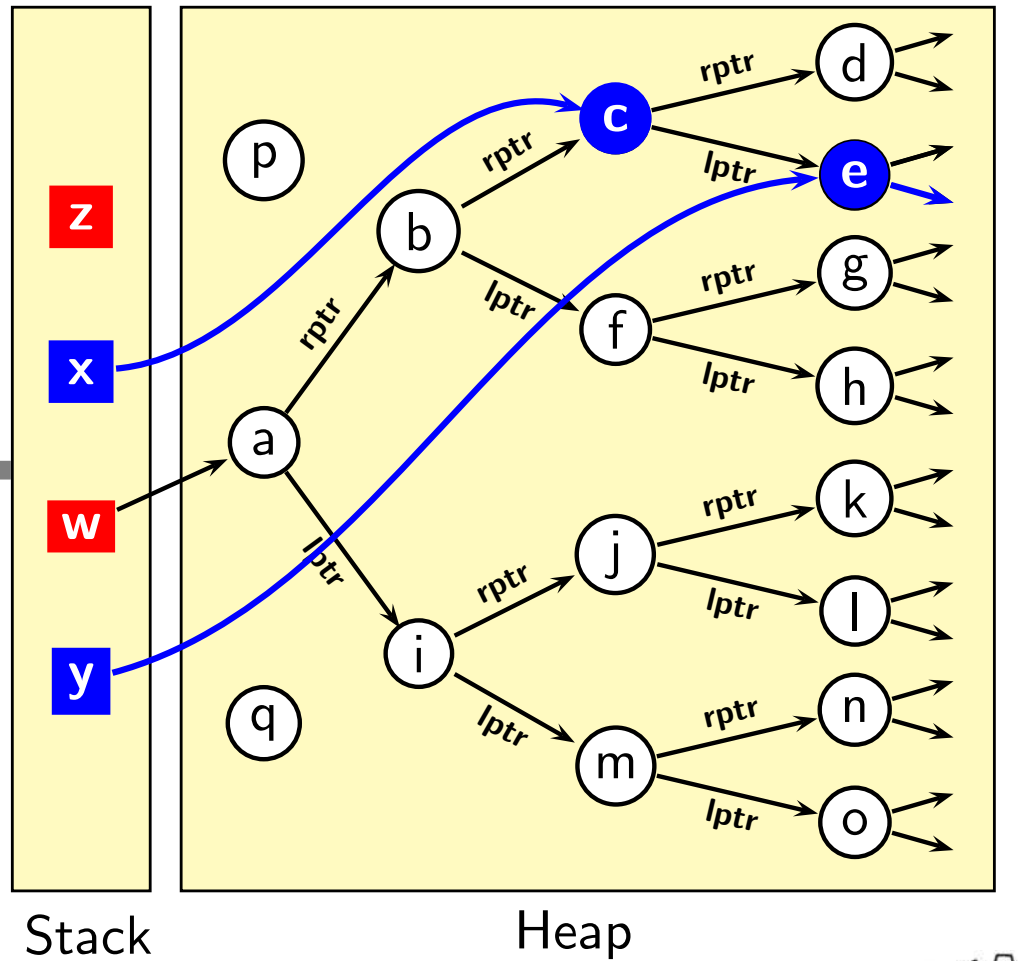


Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the while loop is executed twice.

```
1    w = x          // x points to m_a
2    while  (x.data < max)
3           x = x.rptr
4    y = x.lptr

5    z = New  class_of_z
6    y = y.lptr
7    z.sum = x.data + y.data
```



The Moral of the Story

- Mappings between access expressions and l-values keep changing
- This is a *rule* for heap data

  For stack and static data, it is an *exception*!
- Static analysis of programs has made significant progress for stack and static data.

  What about heap data?
  - Given two access expressions at a program point, do they have the same l-value?
  - Given the same access expression at two program points, does it have the same l-value?

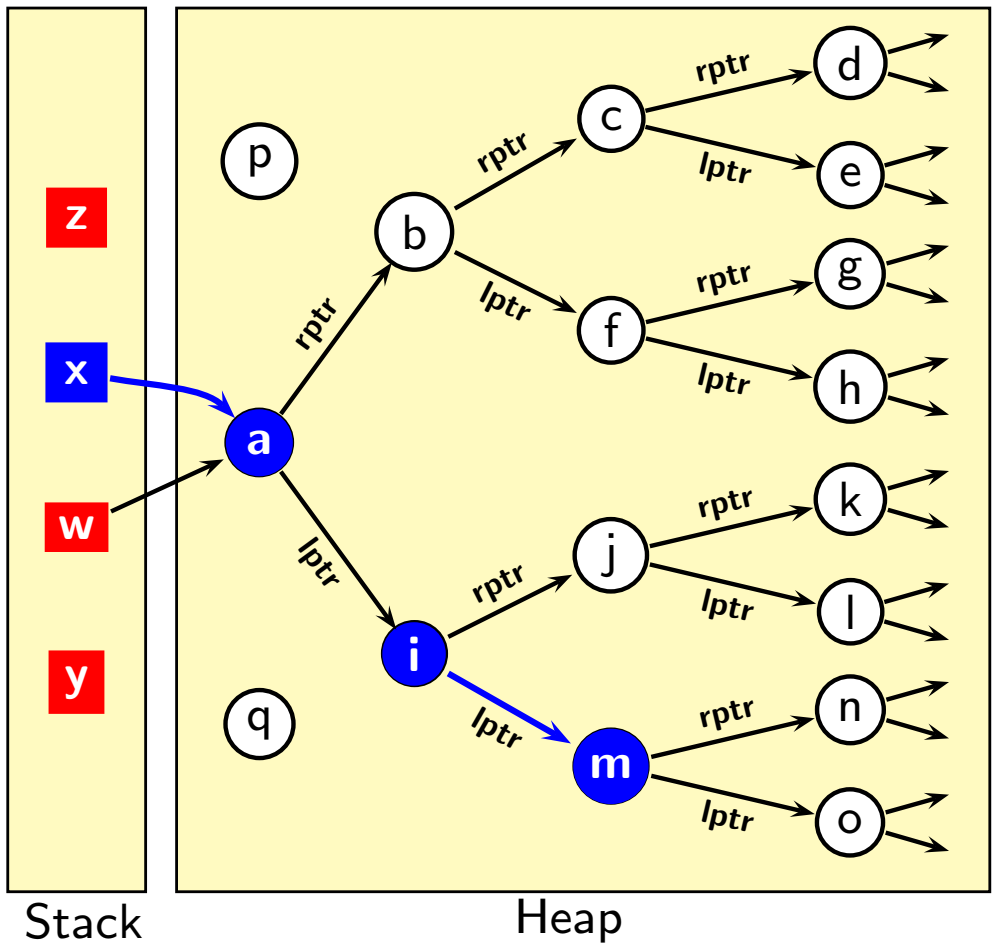
Our Solution

		y = z = null
1	w = x	
		w = null
2	while (x.data < max)	
	{	x.lptr = null
3	x = x.rptr }	
		x.rptr = x.lptr.rptr = null
		x.lptr.lptr.lptr = null
		x.lptr.lptr.rptr = null
4	y = x.lptr	
		x.lptr = y.rptr = null
		y.lptr.lptr = y.lptr.rptr = null
5	z = *New class_of_z*	
		z.lptr = z.rptr = null
6	y = y.lptr	
		y.lptr = y.rptr = null
7	z.sum = x.data + y.data	
		x = y = z = null

