

Introduction to Program Analysis

Uday Khedker

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

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay



July 2017

Part 1

About These Slides

Copyright

These slides constitute the lecture notes for CS618 Program Analysis course at IIT Bombay and have been made available as teaching material accompanying the book:

- Uday Khedker, Amitabha Sanyal, and Bageshri Karkare.
Data Flow Analysis: Theory and Practice. CRC Press (Taylor and Francis Group). 2009.
(Indian edition published by Ane Books in 2013)

Apart from the above book, some slides are based on the material from the following books

- A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. 2006.
- M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland Inc. 1977.

These slides are being made available under GNU FDL v1.2 or later purely for academic or research use.



Motivating the Need of Program Analysis

- Some representative examples
 - ▶ Classical optimizations performed by compilers
 - ▶ Optimizing heap memory usage
- Course details, schedule, assessment policies etc.
- Program Model
- Soundness and Precision



Part 2

Classical Optimizations

Examples of Optimising Transformations (ALSU, 2006)

A C program and its optimizations

```

void quicksort(int m, int n)
{
  int i, j, v, x;
  if (n <= m) return;
  i = m-1; j = n; v = a[n];           /* v is the pivot */
  while(1)                             /* Move values smaller */
  {
    do i = i + 1; while (a[i] < v);     /* than v to the left of */
    do j = j - 1; while (a[j] > v);     /* the split point (sp) */
    if (i >= j) break;                 /* and other values */
    x = a[i]; a[i] = a[j]; a[j] = x;    /* to the right of sp */
  }                                     /* of the split point */
  x = a[i]; a[i] = a[n]; a[n] = x;     /* Move the pivot to sp */
  quicksort(m,i); quicksort(i+1,n);    /* sort the partitions to */
}                                       /* the left of sp and to the right of sp independently */

```



Intermediate Code

For the boxed source code

1. $i = m - 1$
2. $j = n$
3. $t1 = 4 * n$
4. $t6 = a[t1]$
5. $v = t6$
6. $i = i + 1$
7. $t2 = 4 * i$
8. $t3 = a[t2]$
9. if $t3 < v$ goto 6
10. $j = j - 1$
11. $t4 = 4 * j$
12. $t5 = a[t4]$
13. if $t5 > v$ goto 10
14. if $i \geq j$ goto 25
15. $t2 = 4 * i$
16. $t3 = a[t2]$
17. $x = t3$
18. $t2 = 4 * i$
19. $t4 = 4 * j$
20. $t5 = a[t4]$
21. $a[t2] = t5$
22. $t4 = 4 * j$
23. $a[t4] = x$
24. goto 6
25. $t2 = 4 * i$
26. $t3 = a[t2]$
27. $x = t3$
28. $t2 = 4 * i$
29. $t1 = 4 * n$
30. $t6 = a[t1]$
31. $a[t2] = t6$
32. $t1 = 4 * n$
33. $a[t1] = x$



Intermediate Code : Observations

- Multiple computations of expressions
- Simple control flow (conditional/unconditional goto)
Yet undecipherable!
- Array address calculations



Understanding Control Flow

- Identify maximal sequences of linear control flow
⇒ Basic Blocks
- No transfer into or out of basic blocks except the first and last statements
Control transfer into the block : only at the first statement.
Control transfer out of the block : only at the last statement.



Intermediate Code with Basic Blocks

```
1. i = m - 1
2. j = n
3. t1 = 4 * n
4. t6 = a[t1]
5. v = t6
```

```
6. i = i + 1
7. t2 = 4 * i
8. t3 = a[t2]
9. if t3 < v goto 6
```

```
10. j = j - 1
11. t4 = 4 * j
```

```
12. t5 = a[t4]
13. if t5 > v goto 10
```

```
14. if i >= j goto 25
```