Our Solution

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

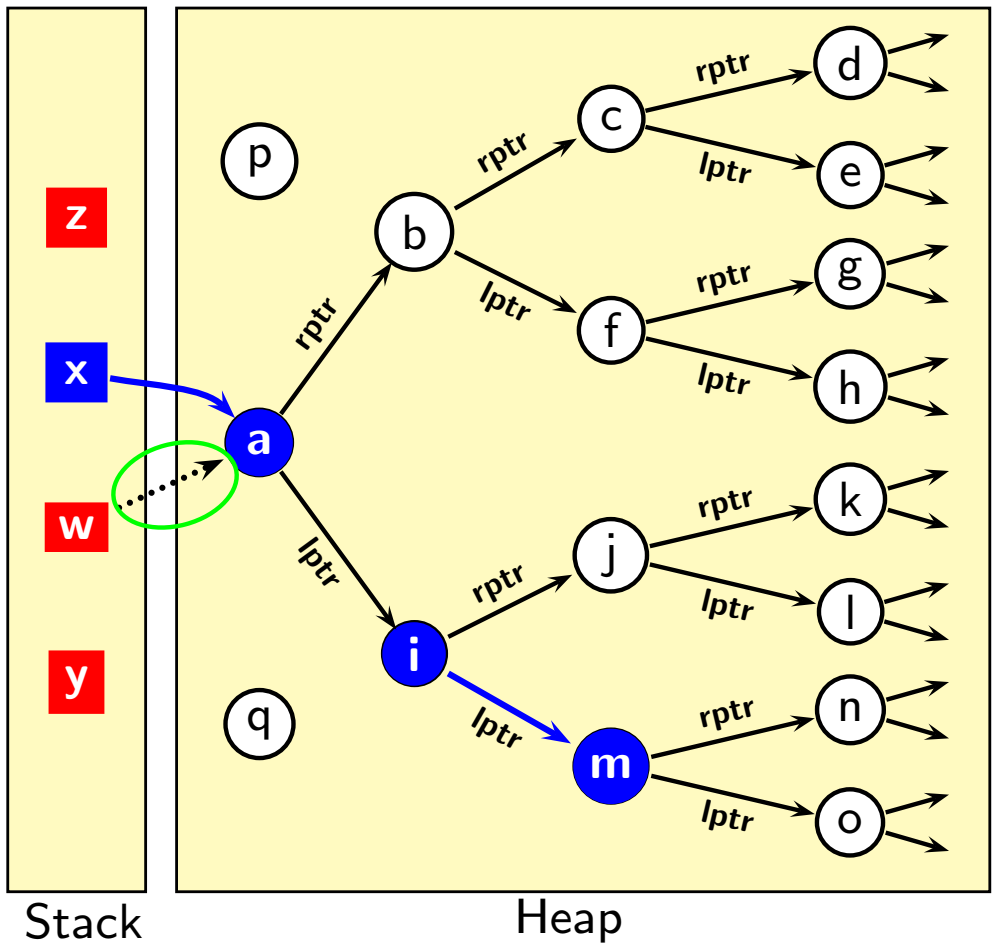
While loop is not executed even once



Our Solution

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

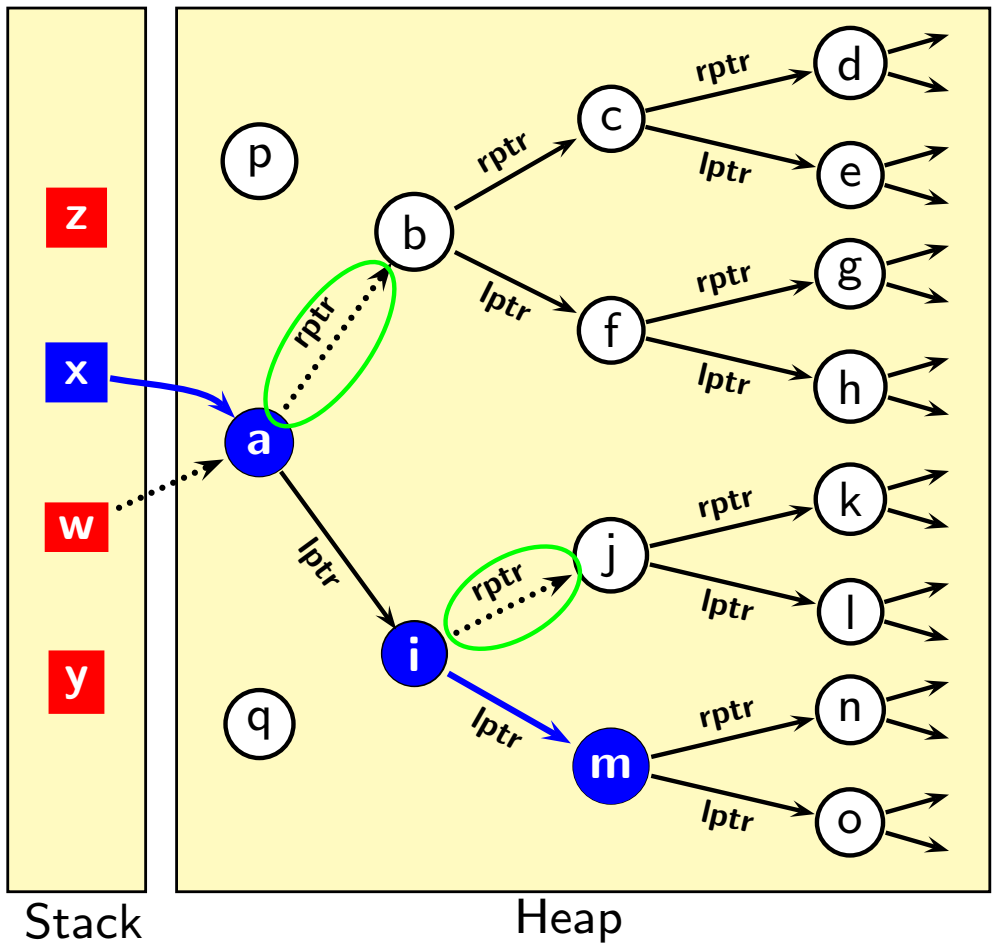
While loop is not executed even once



Our Solution

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

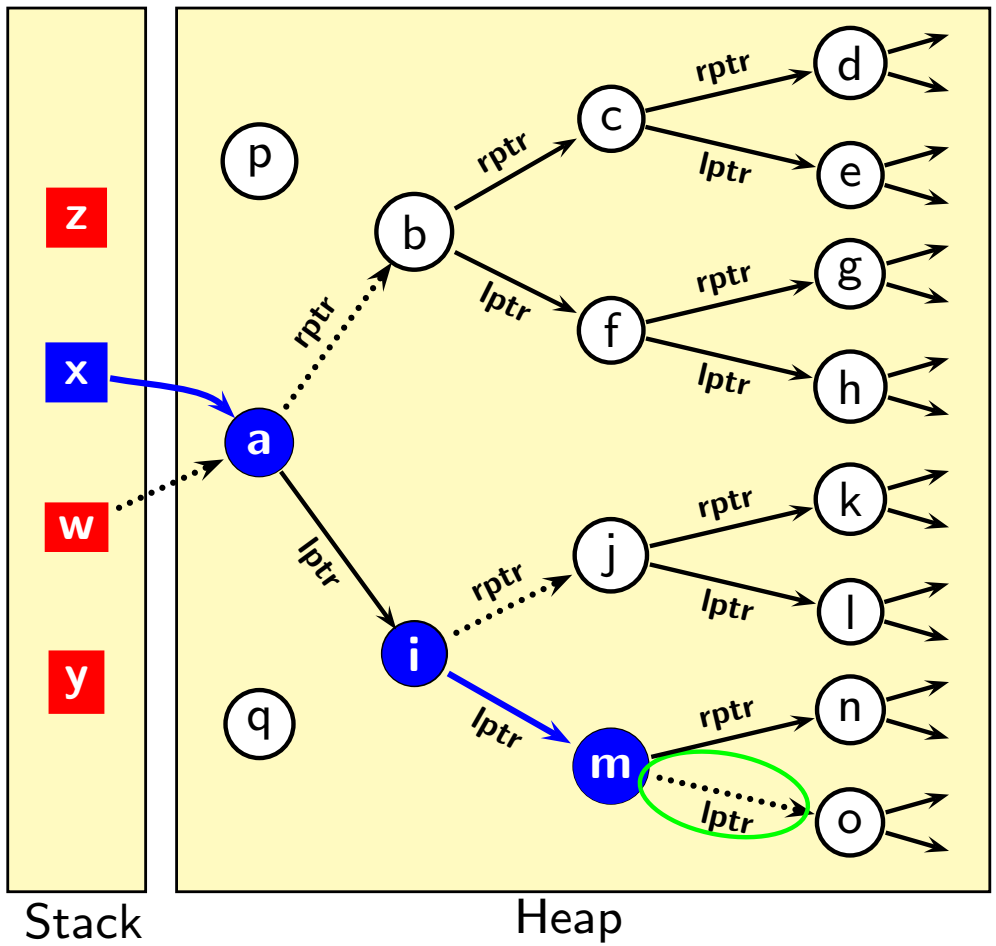
While loop is not executed even once



Our Solution

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

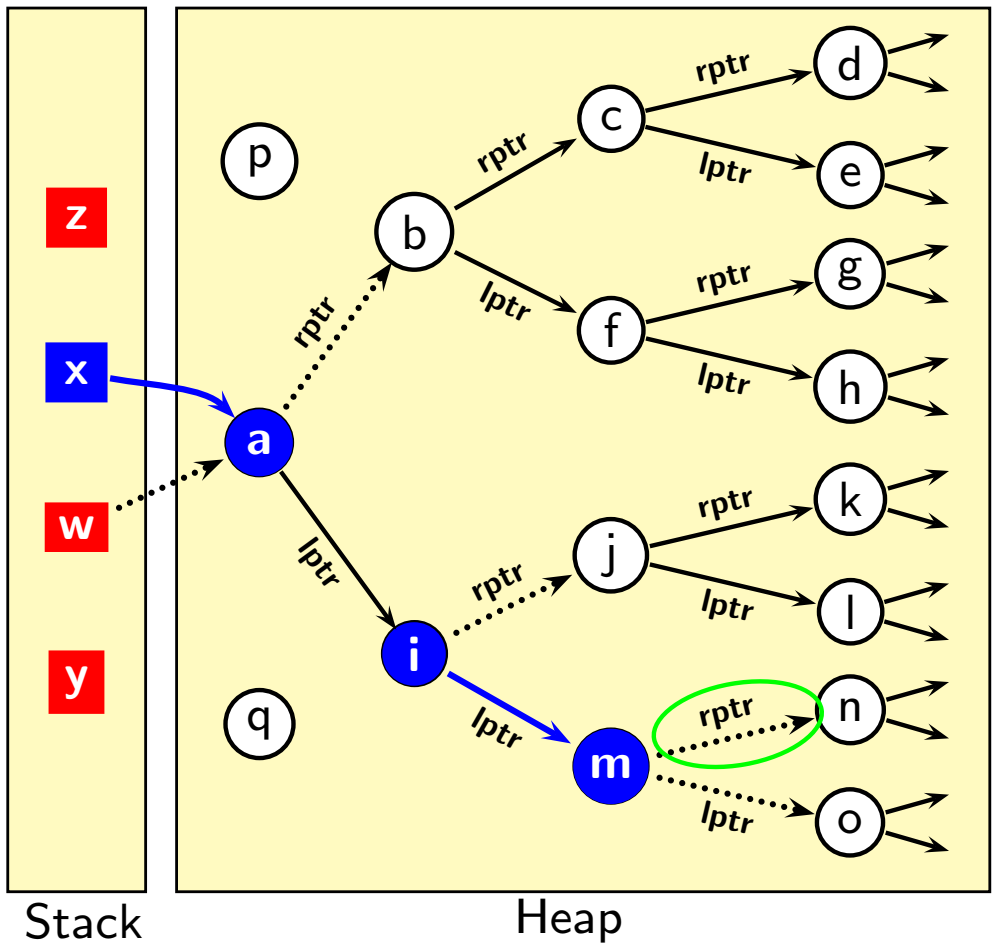
While loop is not executed even once



Our Solution

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

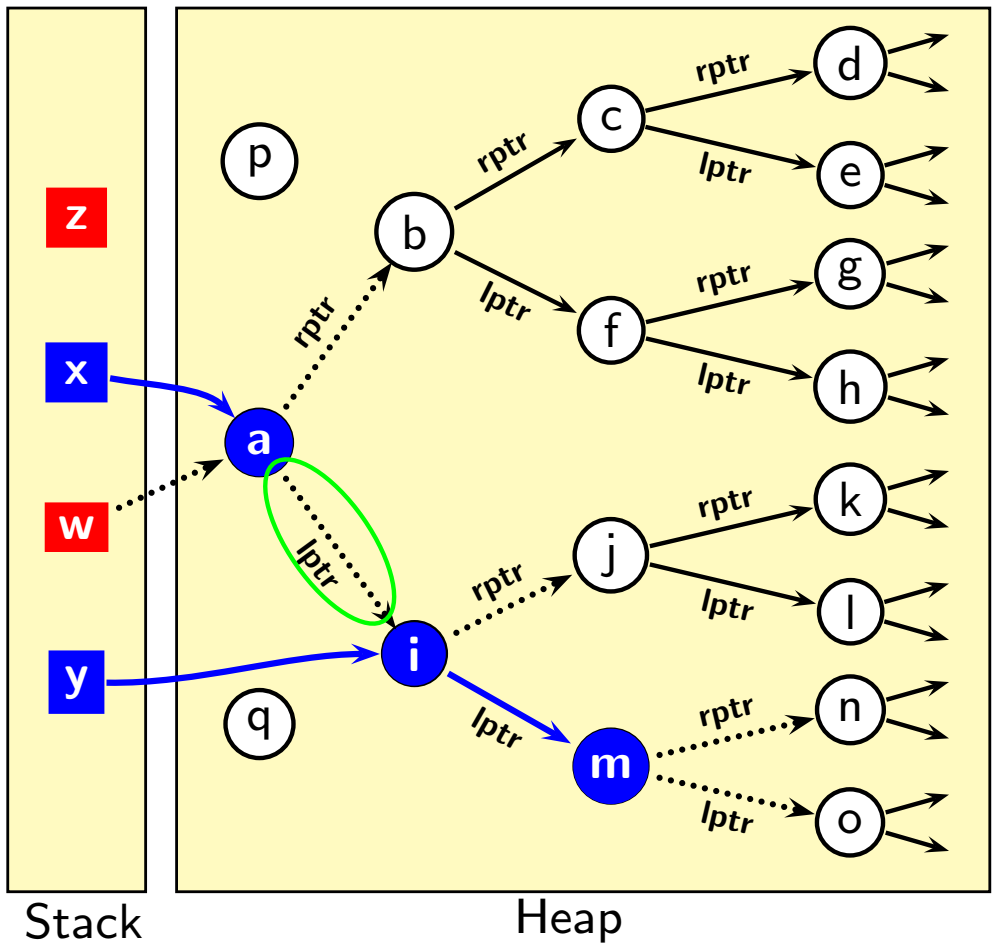
While loop is not executed even once



Our Solution

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

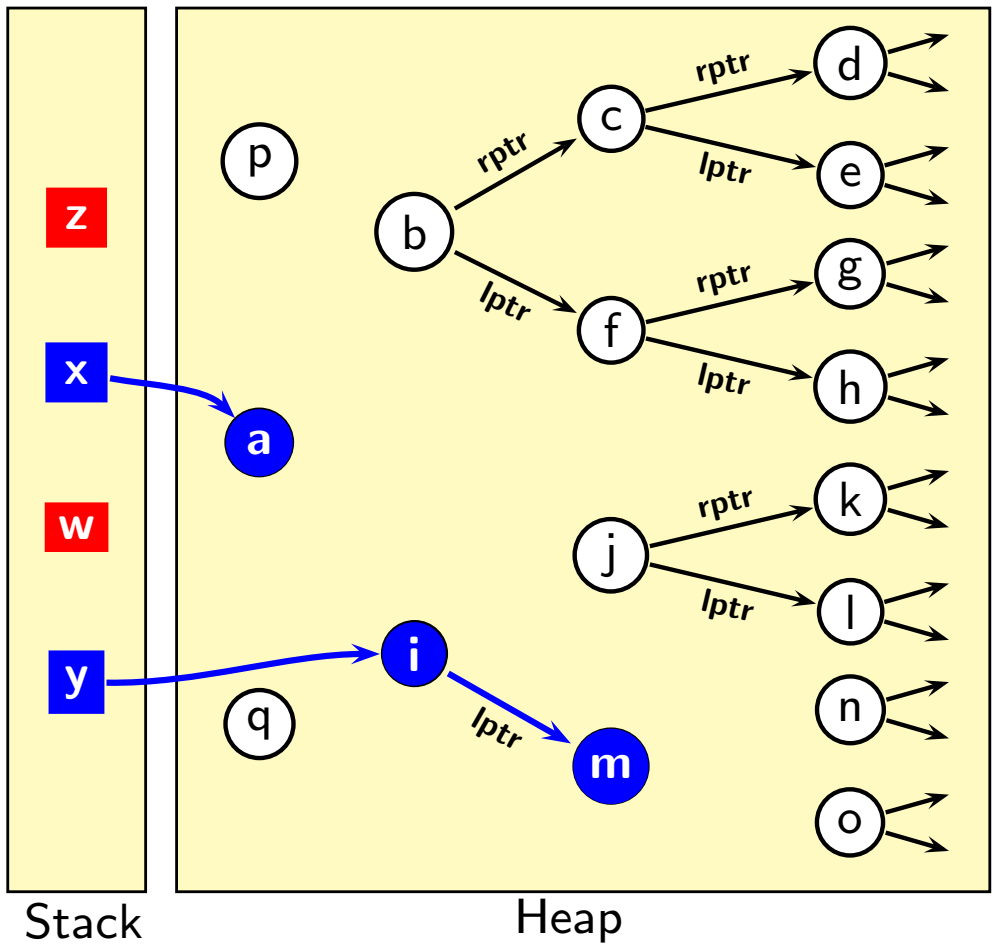
While loop is not executed even once



Our Solution

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

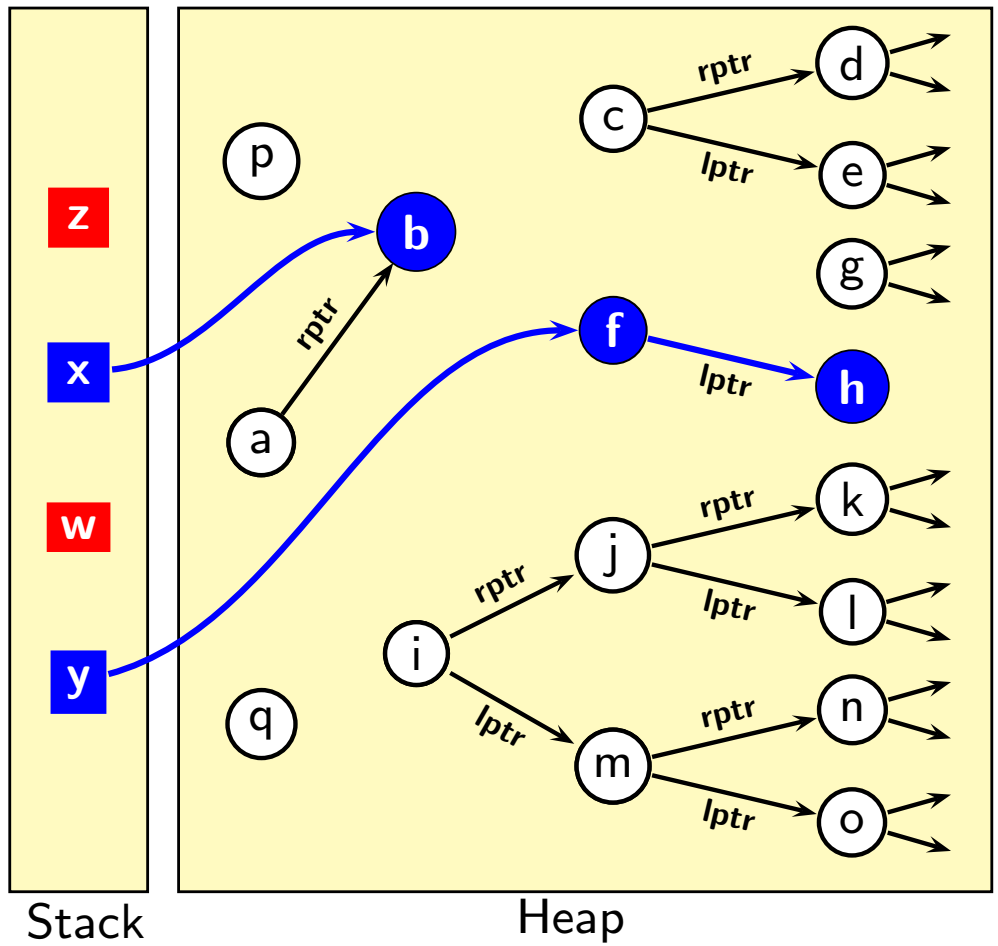
While loop is not executed even once



Our Solution

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

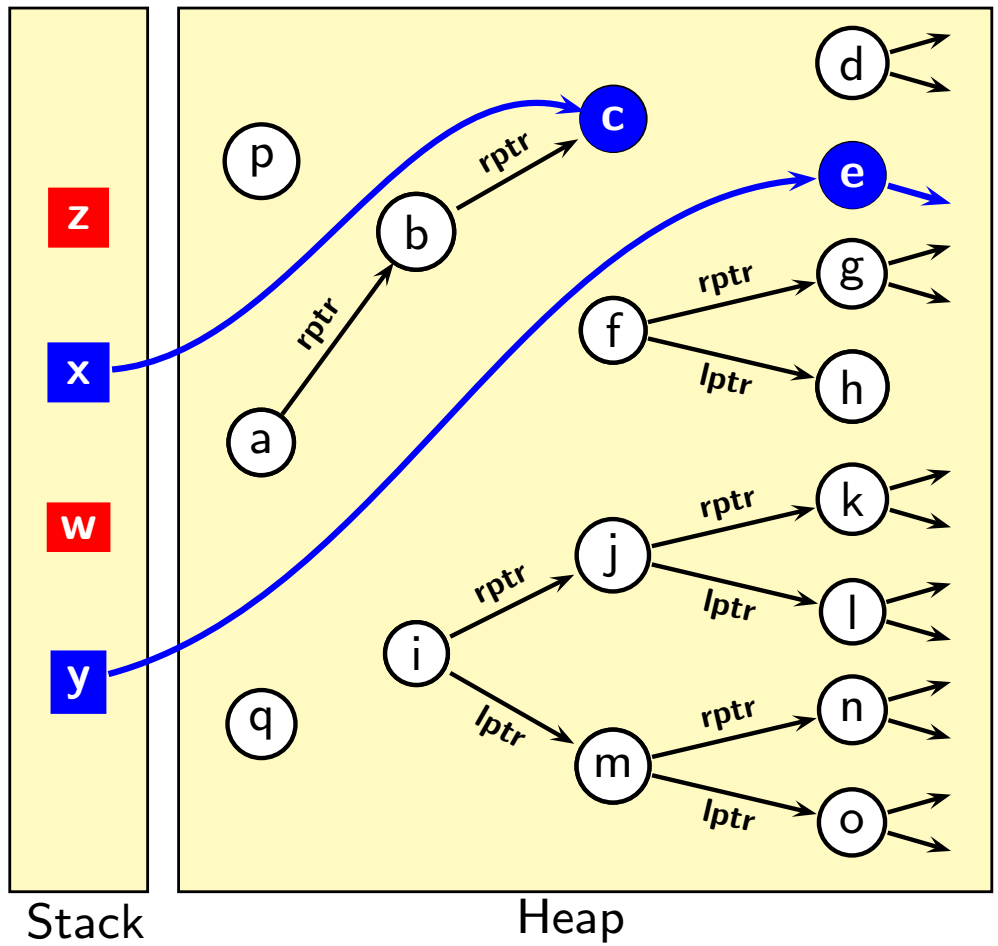
While loop is executed once



Our Solution

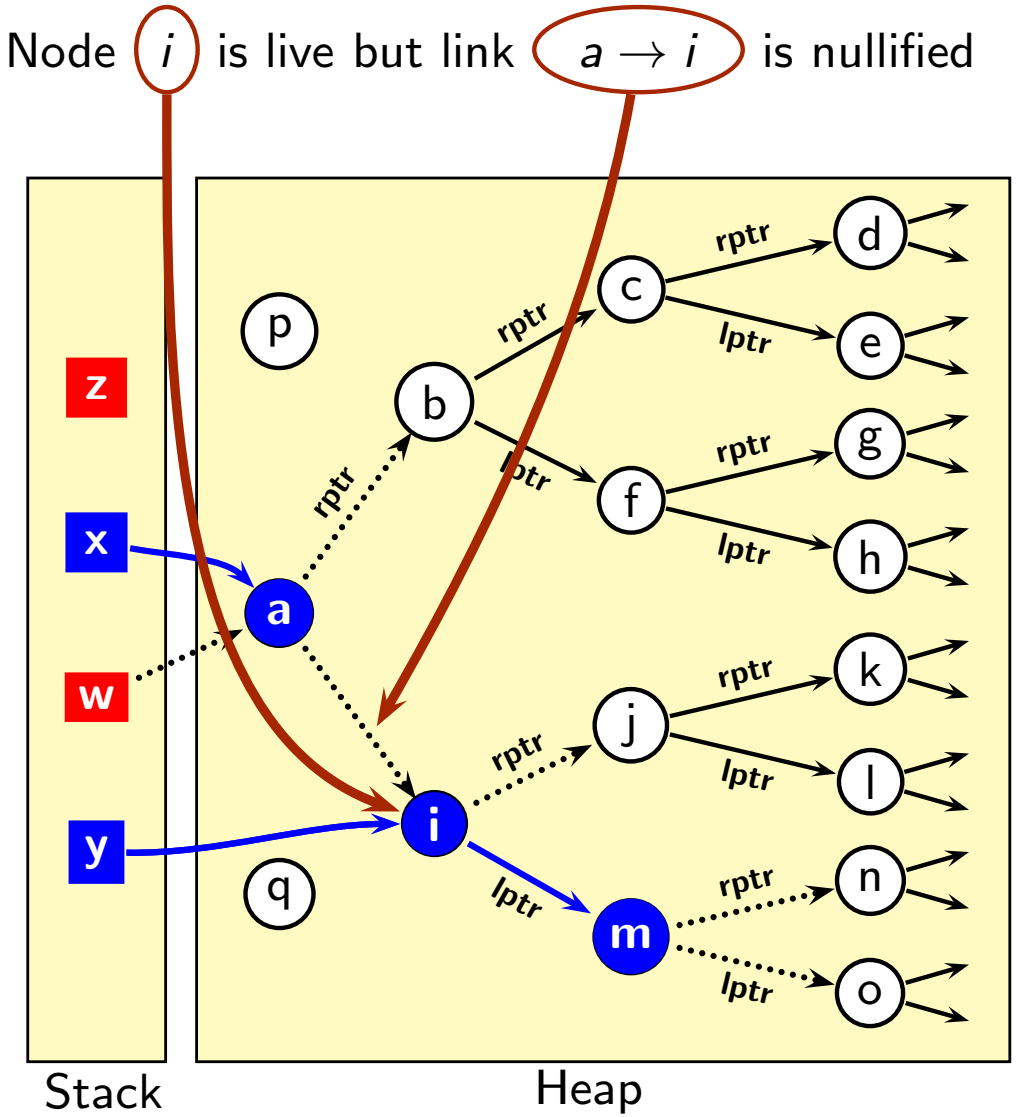
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

While loop is executed twice



Some Observations

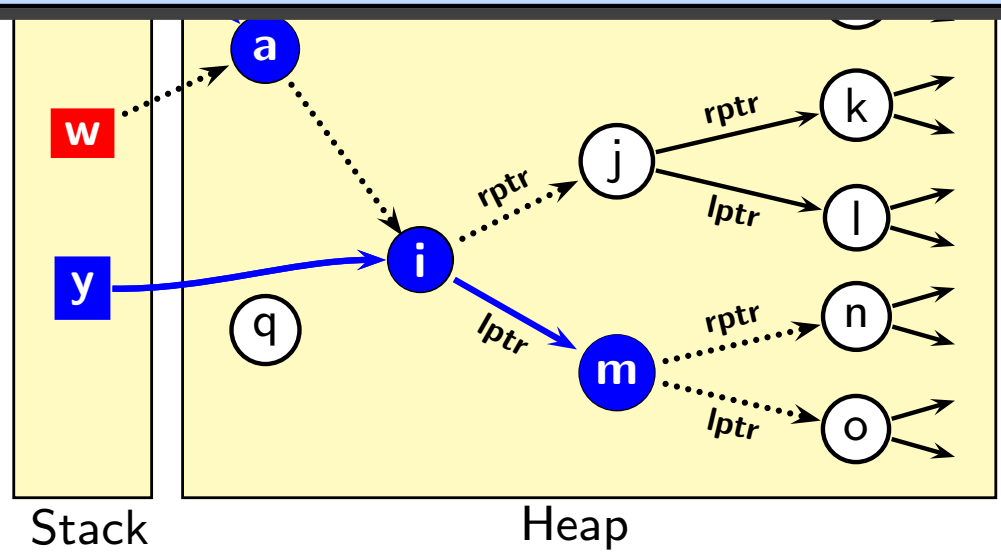
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```



Some Observations

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

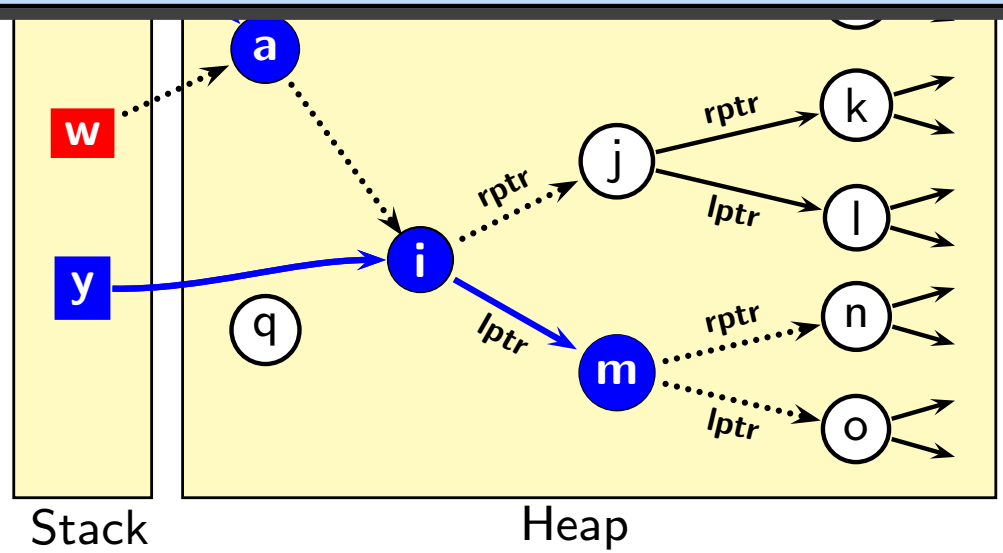
- The memory address that x holds when the execution reaches a given program point is not an invariant of program execution



Some Observations

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

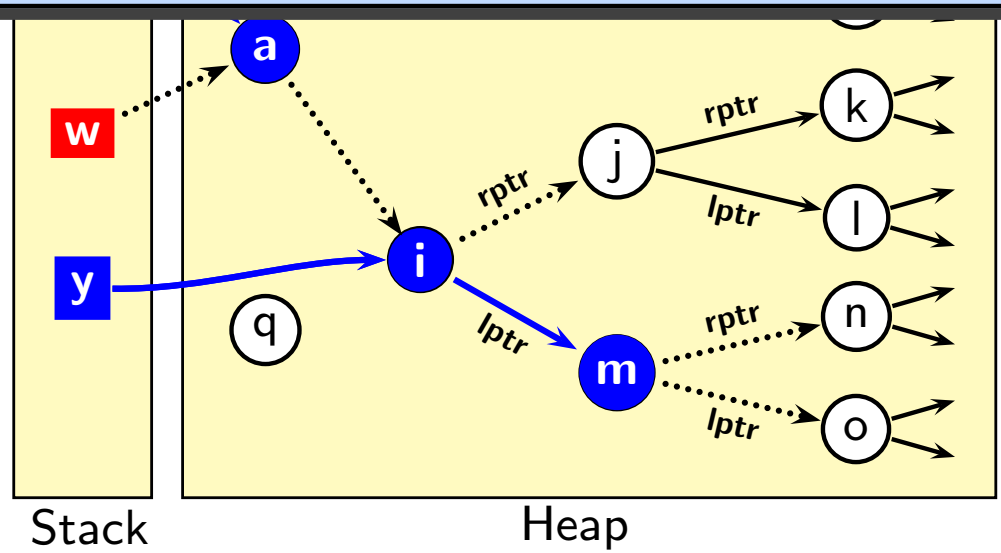
- The memory address that x holds when the execution reaches a given program point is not an invariant of program execution
- Whether we dereference lptr out of x or rptr out of x at a given program point is an invariant of program execution



Some Observations

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

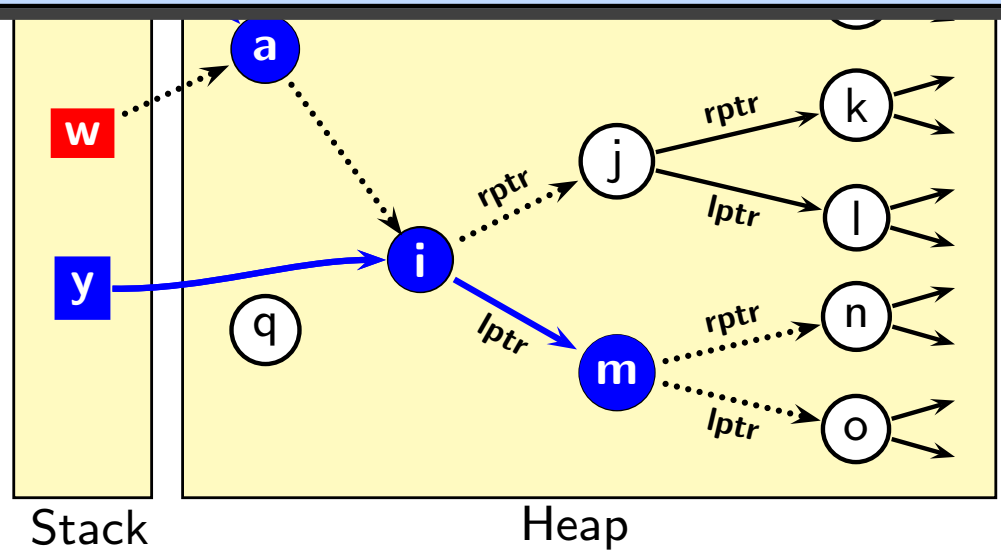
- The memory address that x holds when the execution reaches a given program point is not an invariant of program execution
- Whether we dereference lptr out of x or rptr out of x at a given program point is an invariant of program execution
- *A static analysis can discover only invariants*



Some Observations

```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr      }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New  class_of_z
    z.lptr = z.rptr = null
6   y = y.lptr
    y.lptr = y.rptr = null
7   z.sum = x.data + y.data
    x = y = z = null
```

- The memory address that x holds when the execution reaches a given program point is not an invariant of program execution
- Whether we dereference lptr out of x or rptr out of x at a given program point is an invariant of program execution
- *A static analysis can discover only some invariants*



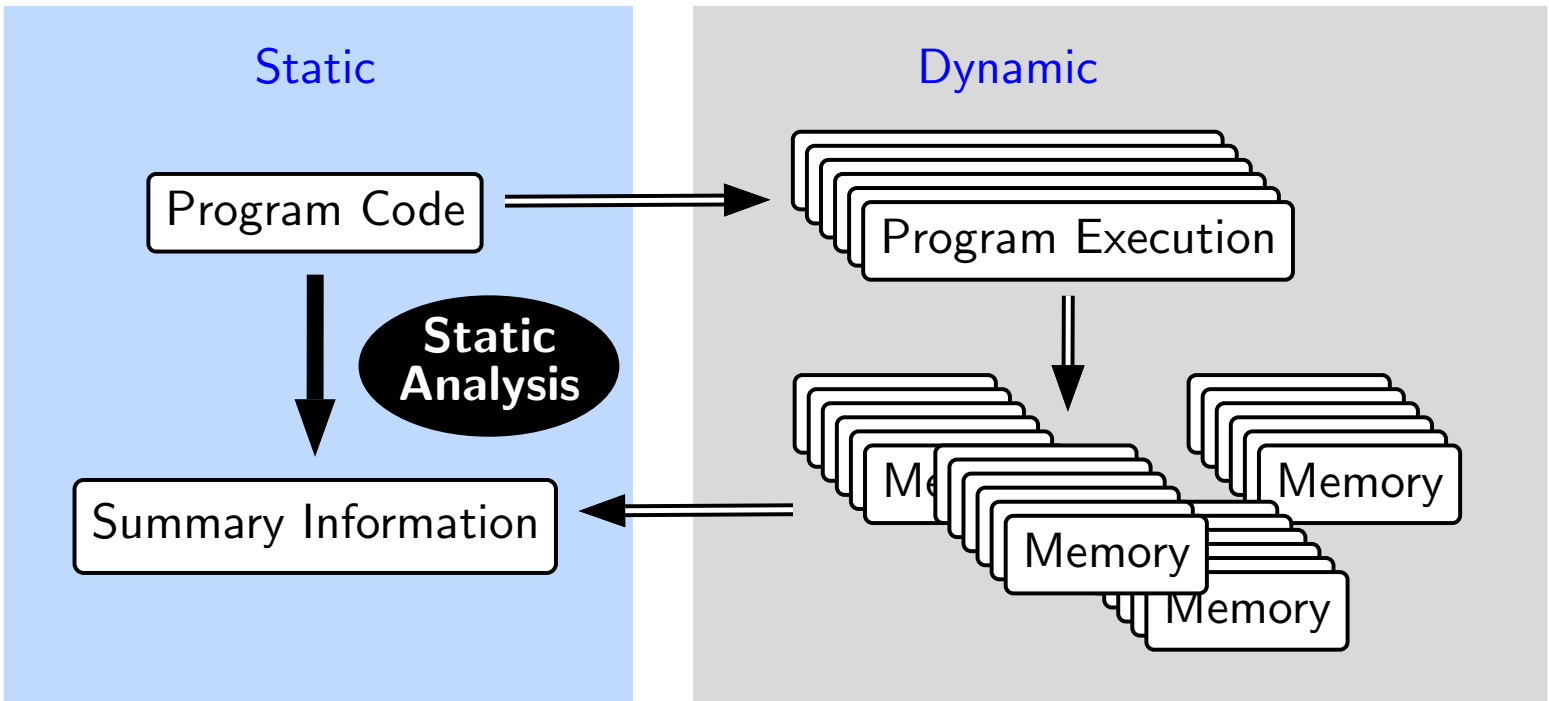
The Main Theme of (Static) Program Analysis

Constructing *suitable abstractions* for
sound & precise modelling of
runtime behaviour of programs
efficiently

The Main Theme of (Static) Program Analysis

Constructing *suitable abstractions* for
sound & precise modelling of
runtime behaviour of programs
efficiently

Abstract, Bounded, Single Instance | Concrete, Unbounded, Infinitely Many



Part 2

Soundness and Precision

Program Representation

- Three address code statements
  - Result, operator, operand1, operand2
  - Assignments, expressions, conditional jumps
  - Pointer expressions (including structure accesses)
    Features will be introduced as and when needed
- Control flow graph representation
  - Nodes represent maximal groups of statements
    devoid of any control transfer except fall through
  - Edges represent control transfers across basic blocks
  - A unique *Start* node and a unique *End* node
    Every node reachable from *Start*, and *End* reachable from every node
- Initially only intraprocedural programs

  Function calls brought in later

Motivating Example for Introducing Soundness and Precision

Example Program

```
int a;
int f(int b)
{  int c;
   c = a%2;
   b = - abs(b);
   while (b < c)
      b = b+1;
   if (b > 0)
      b = 0;
   return b;
}
```

Control Flow Graph

Motivating Example for Introducing Soundness and Precision

Example Program

Absolute

```
int a;
int f(int b)
{  int c;
   c = a%2;
   b = - abs(b);
   while (b < c)
      b = b+1;
   if (b > 0)
      b = 0;
   return b;
}
```

Control Flow Graph

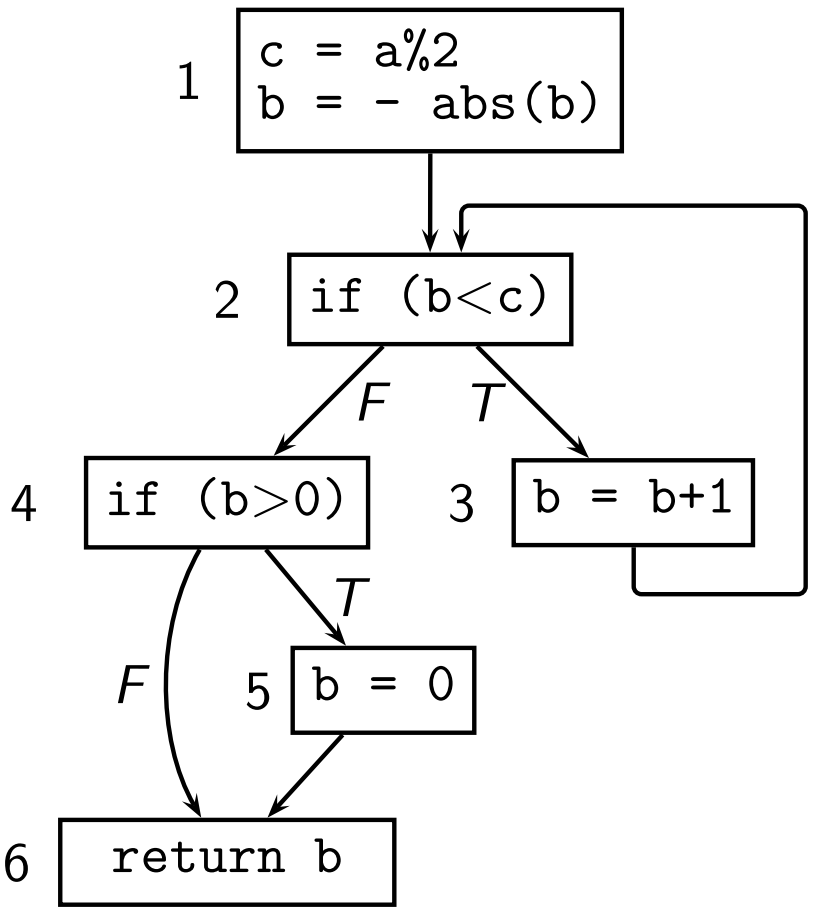
Motivating Example for Introducing Soundness and Precision

Example Program

Absolute

```
int a;
int f(int b)
{   int c;
    c = a%2;
    b = - abs(b);
    while (b < c)
        b = b+1;
    if (b > 0)
        b = 0;
    return b;
}
```

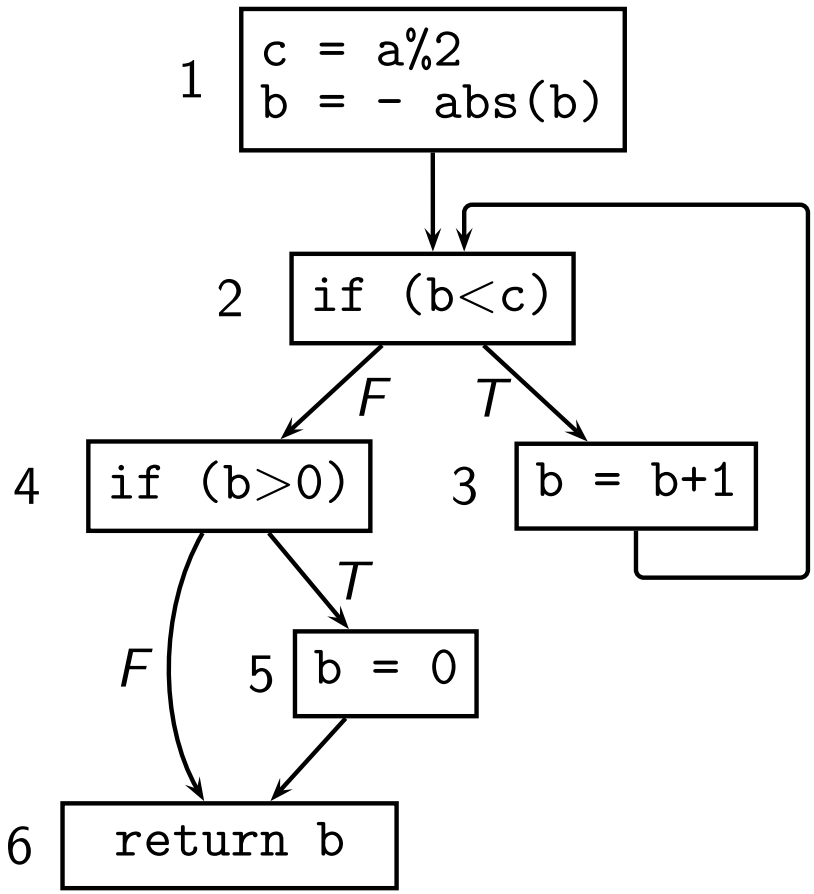
Control Flow Graph



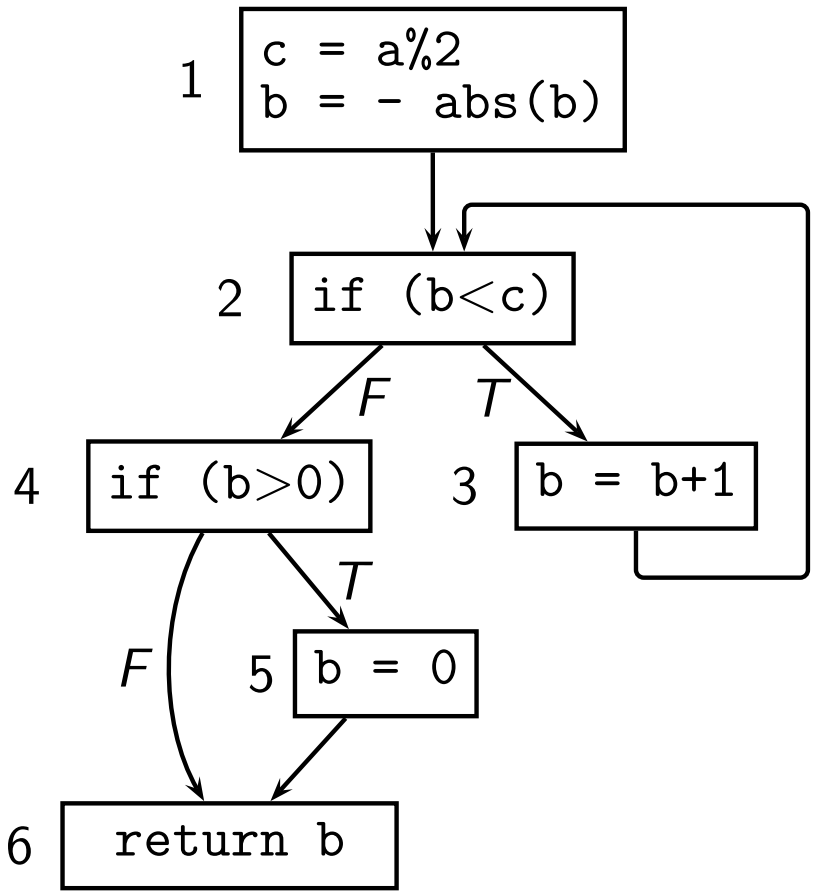
Execution Traces for Concrete Semantics (1)

- States
  - A *data* state: Variables $\rightarrow$ Values
  - A *program* state: (Program Point, A data state)
- Execution traces (or traces, for short)
  - Valid sequences of program states starting with a given initial state

Execution Traces for Concrete Semantics (2)



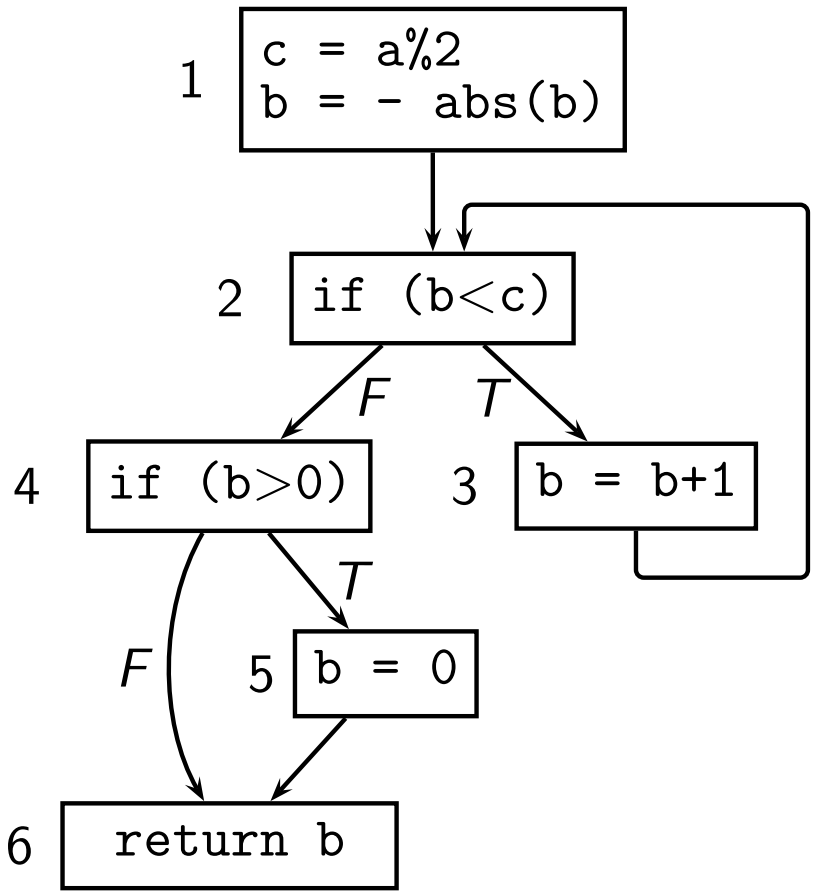
Execution Traces for Concrete Semantics (2)



Trace 1

	a	b	c
$Entry_1$,	5,	2,	7)
$Entry_2$,	5,	-2,	1)
$Entry_3$,	5,	-2,	1)
$Entry_2$,	5,	-1,	1)
$Entry_3$,	5,	-1,	1)
$Entry_2$,	5,	0,	1)
$Entry_3$,	5,	0,	1)
$Entry_2$,	5,	1,	1)
$Entry_4$,	5,	1,	1)
$Entry_5$,	5,	1,	1)
$Entry_6$,	5,	0,	1)

Execution Traces for Concrete Semantics (2)



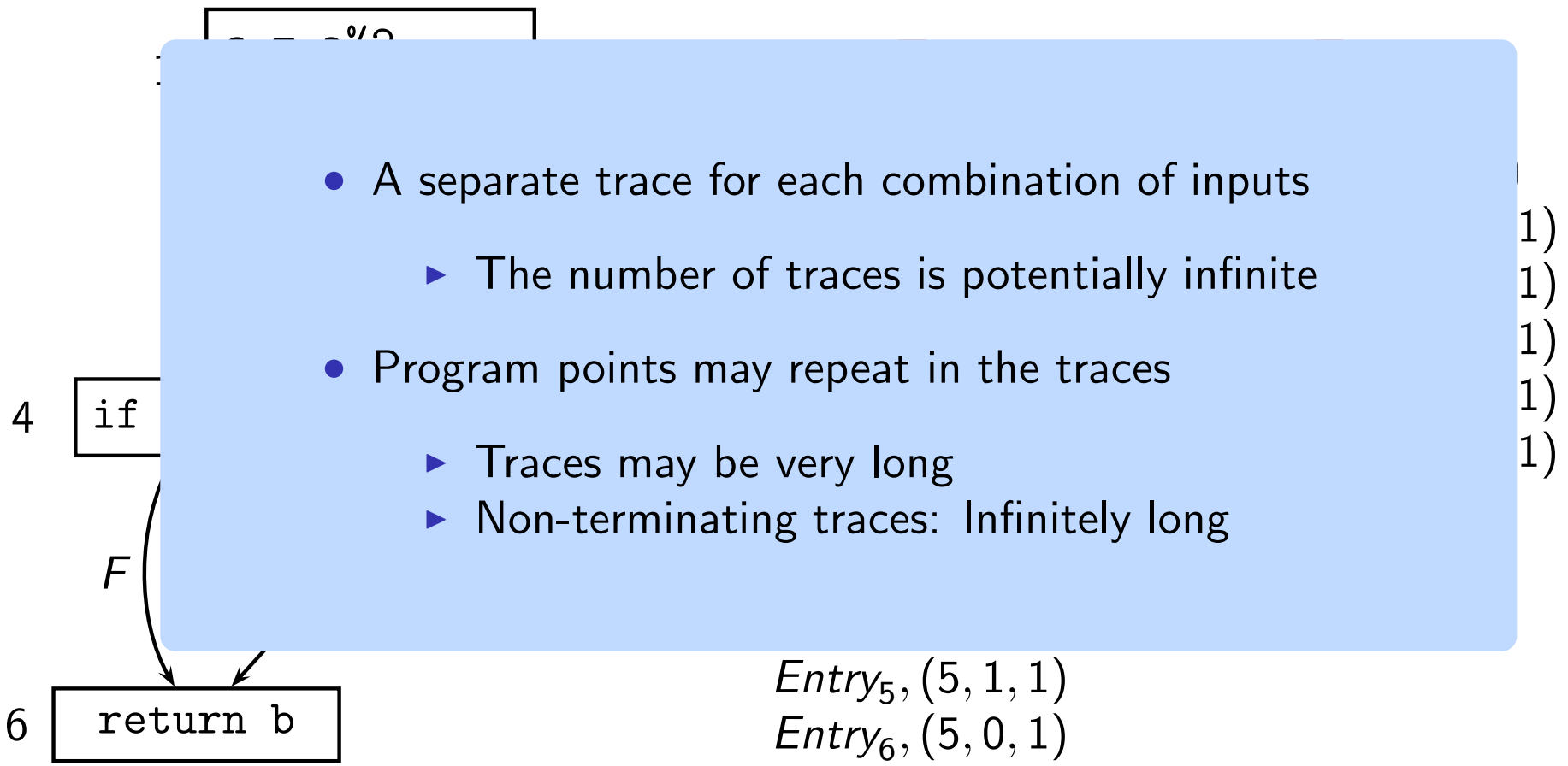
Trace 1

	a	b	c
$Entry_1$	5	2	7
$Entry_2$	5	−2	1
$Entry_3$	5	−2	1
$Entry_2$	5	−1	1
$Entry_3$	5	−1	1
$Entry_2$	5	0	1
$Entry_3$	5	0	1
$Entry_2$	5	1	1
$Entry_4$	5	1	1
$Entry_5$	5	1	1
$Entry_6$	5	0	1

Trace 2

	a	b	c
$Entry_1$	−5	−2	8
$Entry_2$	−5	−2	−1
$Entry_3$	−5	−2	−1
$Entry_2$	−5	−1	−1
$Entry_4$	−5	−1	−1
$Entry_6$	−5	−1	−1

Execution Traces for Concrete Semantics (2)



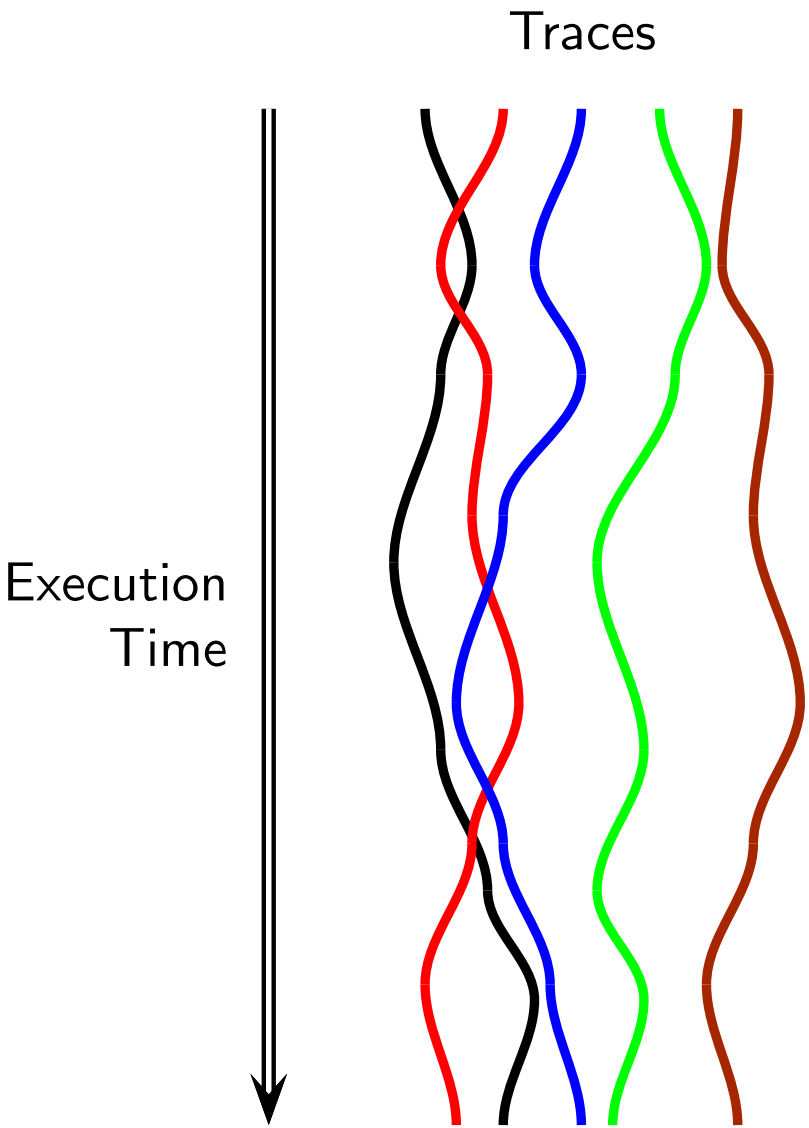
- A separate trace for each combination of inputs
  - The number of traces is potentially infinite
- Program points may repeat in the traces
  - Traces may be very long
  - Non-terminating traces: Infinitely long

Abstract States

A static analysis computes abstract states

- The values are abstract values and are decided by the analysis
- An analysis may record values for other program entities such as expressions, statements, procedures etc.

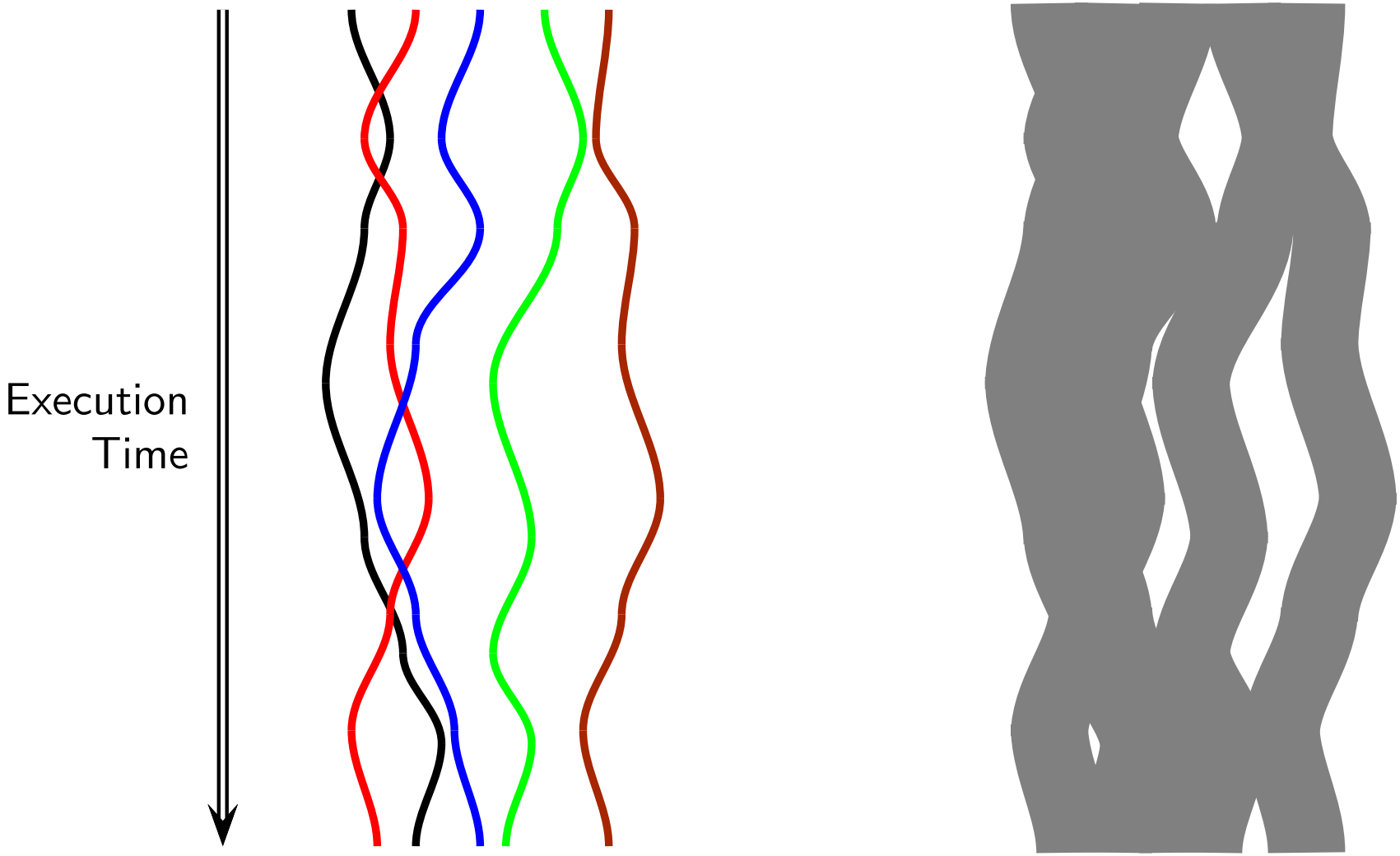
Static Analysis Computes Abstractions of Traces (1)



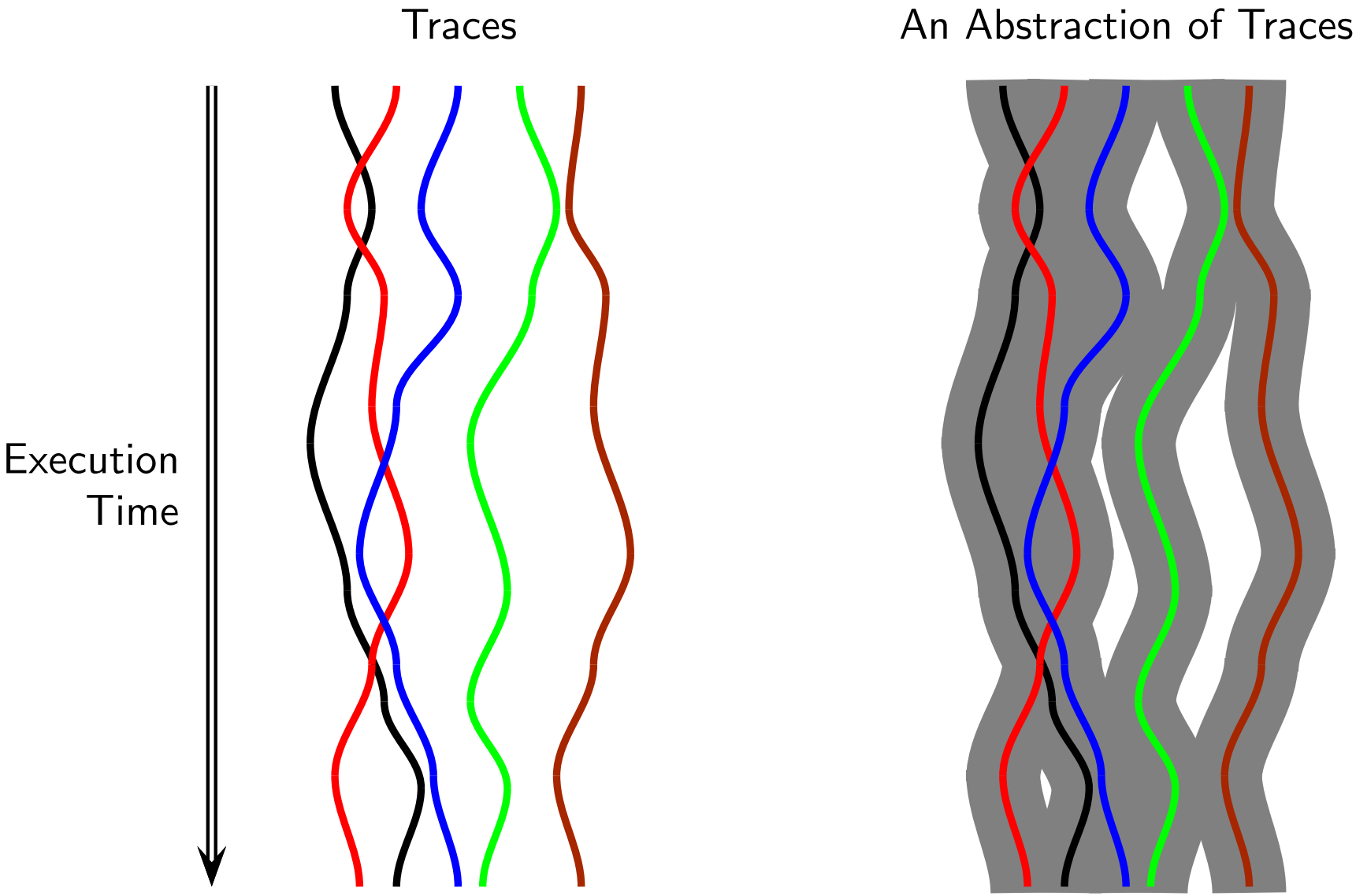
Static Analysis Computes Abstractions of Traces (1)