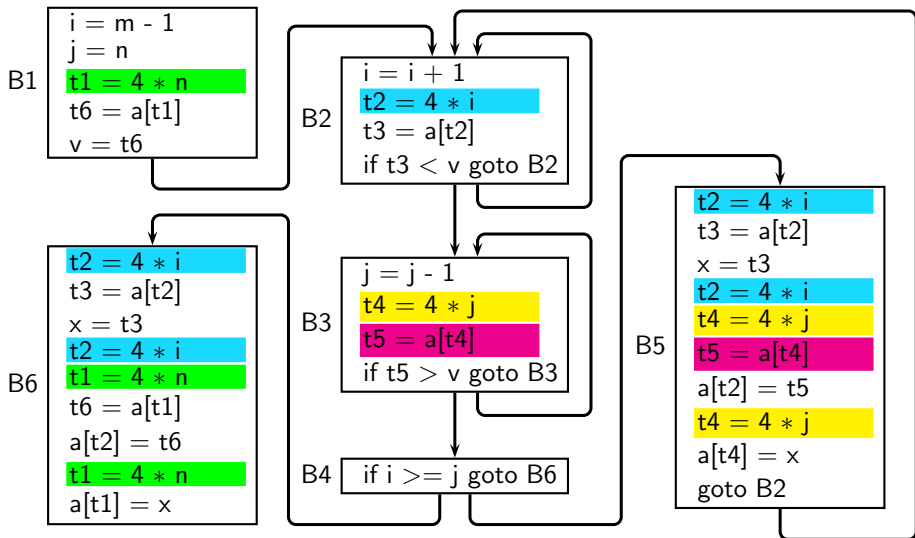
```
15. t2 = 4 * i
16. t3 = a[t2]
17. x = t3
18. t2 = 4 * i
19. t4 = 4 * j
20. t5 = a[t4]
21. a[t2] = t5
22. t4 = 4 * j
```

```
23. a[t4] = x
24. goto 6
```

```
25. t2 = 4 * i
26. t3 = a[t2]
27. x = t3
28. t2 = 4 * i
29. t1 = 4 * n
30. t6 = a[t1]
31. a[t2] = t6
32. t1 = 4 * n
33. a[t1] = x
```



Program Flow Graph

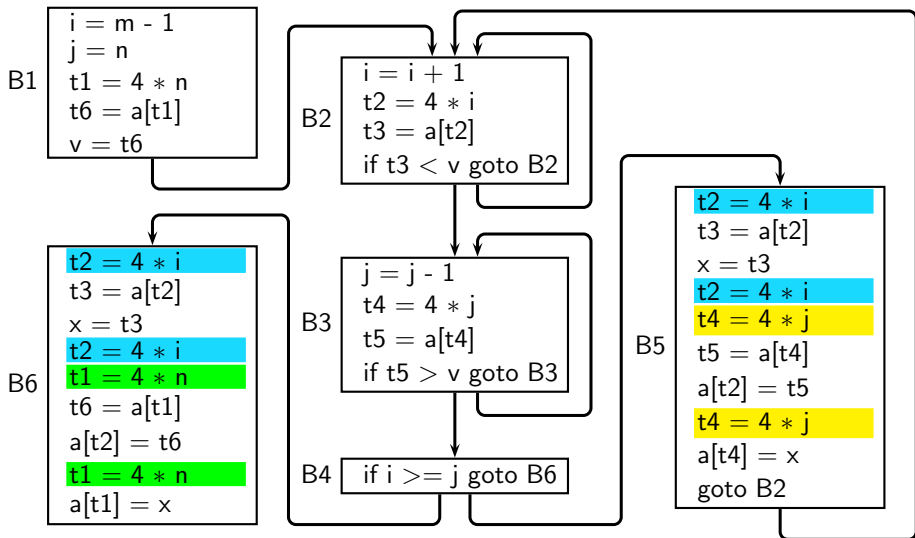


Program Flow Graph : Observations

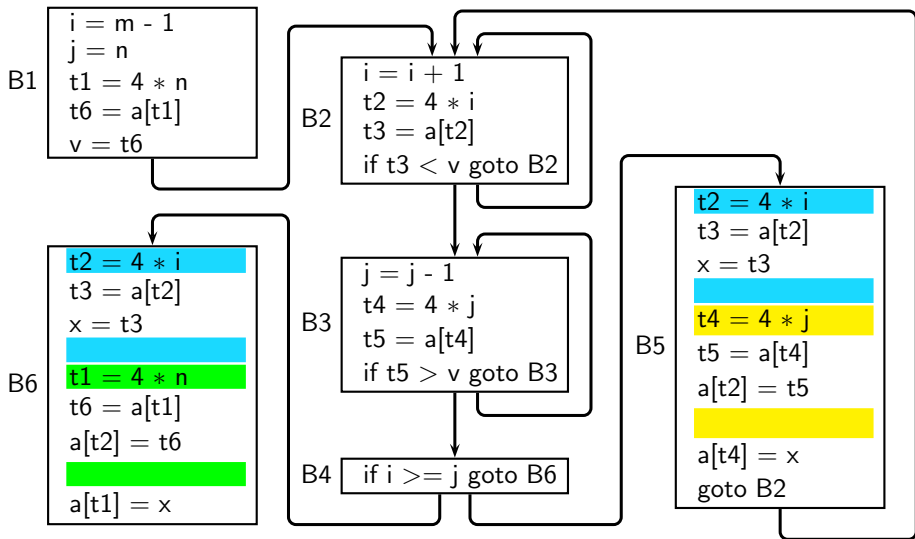
Nesting Level	Basic Blocks	No. of Statements
0	B1, B6	14
1	B4, B5	11
2	B2, B3	8



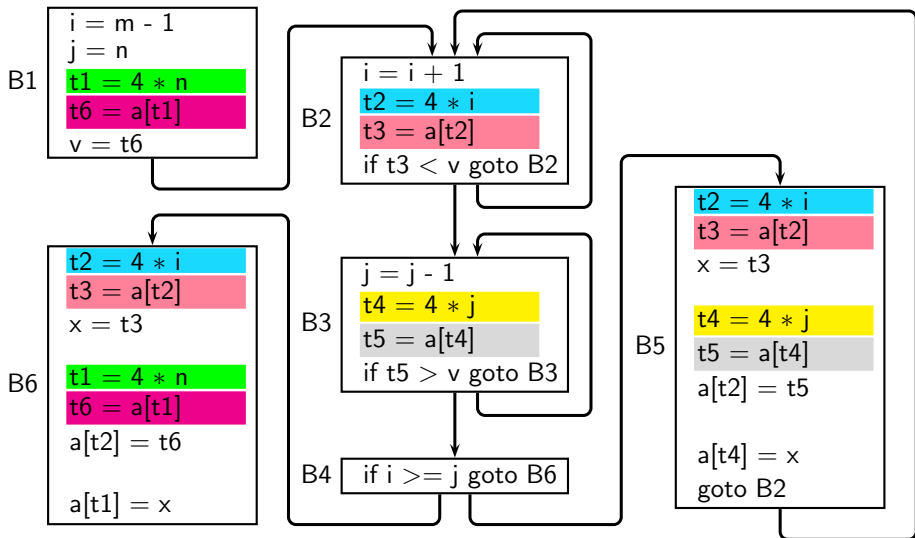
Local Common Subexpression Elimination



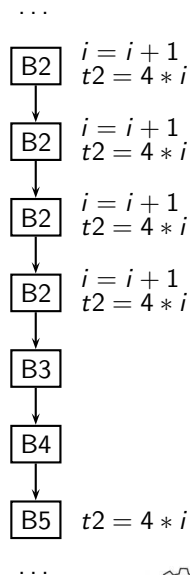
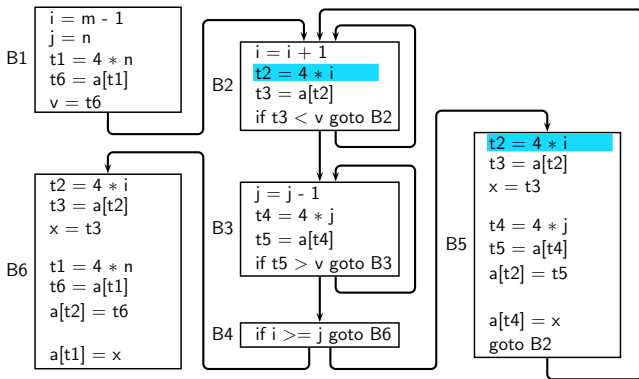
Local Common Subexpression Elimination