Traces

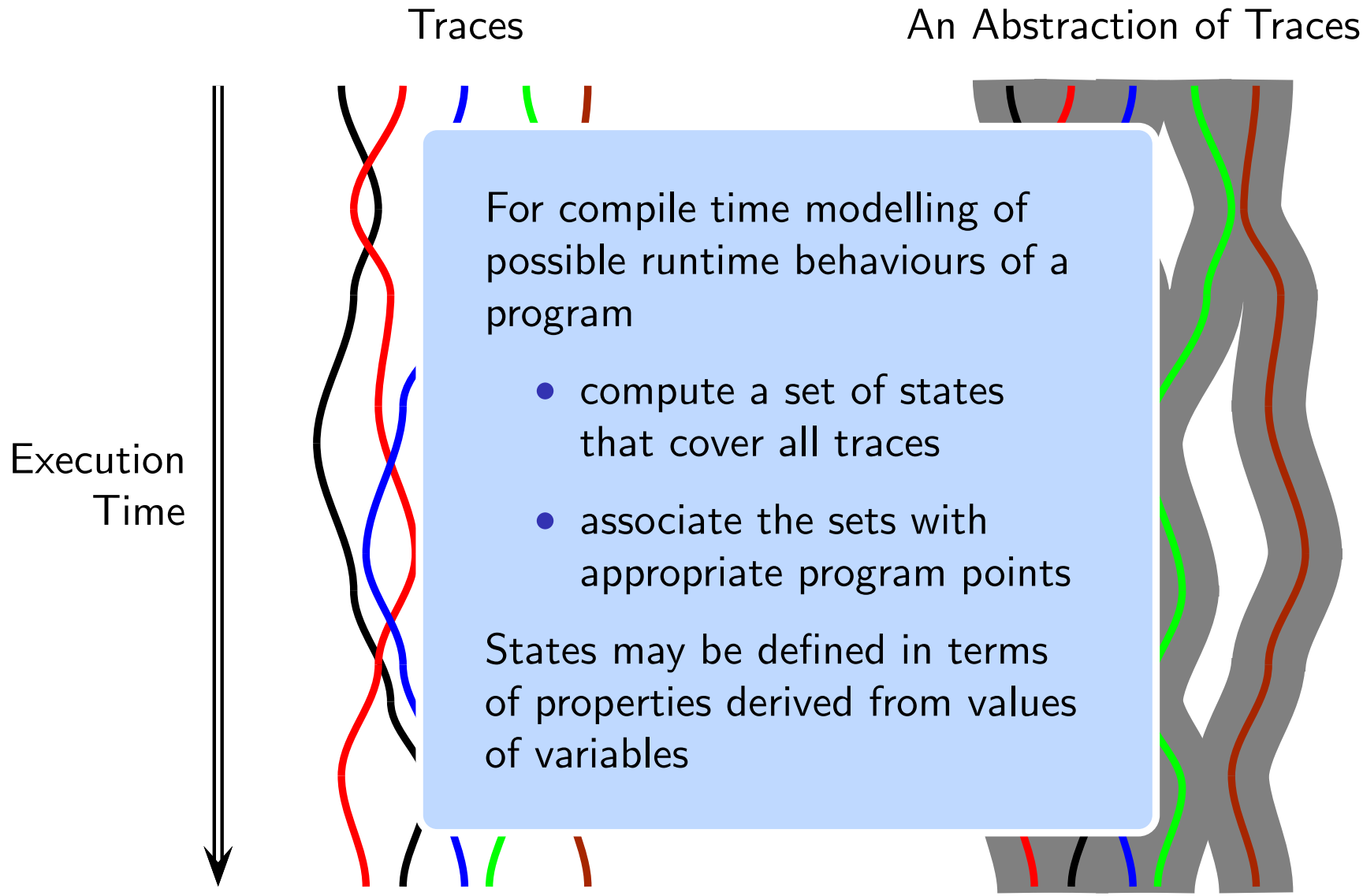
An Abstraction of Traces



Static Analysis Computes Abstractions of Traces (1)



Static Analysis Computes Abstractions of Traces (1)



Static Analysis Computes Abstractions of Traces (2)

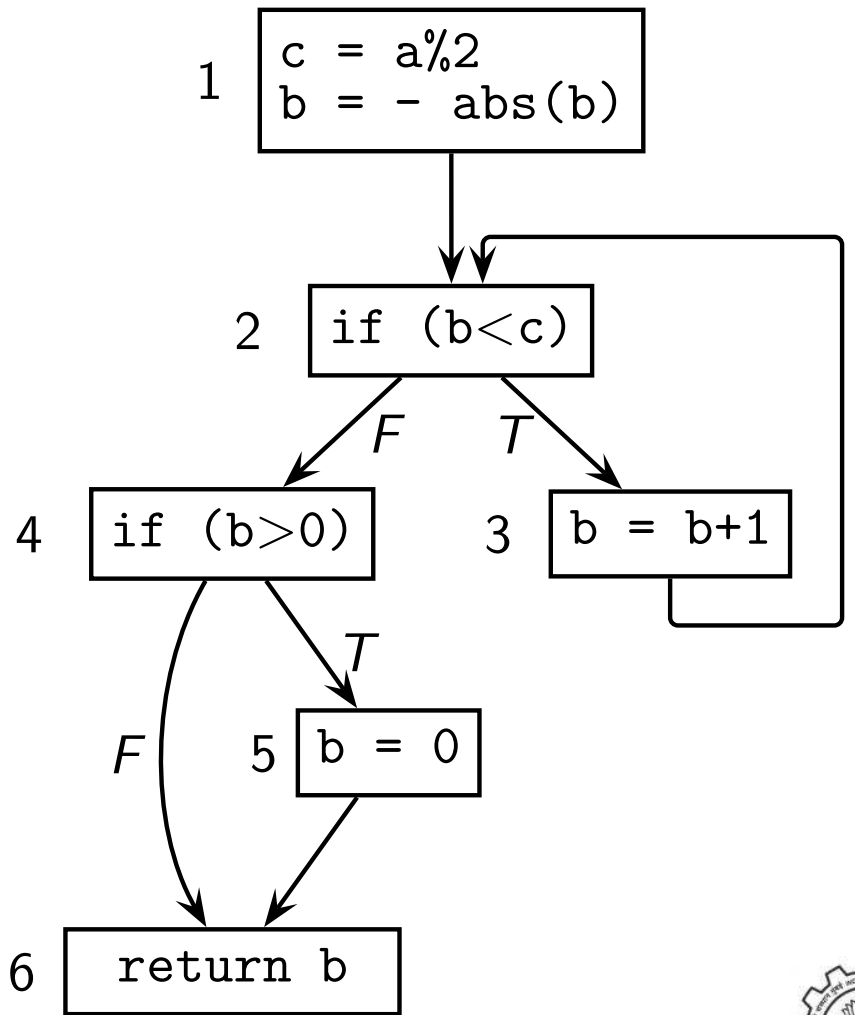
A possible static abstraction using sets

Trace 1

	a	b	c
$Entry_1$,	5,	2,	7)
$Entry_2$,	5,	−2,	1)
$Entry_3$,	5,	−2,	1)
$Entry_2$,	5,	−1,	1)
$Entry_3$,	5,	−1,	1)
$Entry_2$,	5,	0,	1)
$Entry_3$,	5,	0,	1)
$Entry_2$,	5,	1,	1)
$Entry_4$,	5,	1,	1)
$Entry_5$,	5,	1,	1)
$Entry_6$,	5,	0,	1)

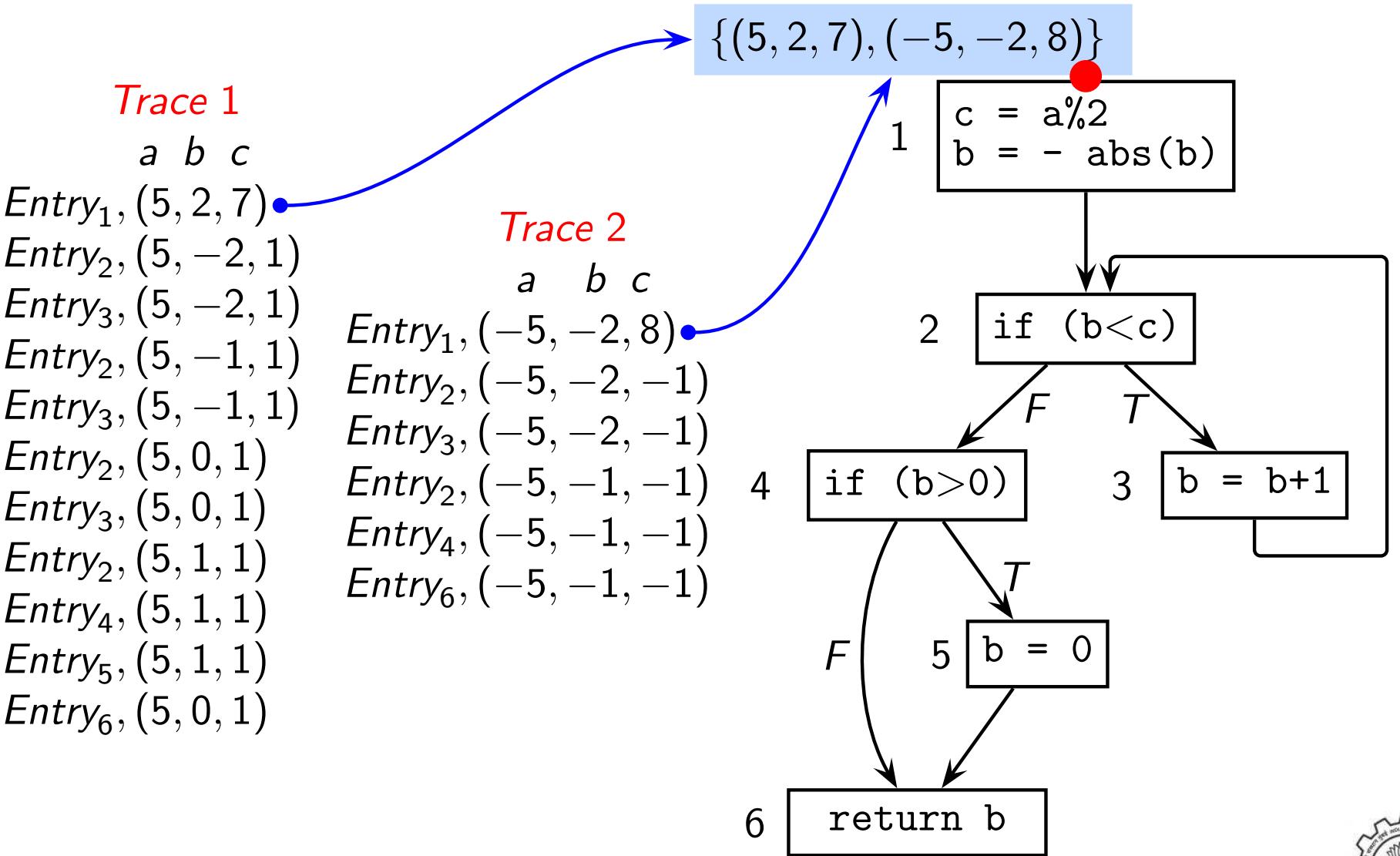
Trace 2

	a	b	c
$Entry_1$,	−5,	−2,	8)
$Entry_2$,	−5,	−2,	−1)
$Entry_3$,	−5,	−2,	−1)
$Entry_2$,	−5,	−1,	−1)
$Entry_4$,	−5,	−1,	−1)
$Entry_6$,	−5,	−1,	−1)



Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets



Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

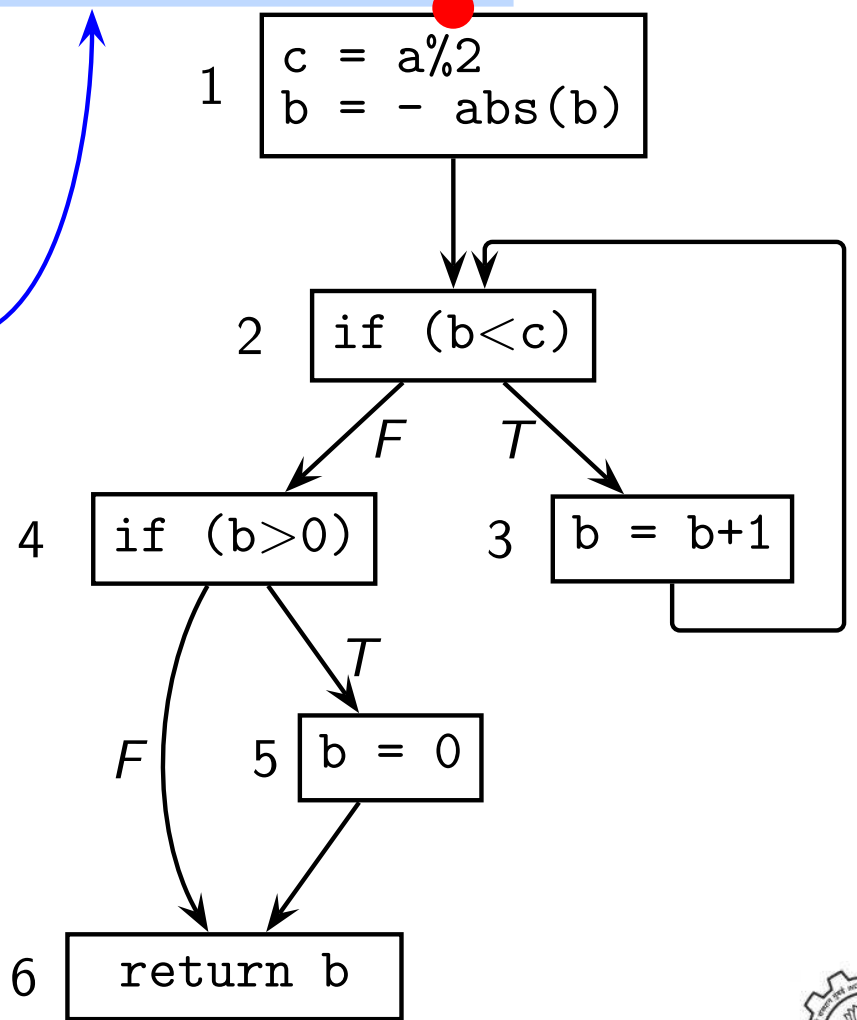
$a=\{-5,5\}, b=\{-2,2\}, c=\{7,8\}$

Trace 1

	a	b	c
$Entry_1$	5	2	7
$Entry_2$	5	−2	1
$Entry_3$	5	−2	1
$Entry_2$	5	−1	1
$Entry_3$	5	−1	1
$Entry_2$	5	0	1
$Entry_3$	5	0	1
$Entry_2$	5	1	1
$Entry_4$	5	1	1
$Entry_5$	5	1	1
$Entry_6$	5	0	1

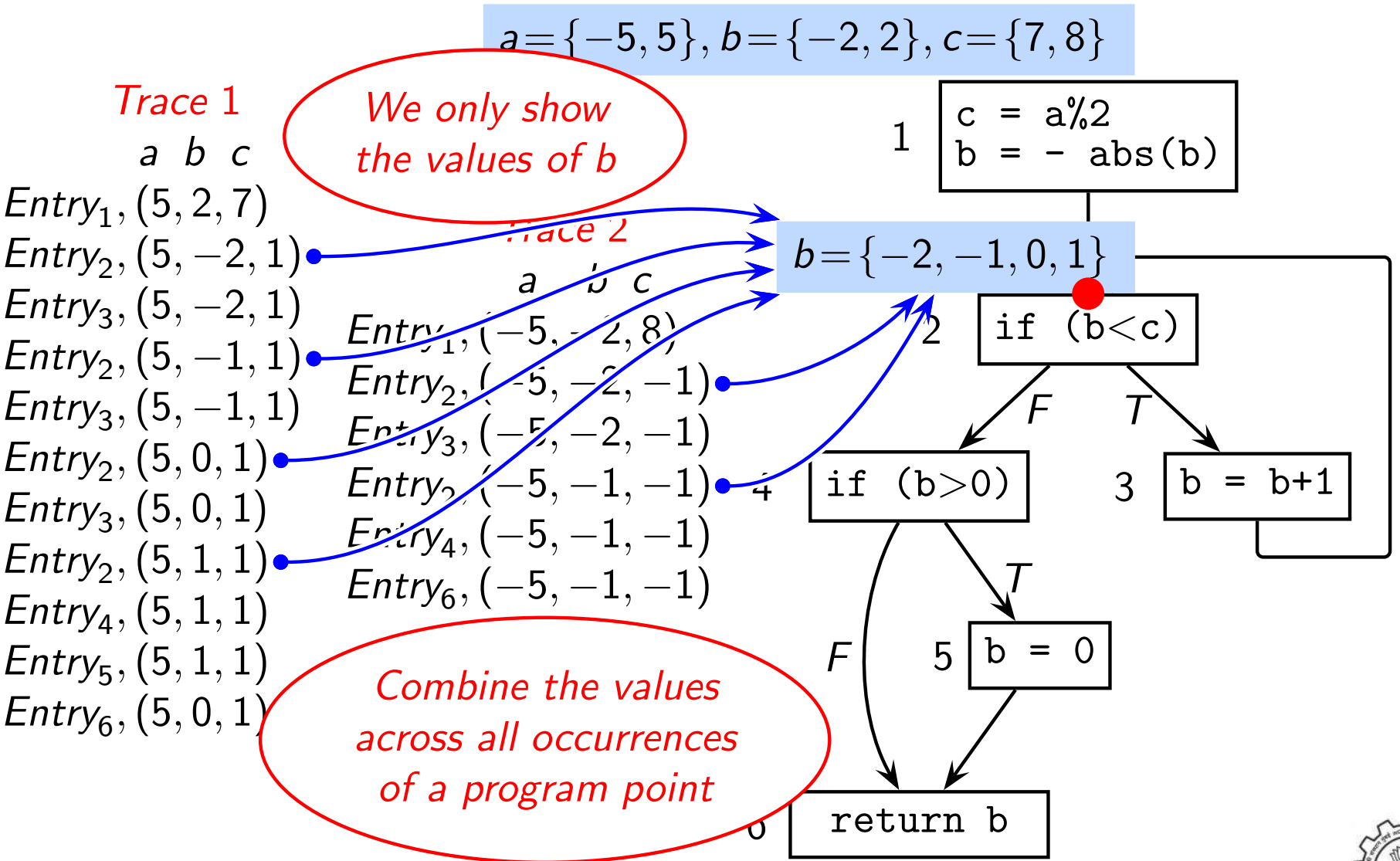
Trace 2

	a	b	c
$Entry_1$	−5	−2	8
$Entry_2$	−5	−2	−1
$Entry_3$	−5	−2	−1
$Entry_2$	−5	−1	−1
$Entry_4$	−5	−1	−1
$Entry_6$	−5	−1	−1



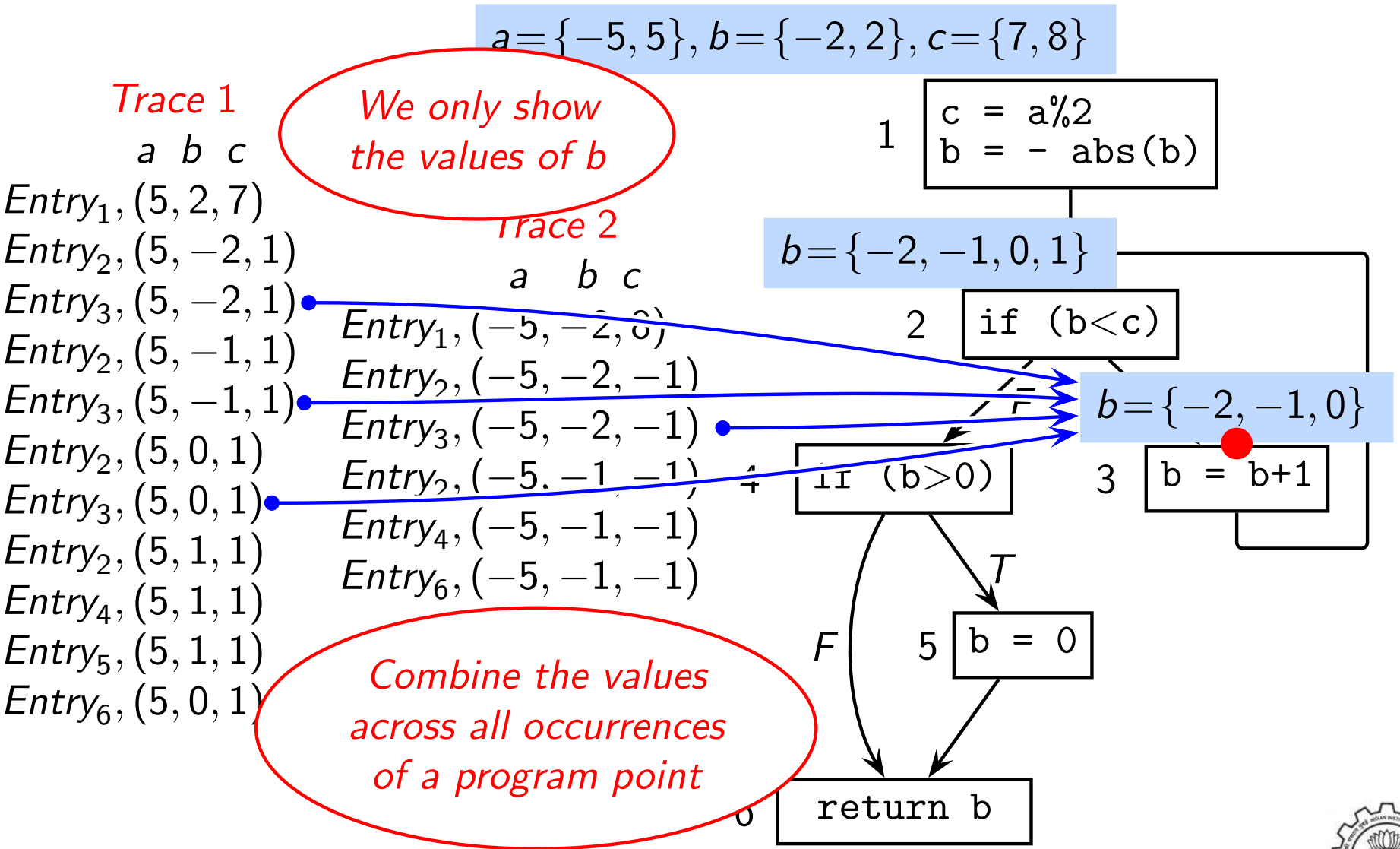
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets



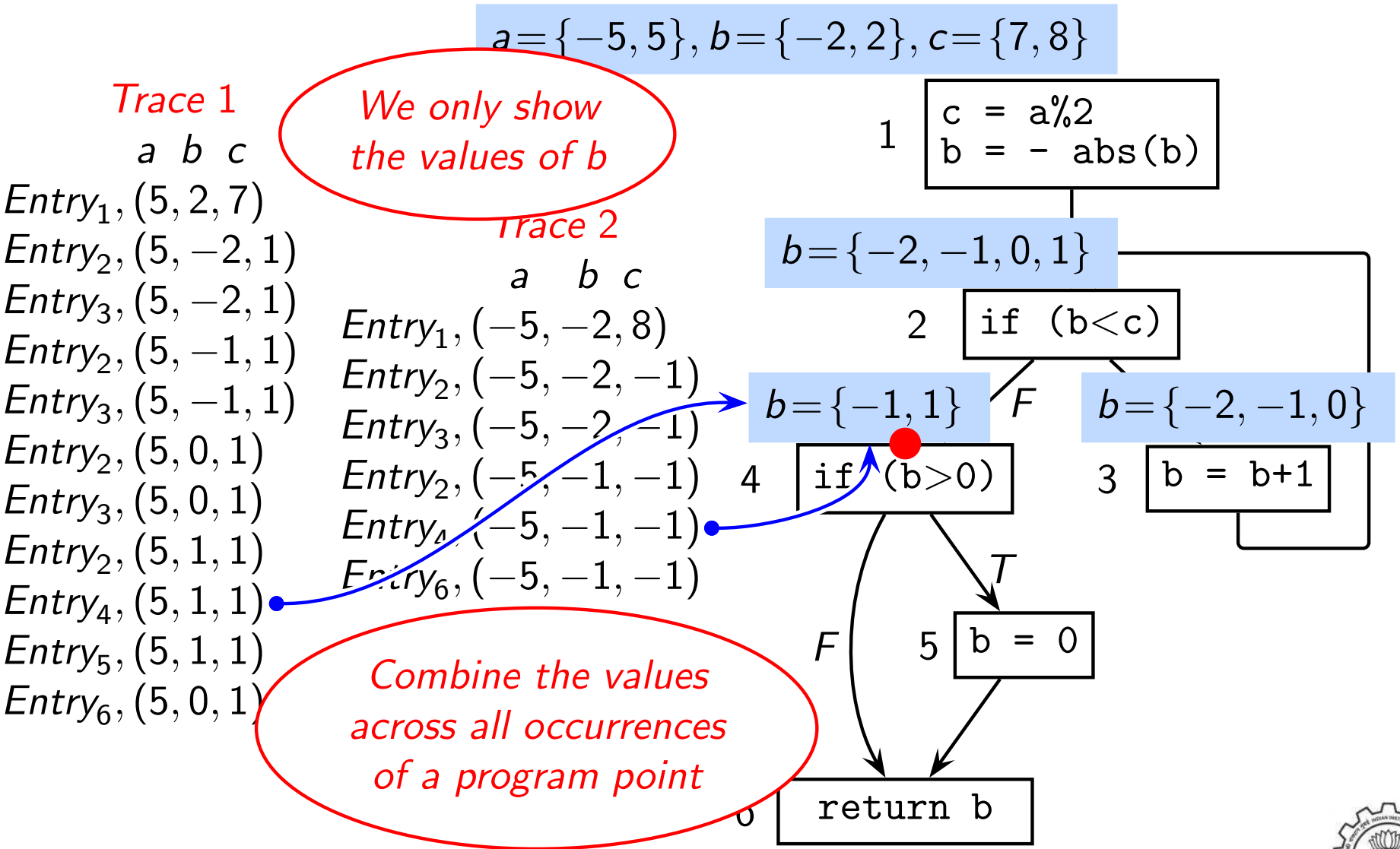
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets



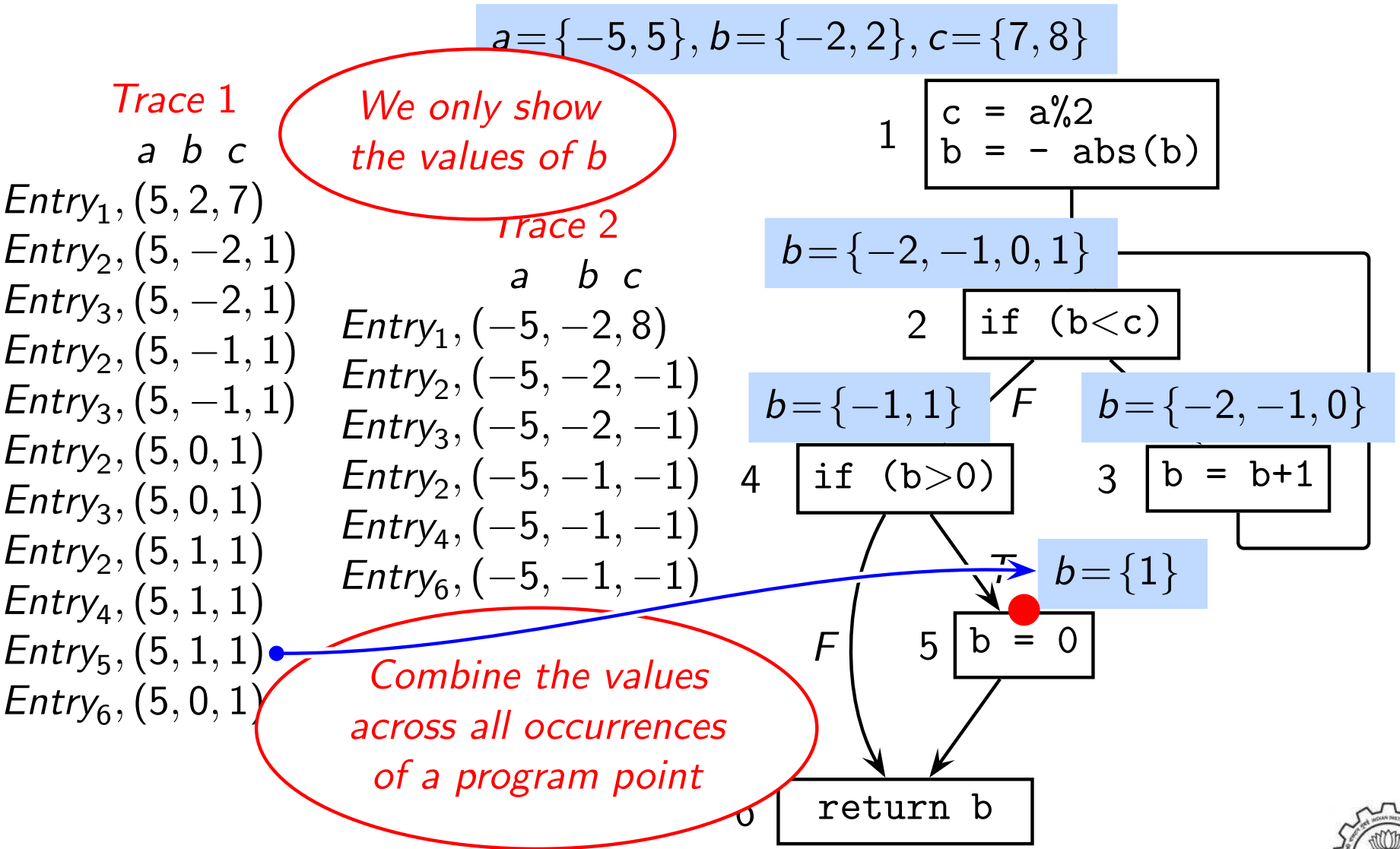
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets



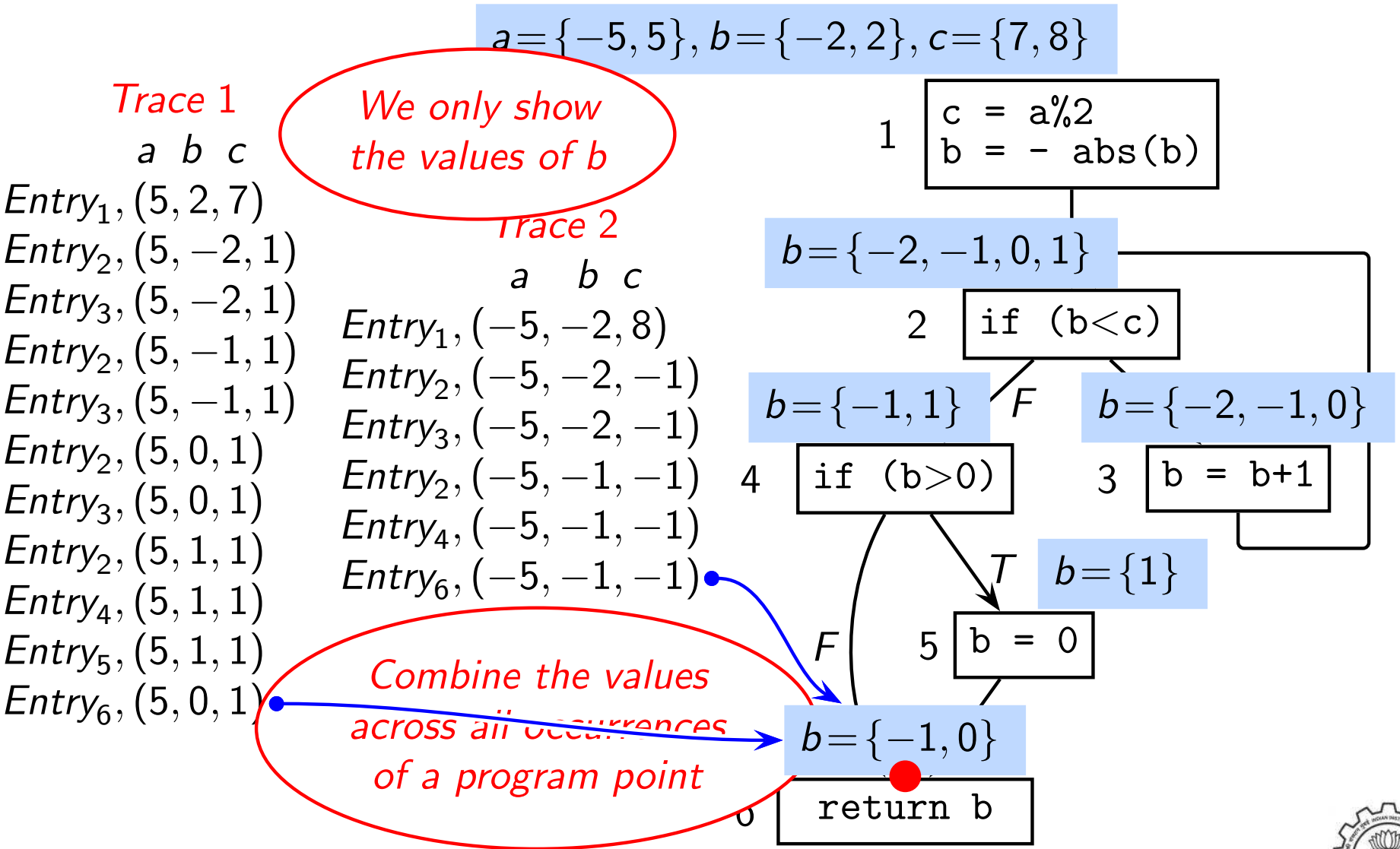
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets



Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets



Computing Static Abstraction for Liveness of Variables

At a program point p

$a \mapsto 1 \Rightarrow a$ is live at p

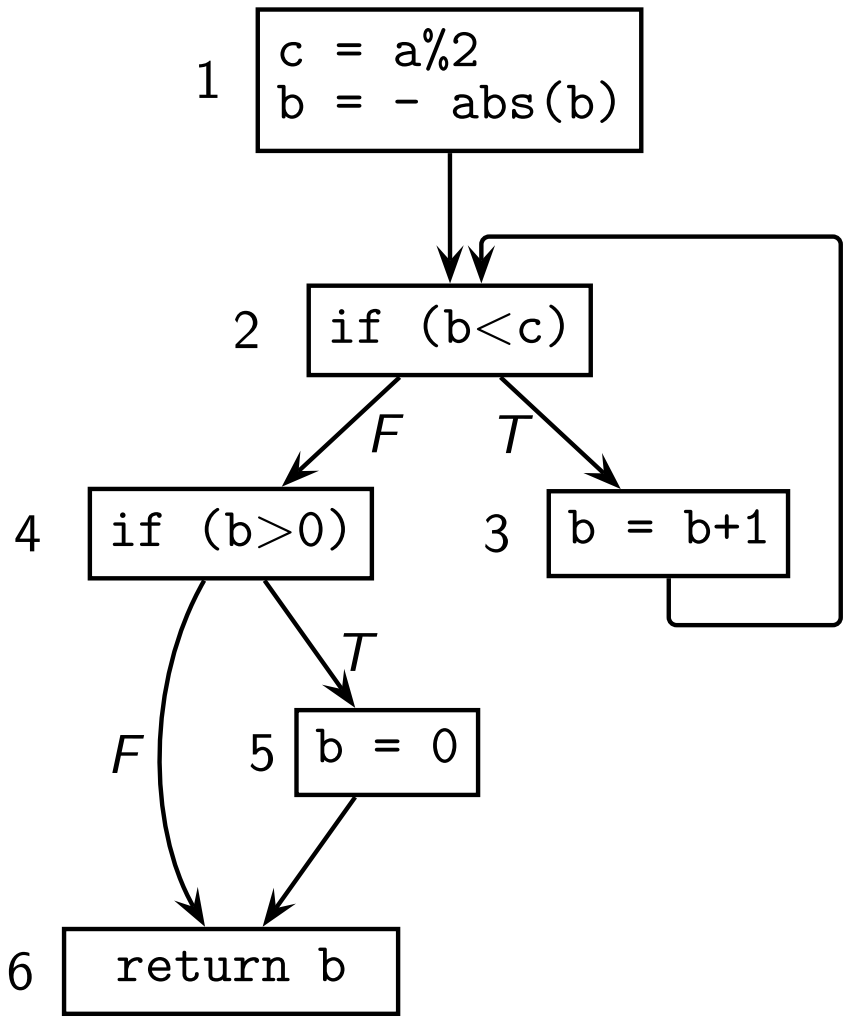
$a \mapsto 0 \Rightarrow a$ is not live at p

Trace 1

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_5$	0	0	0
$Entry_6$	0	1	0

Trace 2

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	0	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_6$	0	1	0



Computing Static Abstraction for Liveness of Variables

At a program point p

$a \mapsto 1 \Rightarrow a$ is live at p

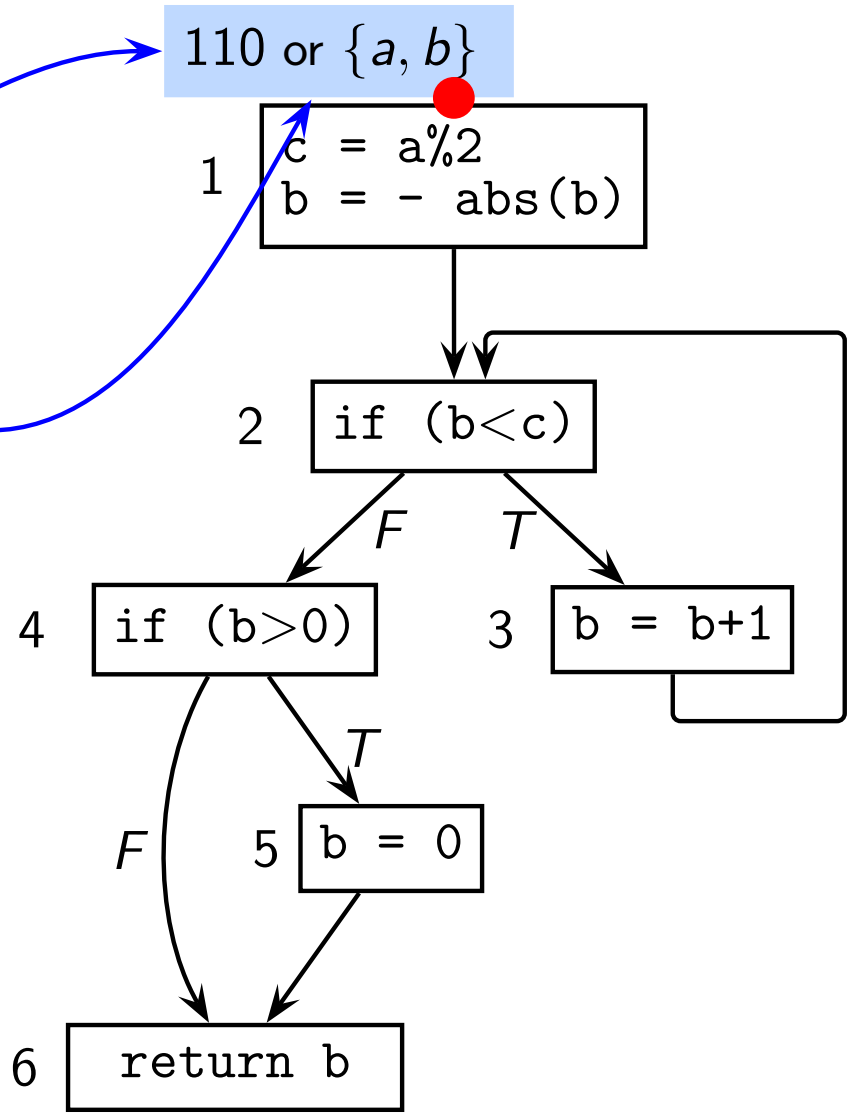
$a \mapsto 0 \Rightarrow a$ is not live at p

Trace 1

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_5$	0	0	0
$Entry_6$	0	1	0

Trace 2

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	0	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_6$	0	1	0

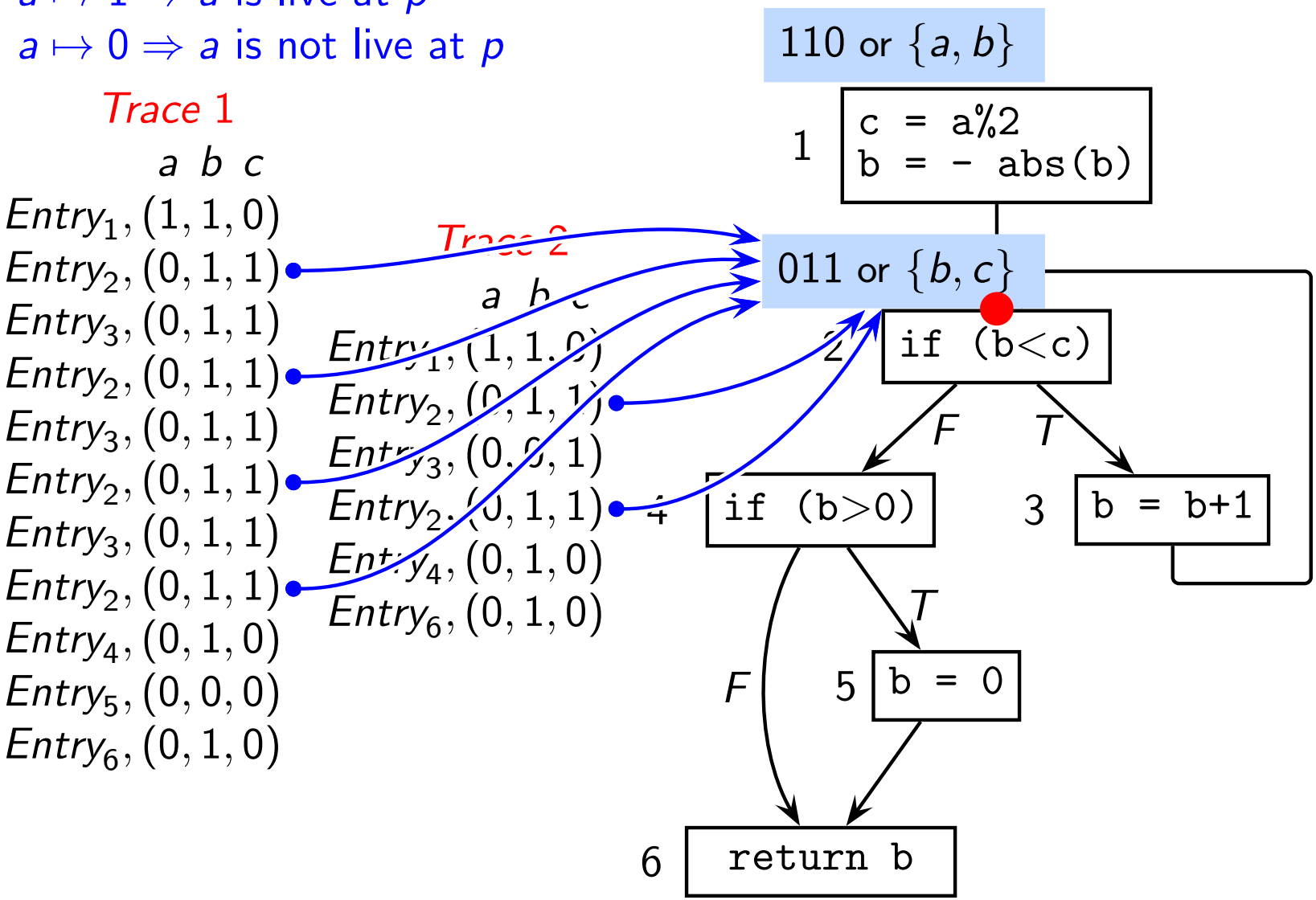


Computing Static Abstraction for Liveness of Variables

At a program point p

$a \mapsto 1 \Rightarrow a$ is live at p

$a \mapsto 0 \Rightarrow a$ is not live at p



Computing Static Abstraction for Liveness of Variables

At a program point p

$a \mapsto 1 \Rightarrow a$ is live at p

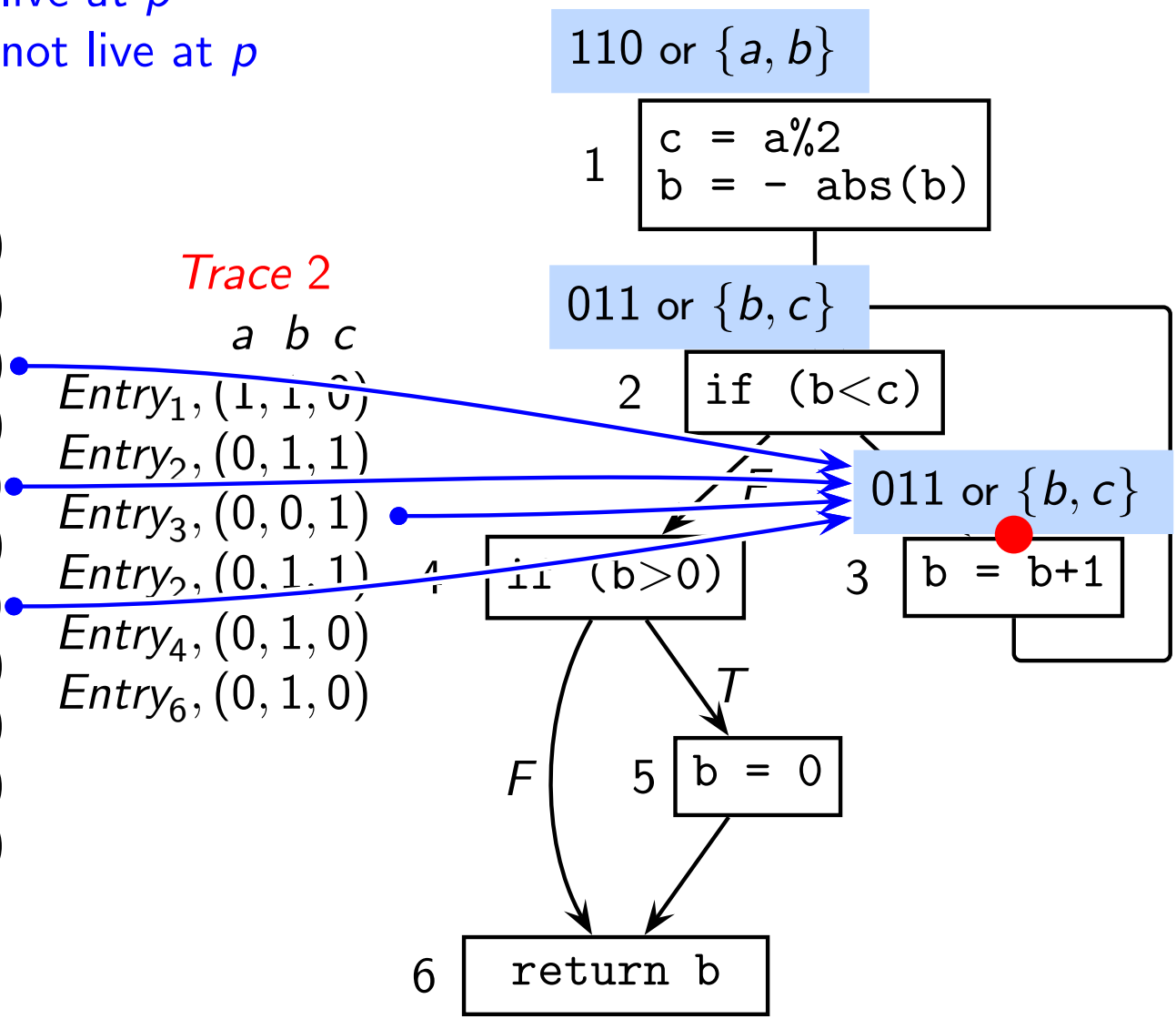
$a \mapsto 0 \Rightarrow a$ is not live at p

Trace 1

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_5$	0	0	0
$Entry_6$	0	1	0

Trace 2

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	0	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_6$	0	1	0



Computing Static Abstraction for Liveness of Variables

At a program point p

$a \mapsto 1 \Rightarrow a$ is live at p

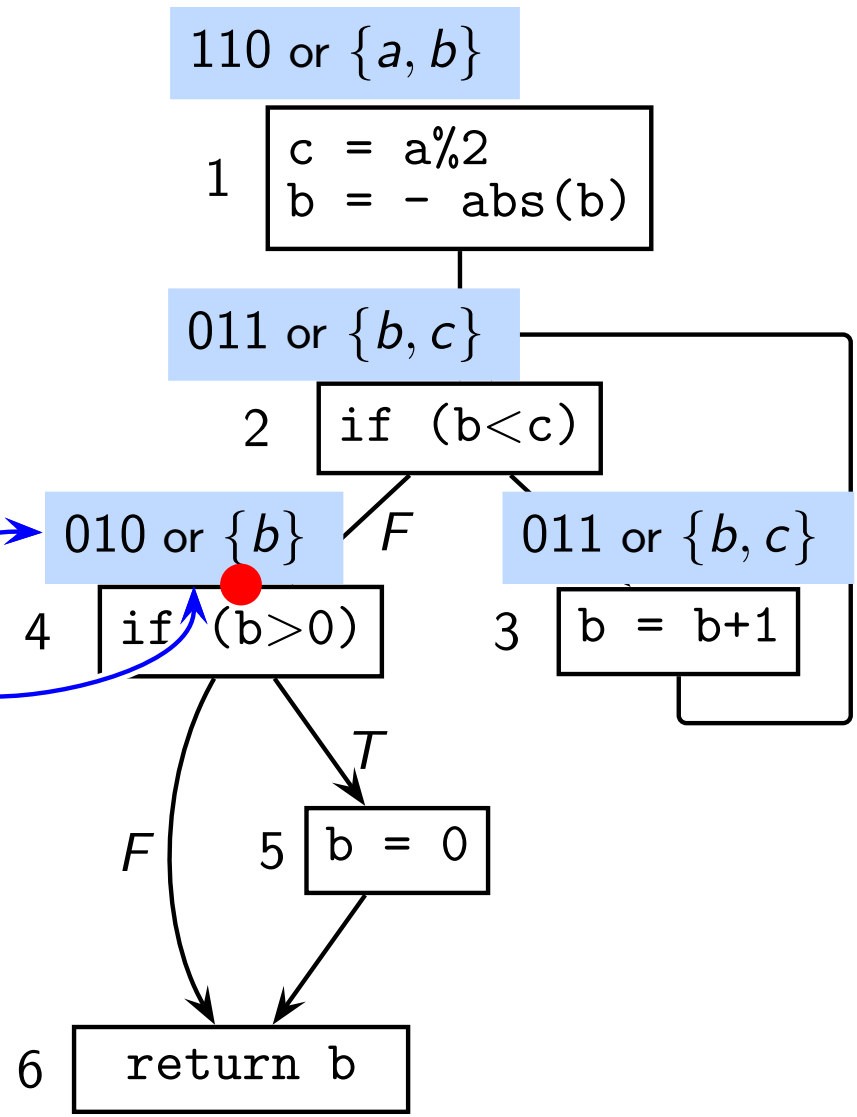
$a \mapsto 0 \Rightarrow a$ is not live at p

Trace 1

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_5$	0	0	0
$Entry_6$	0	1	0

Trace 2

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	0	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_6$	0	1	0