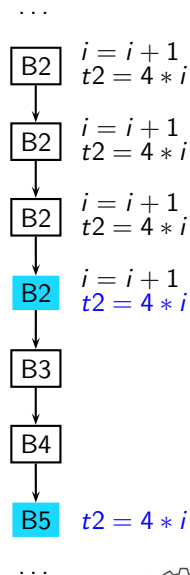
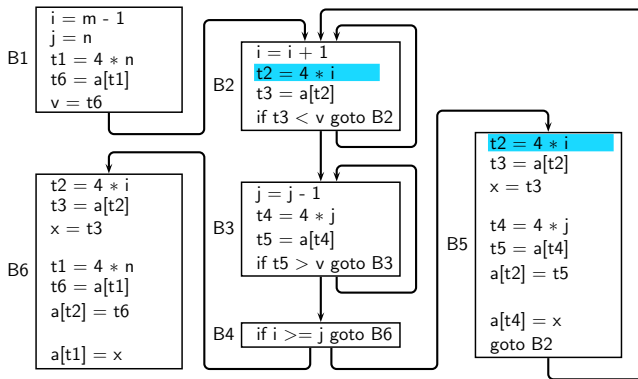
Global Common Subexpression Elimination



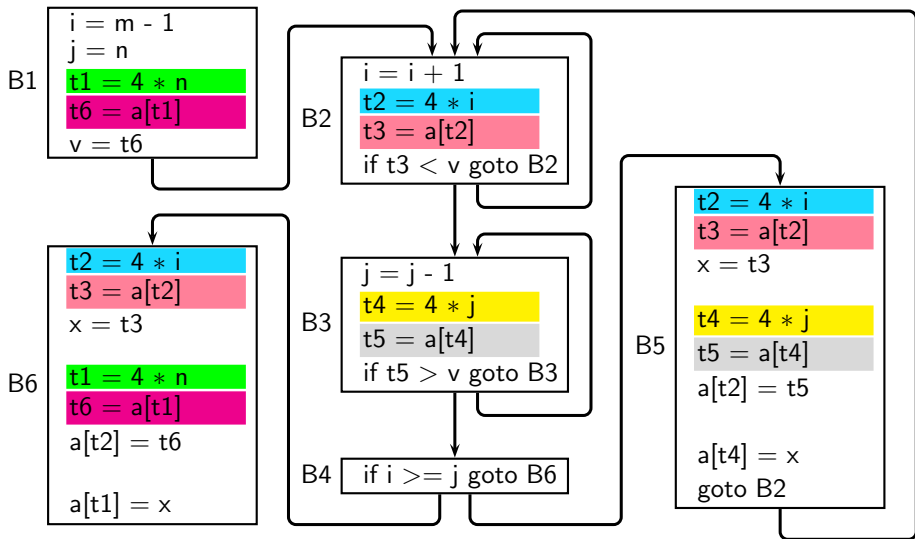
Global Common Subexpression Elimination



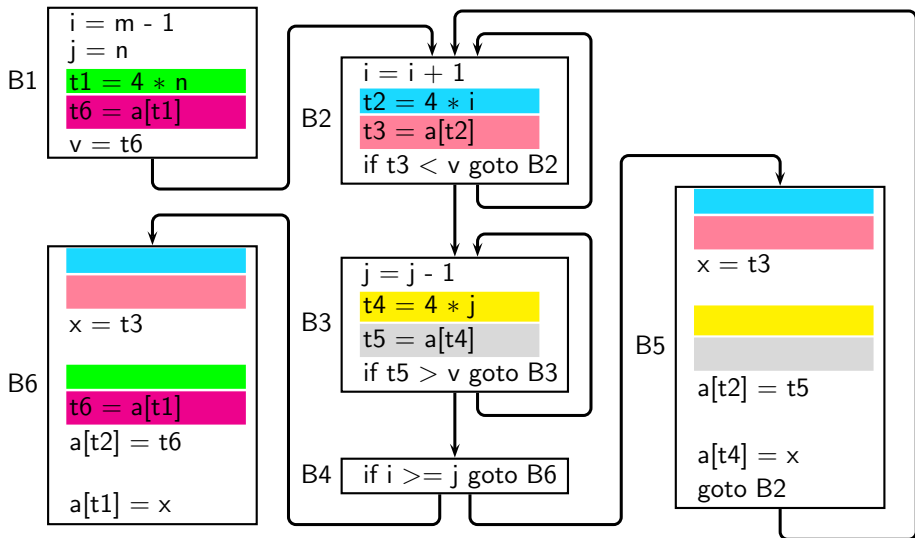
Global Common Subexpression Elimination