Computing Static Abstraction for Liveness of Variables

At a program point p

$a \mapsto 1 \Rightarrow a$ is live at p

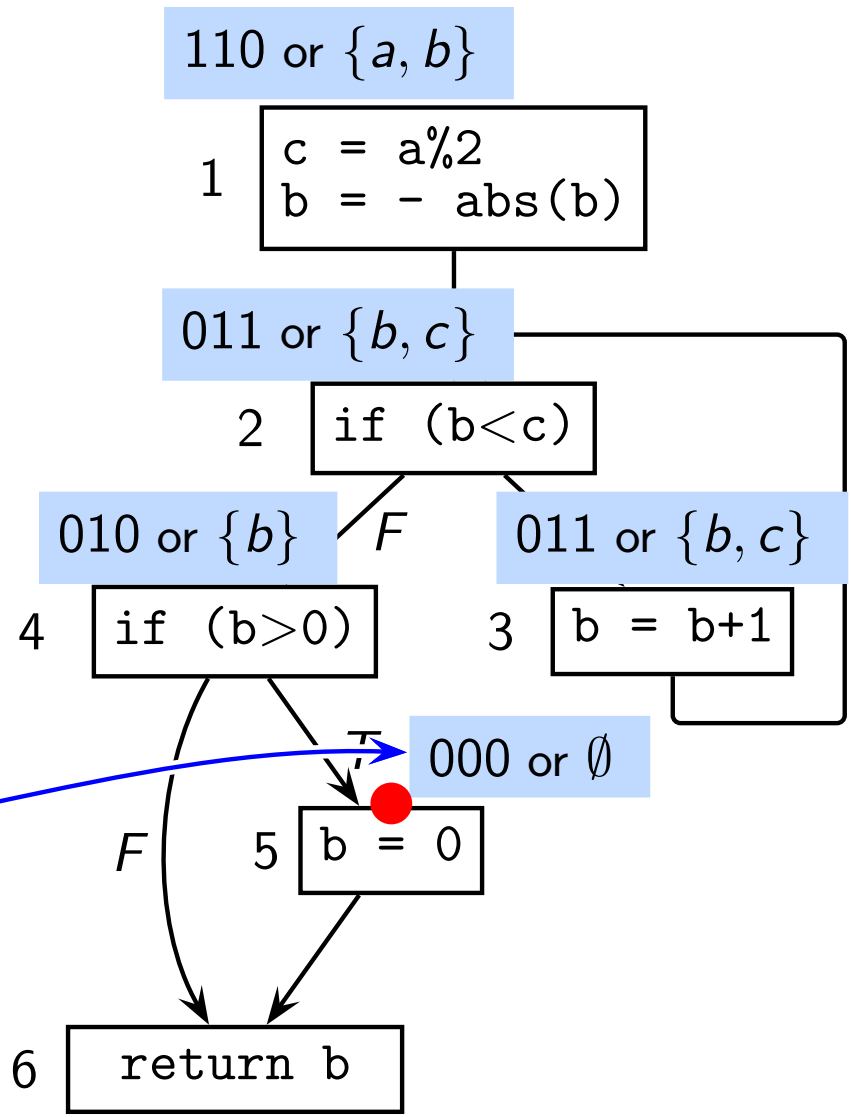
$a \mapsto 0 \Rightarrow a$ is not live at p

Trace 1

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_5$	0	0	0
$Entry_6$	0	1	0

Trace 2

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	0	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_6$	0	1	0



Computing Static Abstraction for Liveness of Variables

At a program point p

$a \mapsto 1 \Rightarrow a$ is live at p

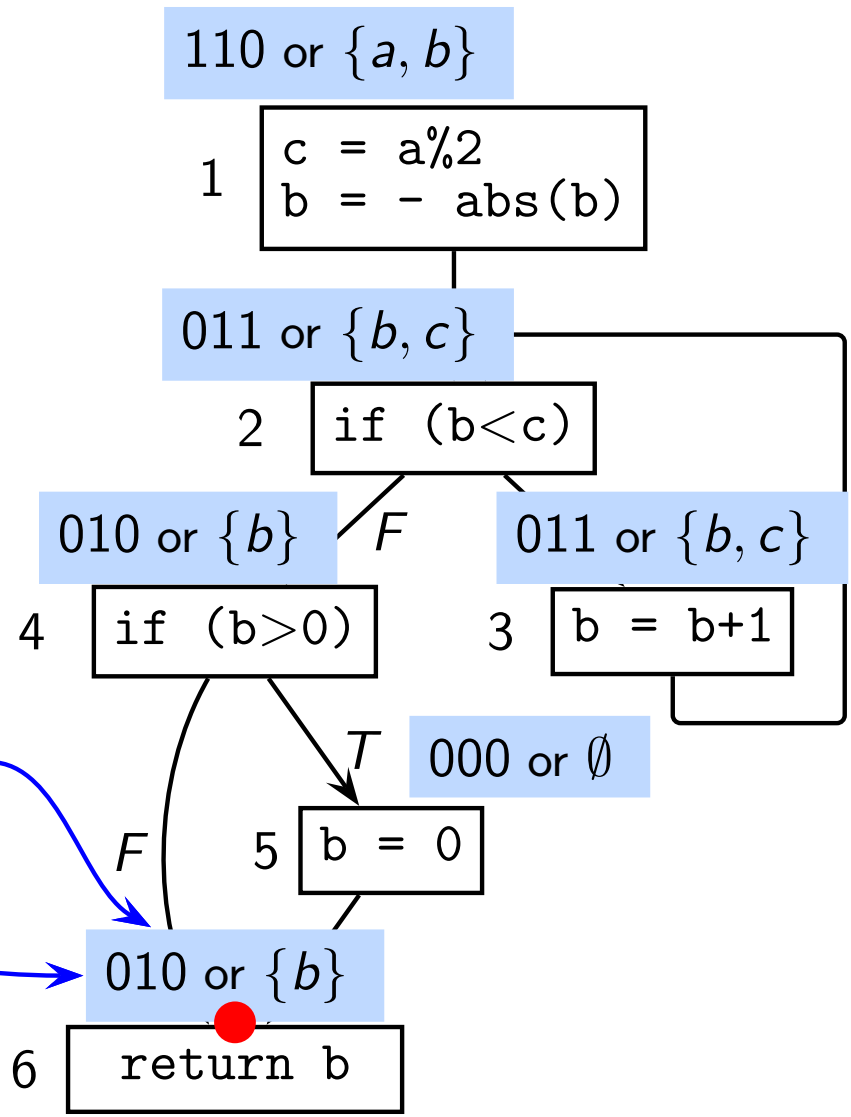
$a \mapsto 0 \Rightarrow a$ is not live at p

Trace 1

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_5$	0	0	0
$Entry_6$	0	1	0

Trace 2

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	0	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_6$	0	1	0



Computing Static Abstraction for Liveness of Variables

At a program point p

$a \mapsto 1 \Rightarrow a$ is live at p

$a \mapsto 0 \Rightarrow a$ is not live at p

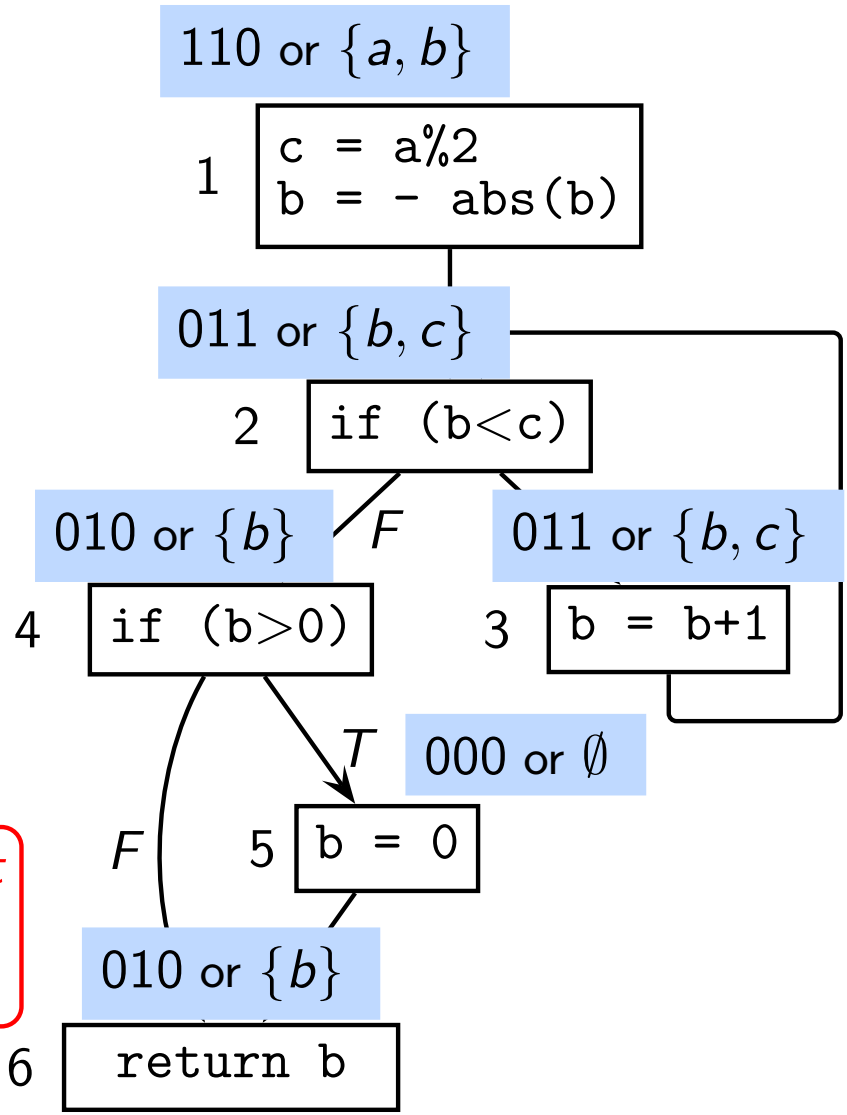
Trace 1

	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_3$	0	1	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_5$	0	0	0
$Entry_6$	0	1	0

Trace 2

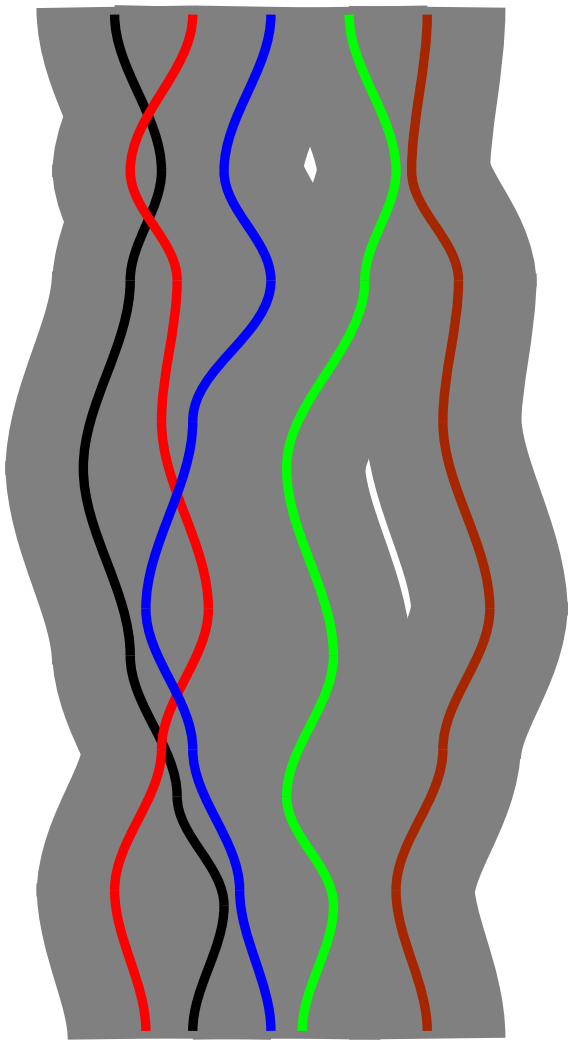
	a	b	c
$Entry_1$	1	1	0
$Entry_2$	0	1	1
$Entry_3$	0	0	1
$Entry_2$	0	1	1
$Entry_4$	0	1	0
$Entry_6$	0	1	0

Trace 2 does not add anything to the abstraction



Soundness of Abstractions (1)

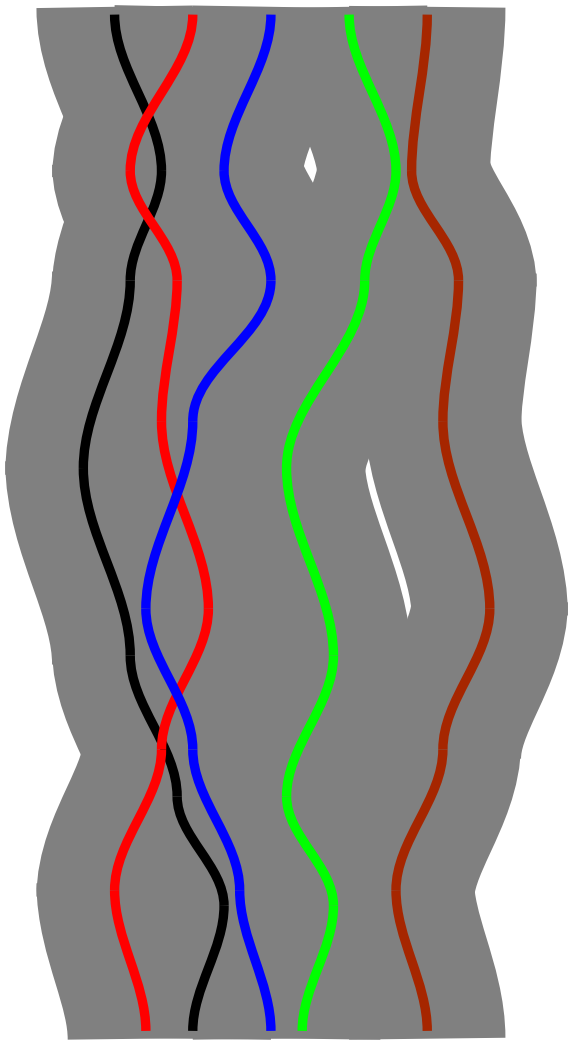
Sound



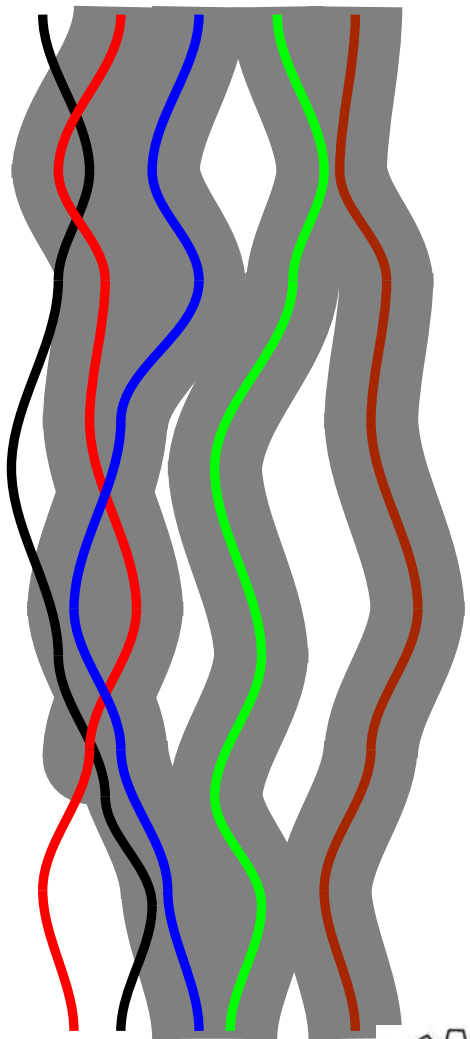
- An over-approximation of traces is sound

Soundness of Abstractions (1)

Sound

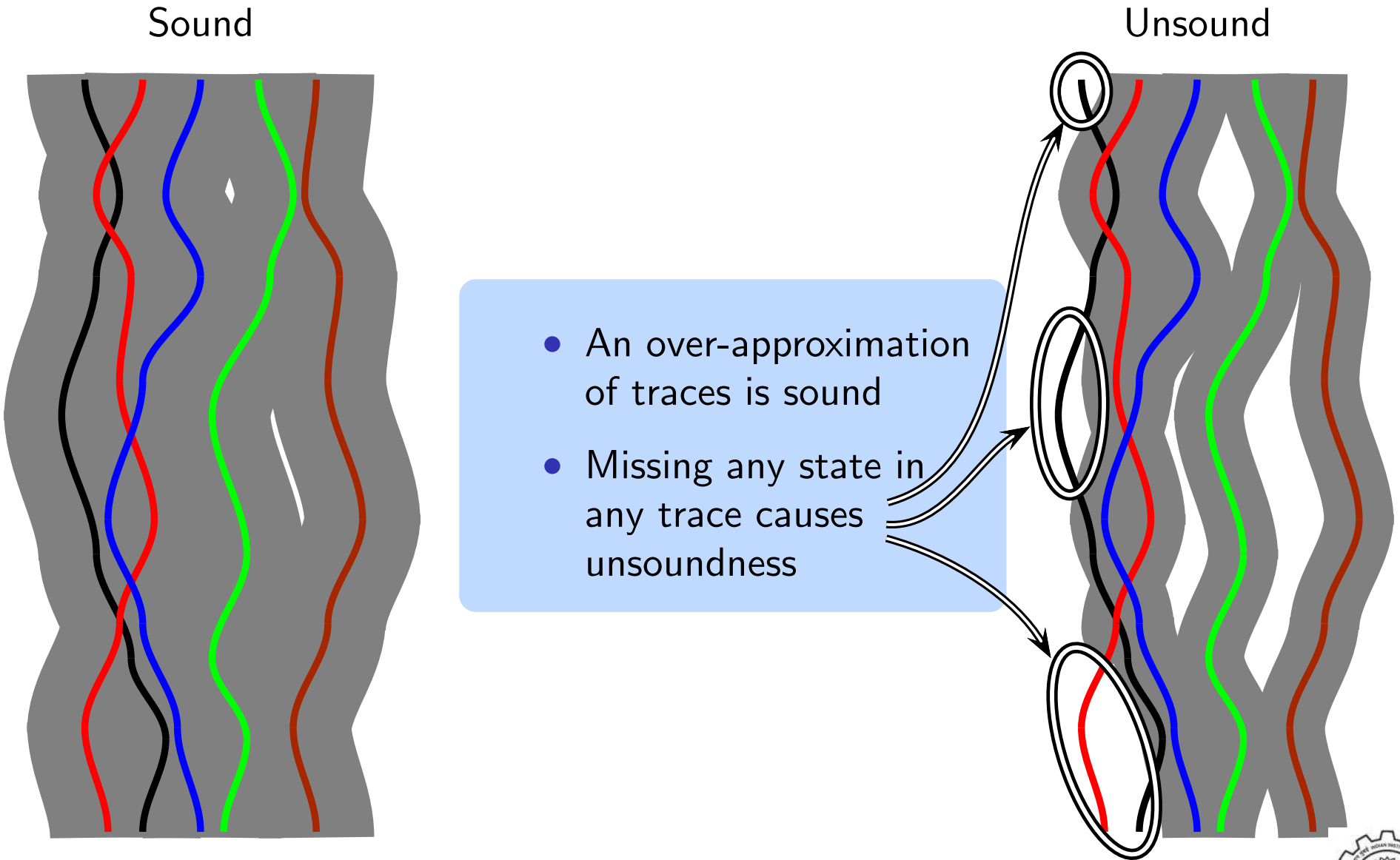


Unsound



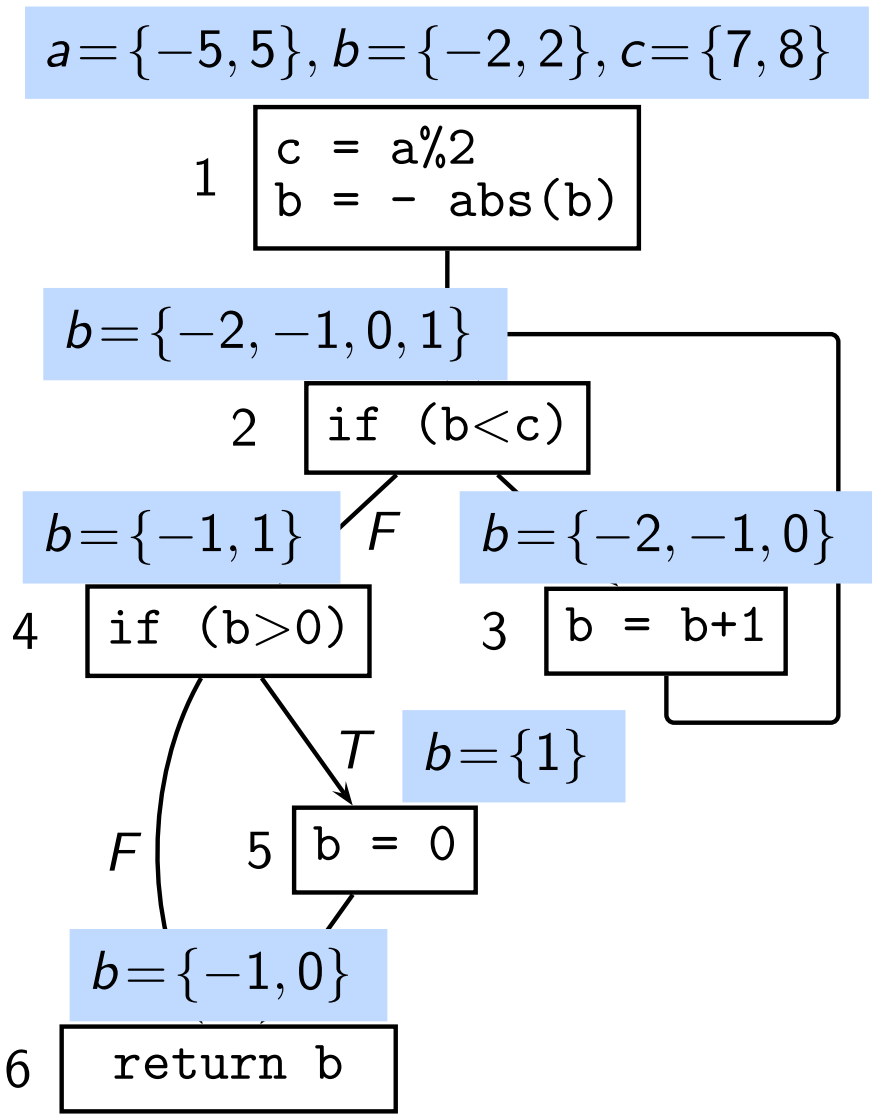
- An over-approximation of traces is sound
- Missing any state in any trace causes unsoundness

Soundness of Abstractions (1)



Soundness of Abstractions (2)

An unsound abstraction



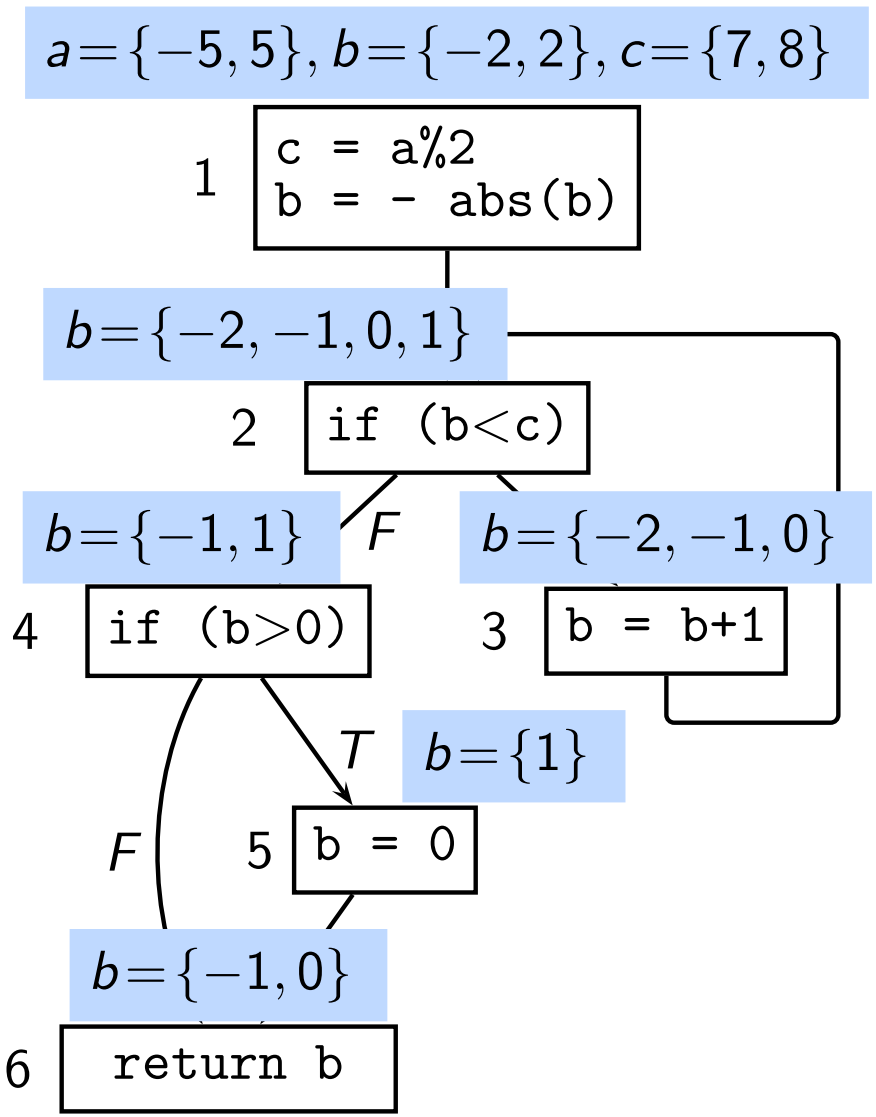
All variables can have arbitrary values at the start.