Global Common Subexpression Elimination



Global Common Subexpression Elimination

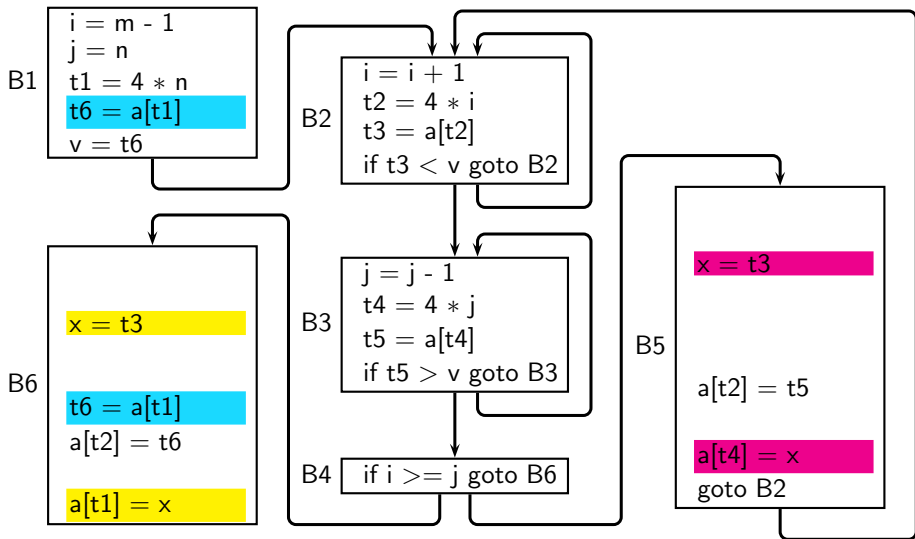


Other Classical Optimizations

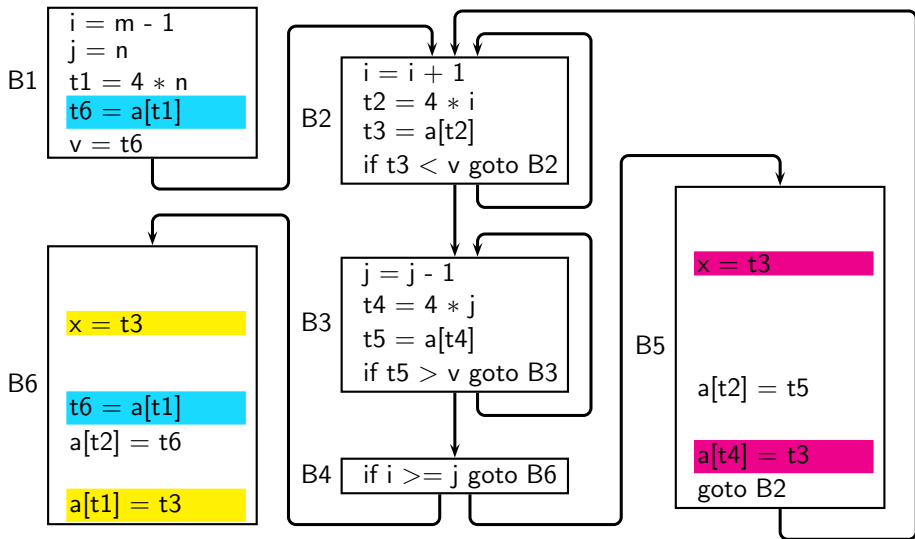
- Copy propagation
- Strength Reduction
- Elimination of Induction Variables
- Dead Code Elimination