b can have many more values at the entry of

- blocks 2 and 3 (e.g. -3, -8, . . .)
- block 4 (e.g. 0)

Soundness of Abstractions (2)

An unsound abstraction



A sound abstraction using intervals

- Over-approximated range of values denoted by

$$\left[low\_limit, high\_limit \right]$$

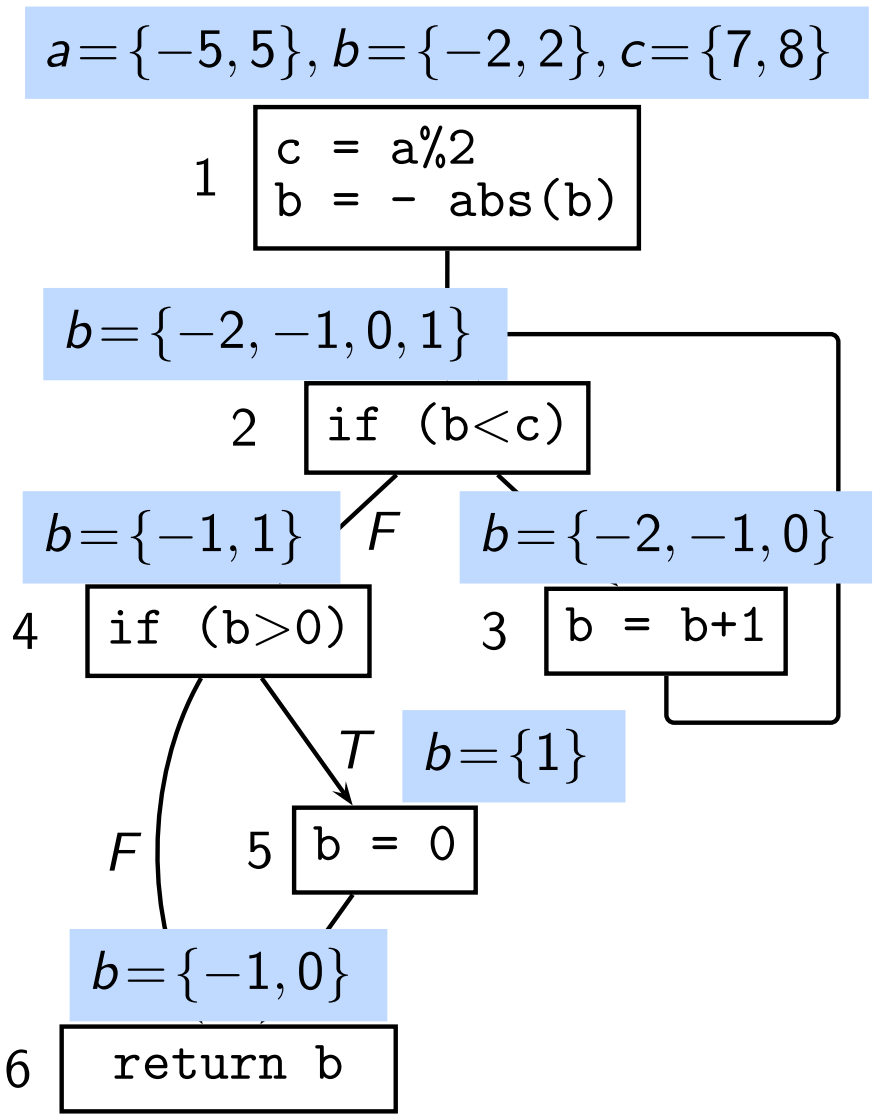
- Inclusive limits with

$$low\_limit \leq high\_limit$$

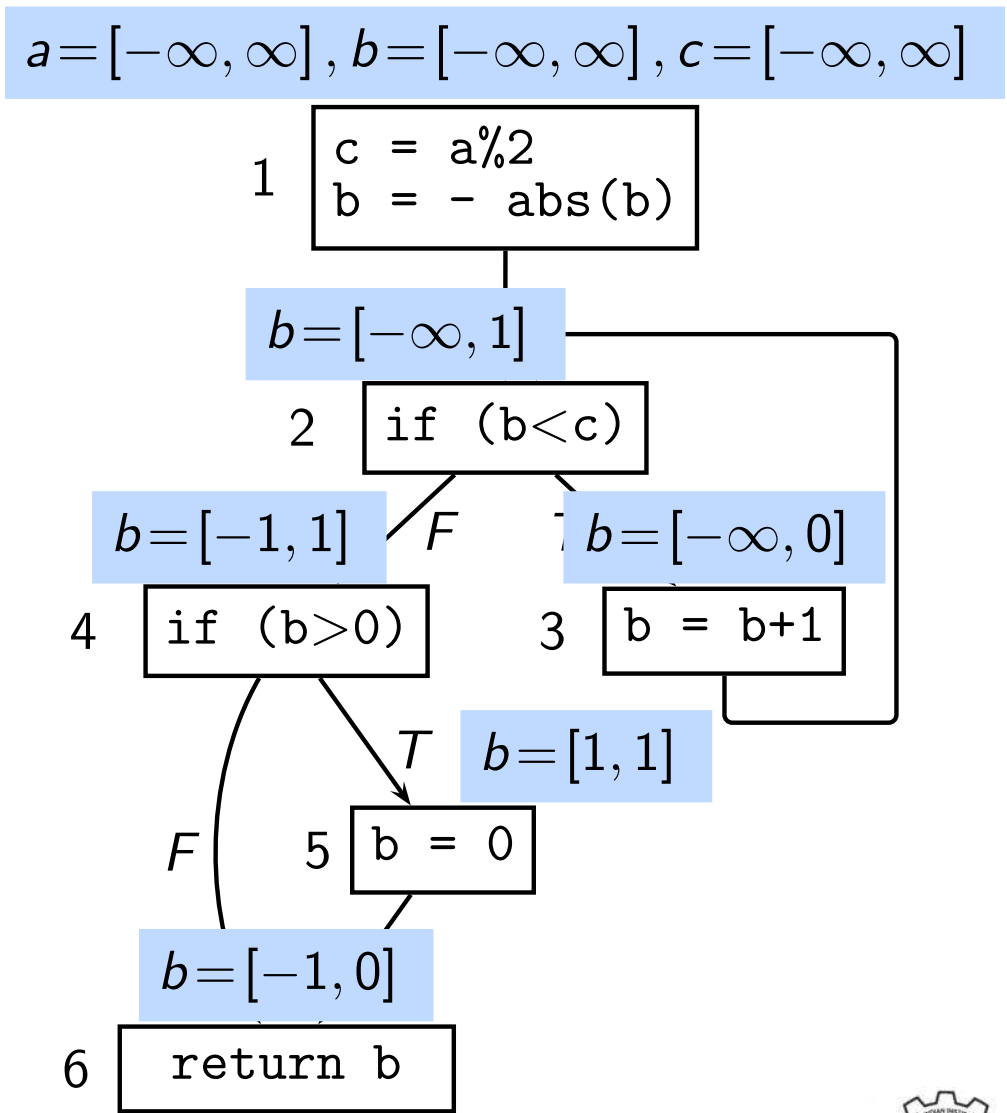
- One contiguous range per variable with no "holes"

Soundness of Abstractions (2)

An unsound abstraction

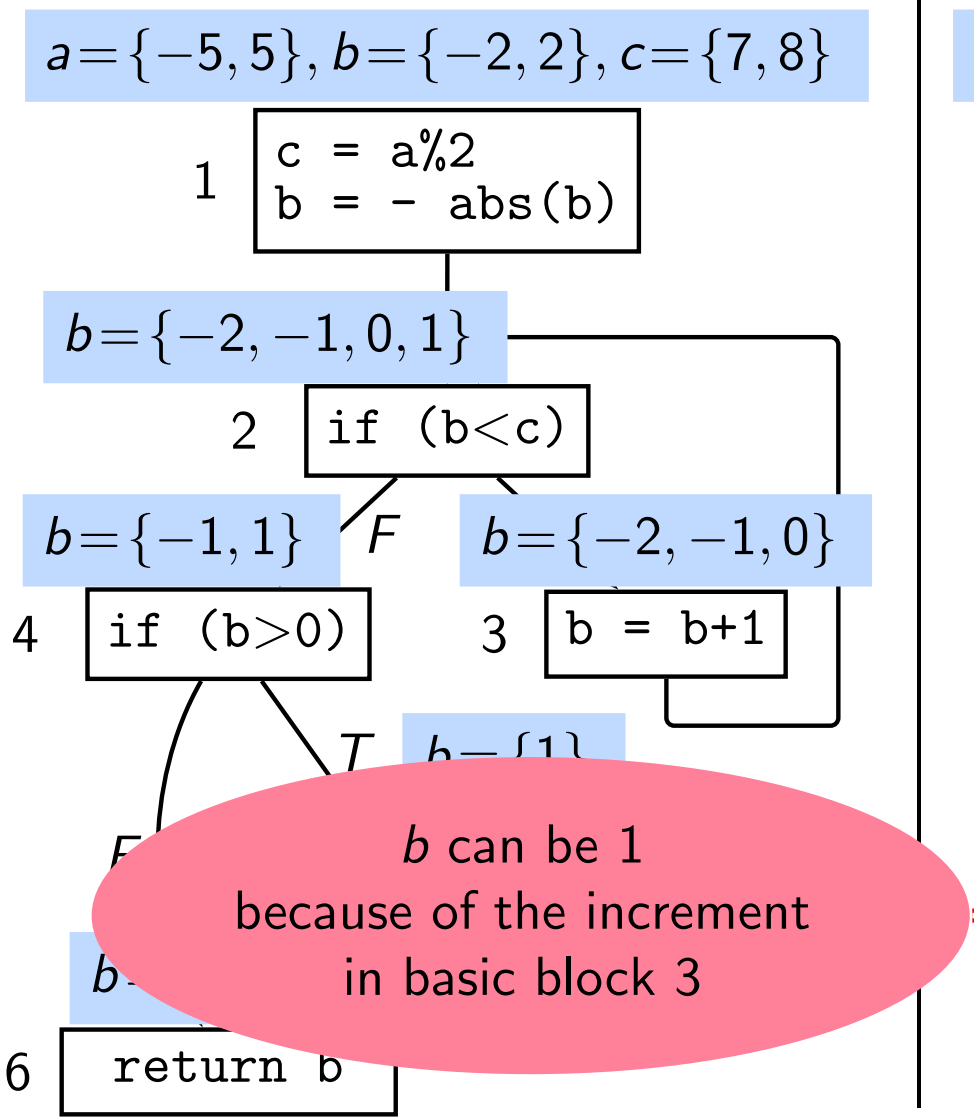


A sound abstraction using intervals

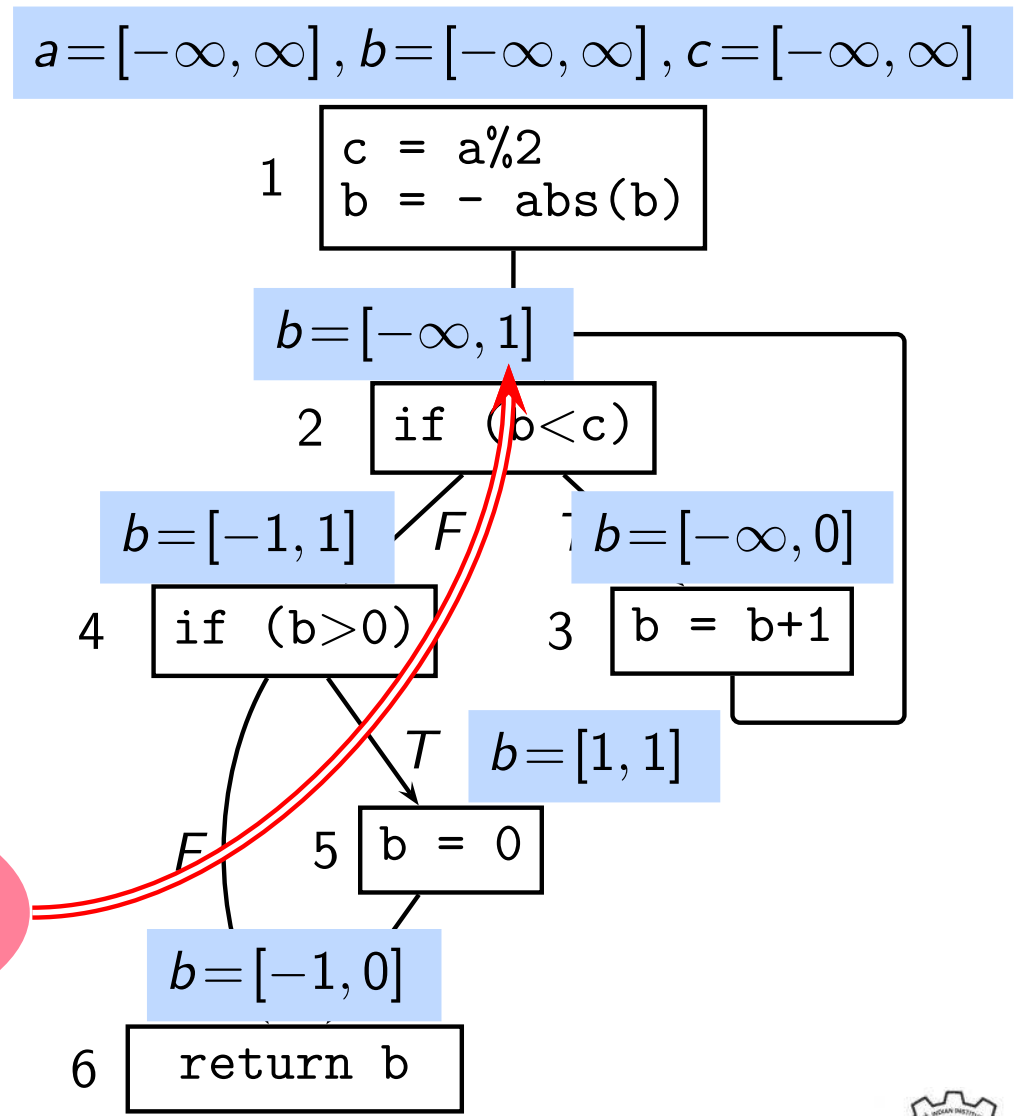


Soundness of Abstractions (2)

An unsound abstraction



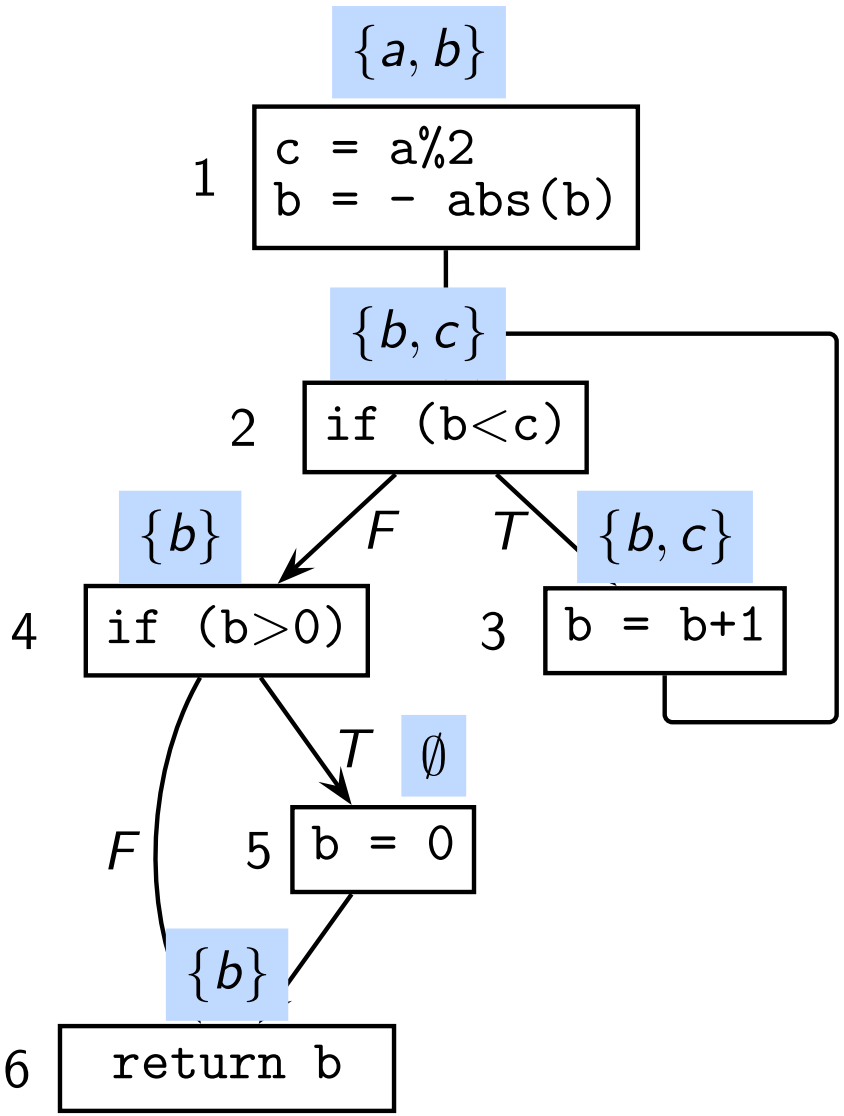
A sound abstraction using intervals



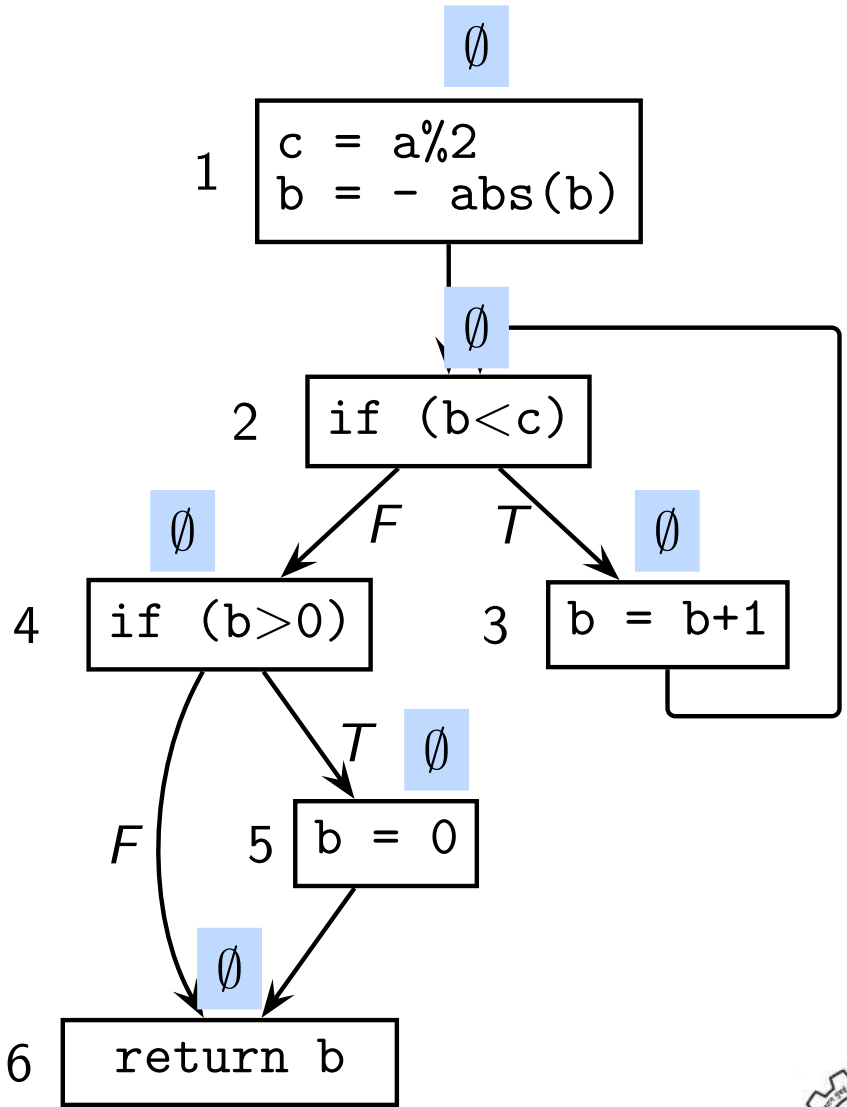
b can be 1
because of the increment
in basic block 3

Soundness of Abstractions for Liveness Analysis

A sound abstraction

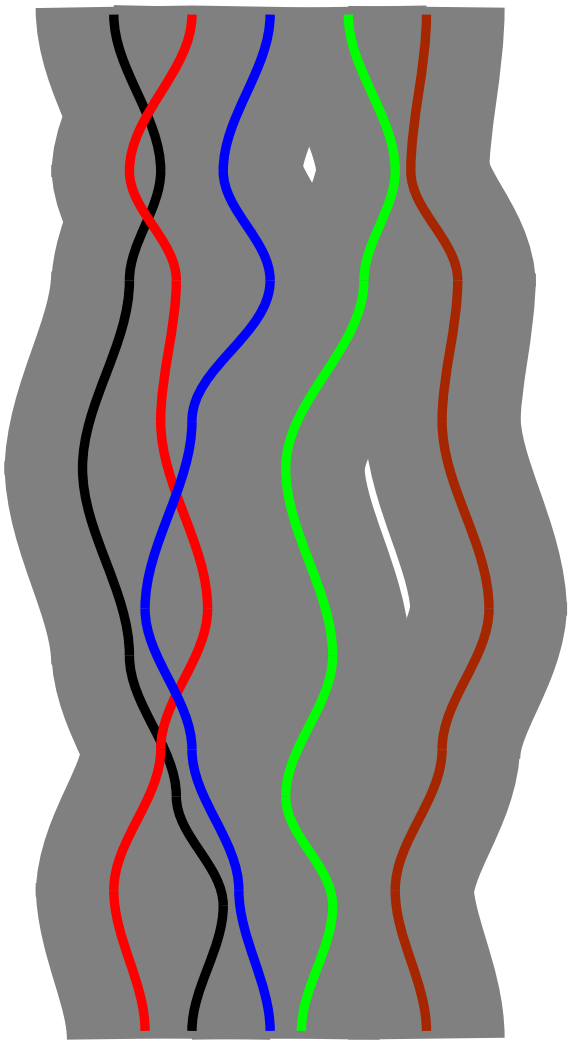


An unsound abstraction



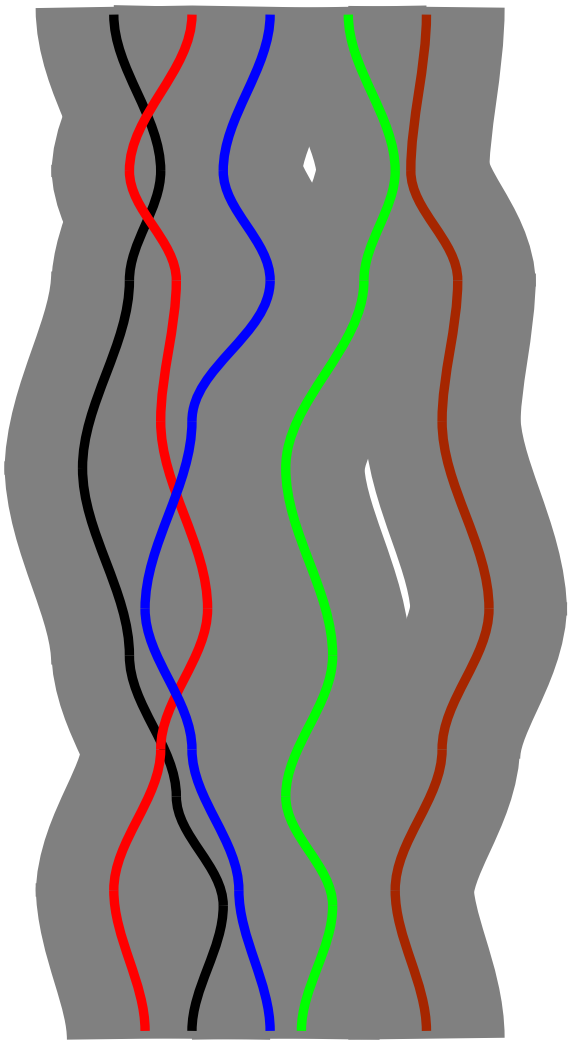
Precision of Sound Abstractions(1)

Sound but imprecise

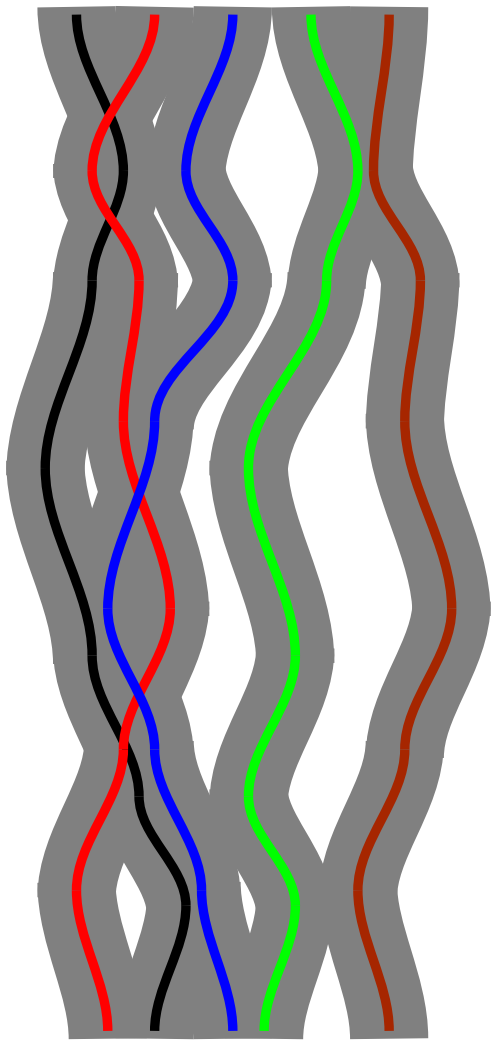


Precision of Sound Abstractions(1)

Sound but imprecise

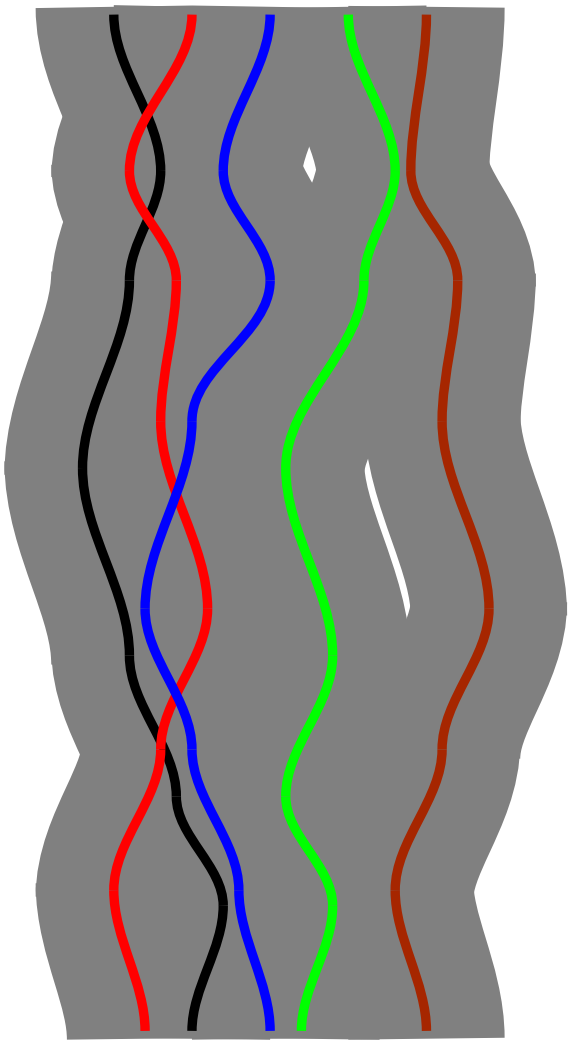


Sound and more precise

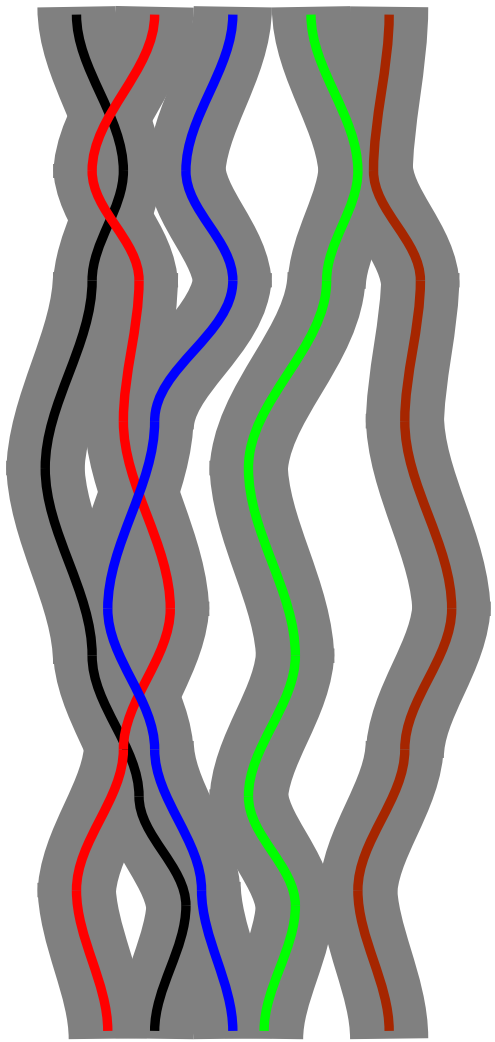


Precision of Sound Abstractions(1)

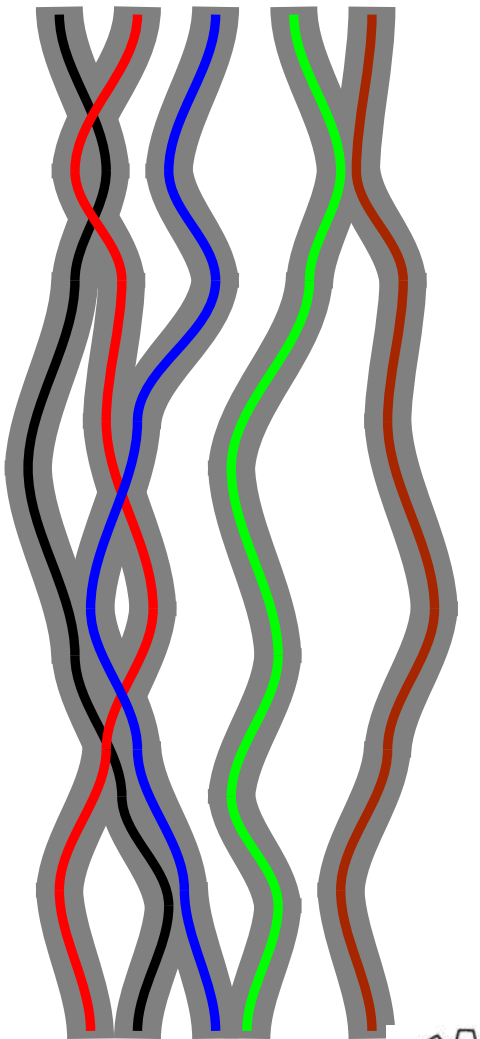
Sound but imprecise



Sound and more precise



Sound and even more precise

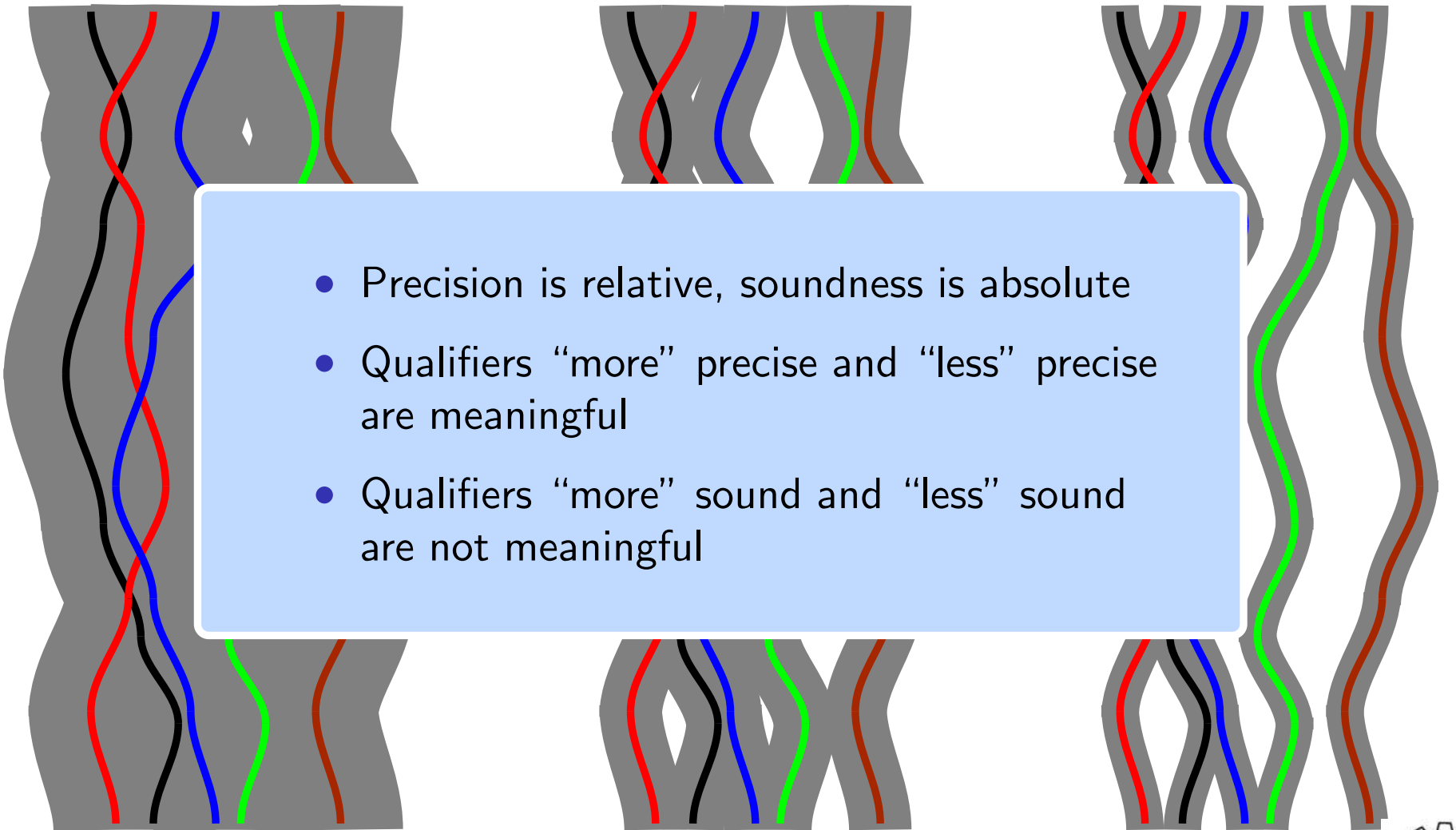


Precision of Sound Abstractions(1)

Sound but imprecise

Sound and more precise

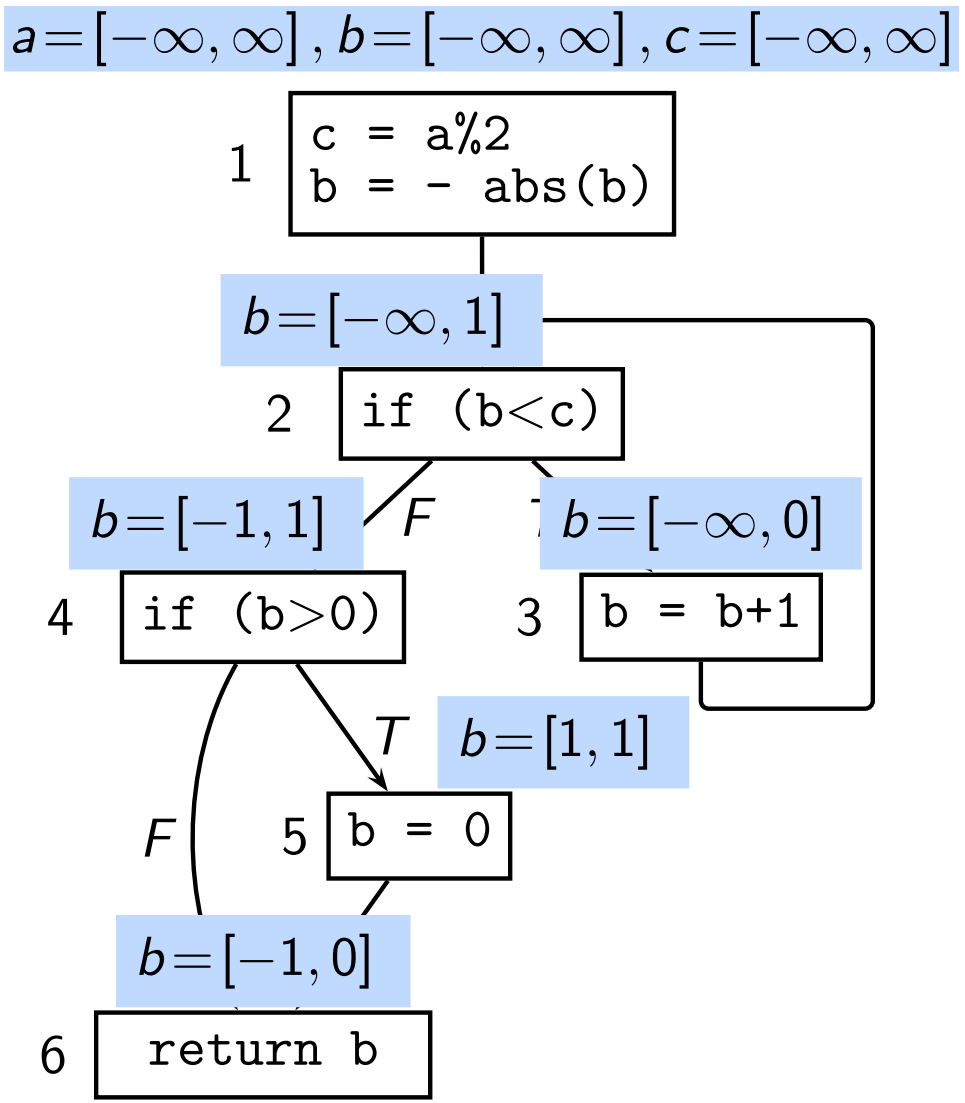
Sound and even more precise



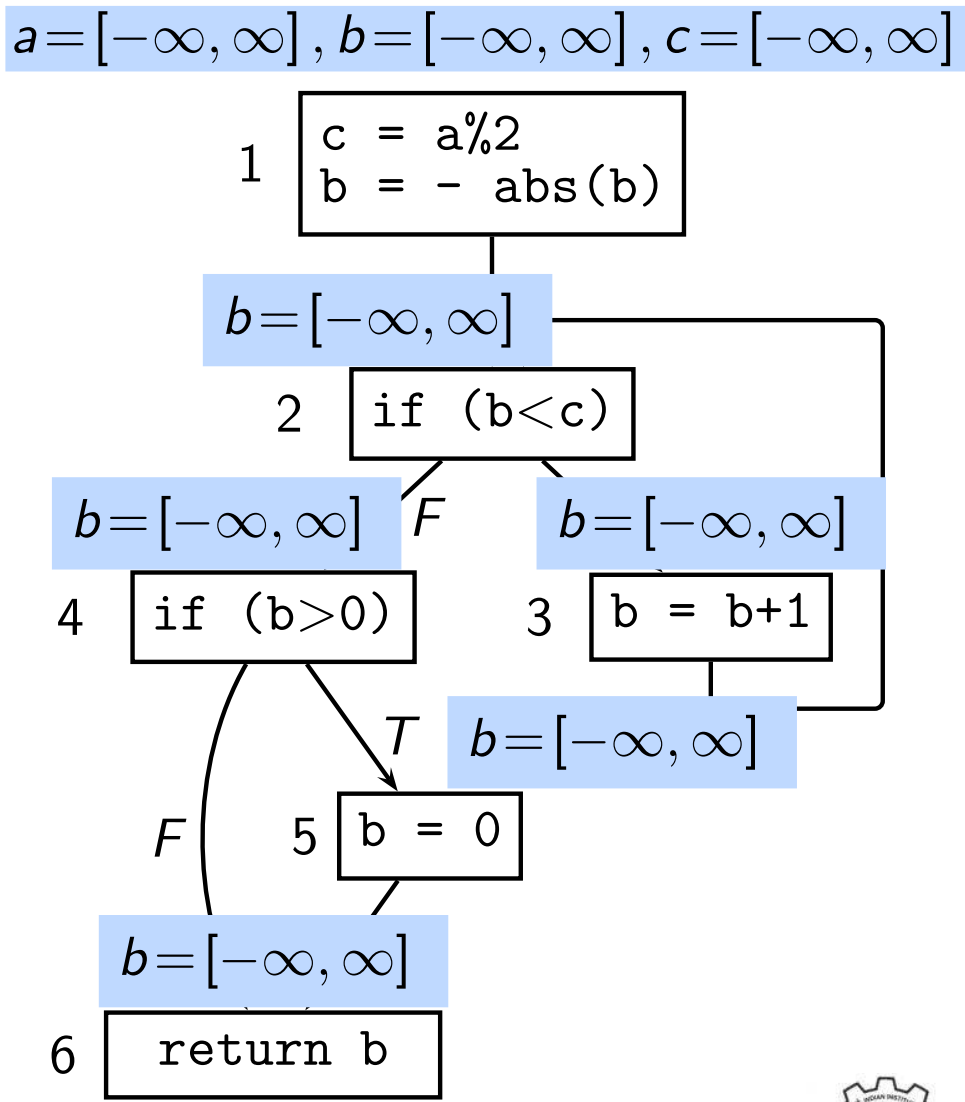
- Precision is relative, soundness is absolute
- Qualifiers "more" precise and "less" precise are meaningful
- Qualifiers "more" sound and "less" sound are not meaningful

Precision of Sound Abstractions(2)

A precise abstraction using intervals

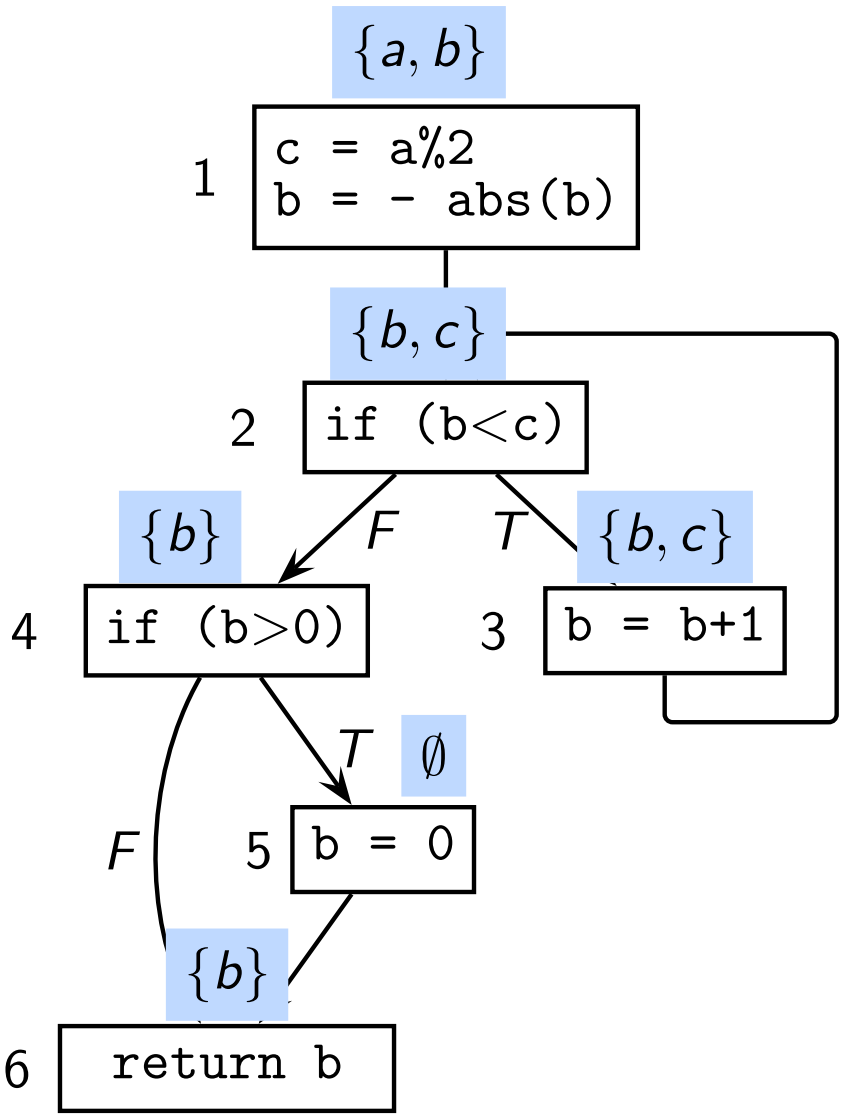


An imprecise abstraction using intervals

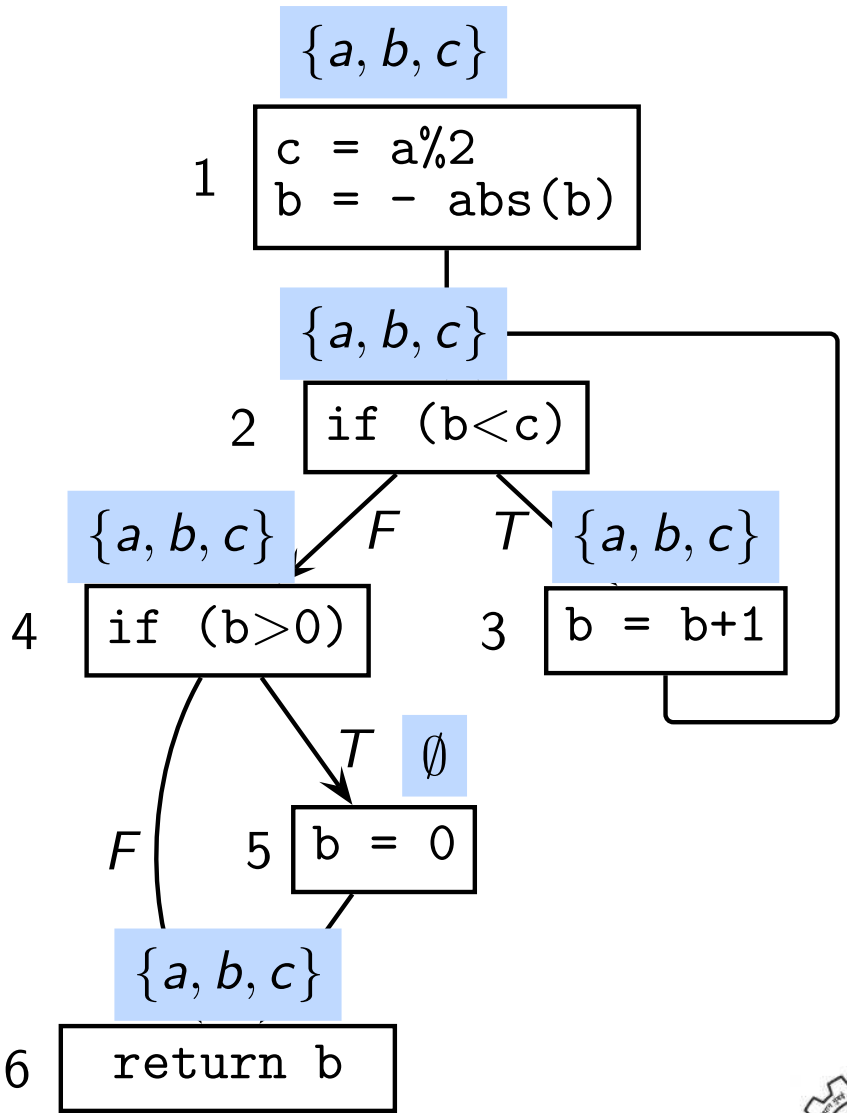


Precision of Abstractions for Liveness Analysis

A precise abstraction



An imprecise abstraction



Limitations of Static Analysis

- In general, the computation of *exact* static abstraction is *undecidable*
  - Possible reasons
    - Values of variables not known
    - Branch outcomes not known
    - Infinitely many paths in the presence of loops or recursion
    - Infinitely many values
  - We have to settle for some imprecision
  - How are data states compared to distinguish between a sound and unsound (or a precise or an imprecise result)?
    - We have introduced the concepts intuitively
    - Formally, the comparison is made by defining a partial order

Practical Static Analysis

- The goodness of a static analysis lies in minimizing imprecision without compromising on soundness

  Additional expectations: Efficiency and scalability

- Some applications (e.g. debugging) do not need to cover all traces

- We have not talked about completeness

  - Some features of a programming language may not be covered

    (e.g. "eval" in JavaScript, aliasing of array indices, effect of libraries)

  - Accept a "soundy" analysis [Livshits et. al. CACM 2015]

    OR

    Tolerate imprecision for complete soundness