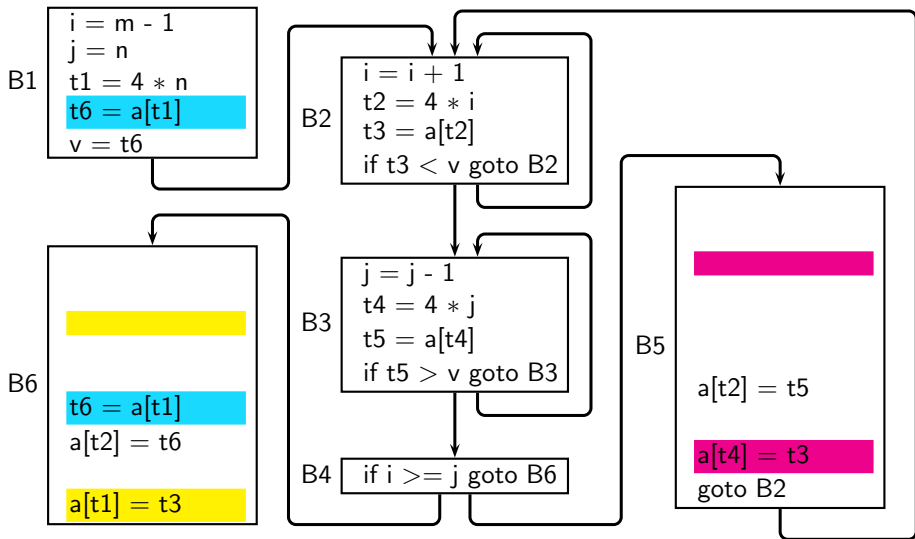
Copy Propagation and Dead Code Elimination



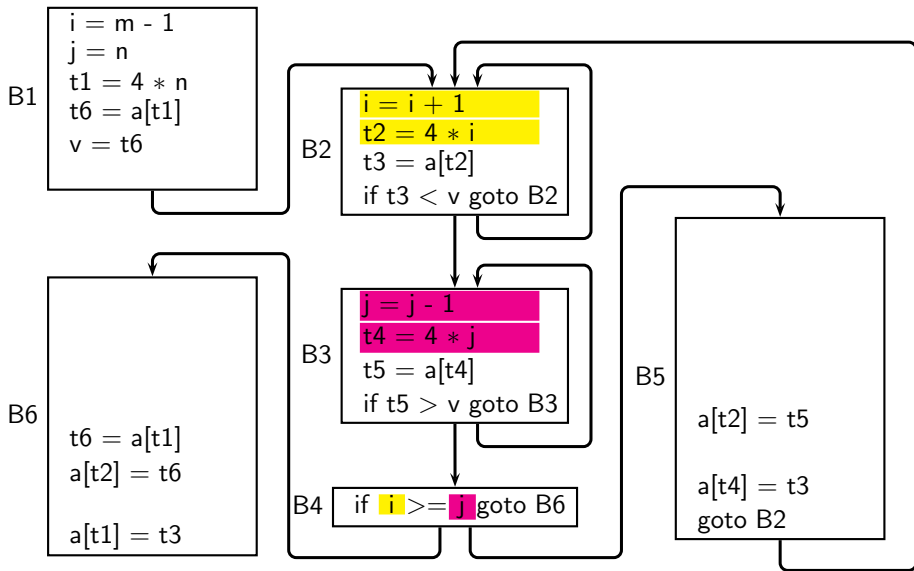
Copy Propagation and Dead Code Elimination



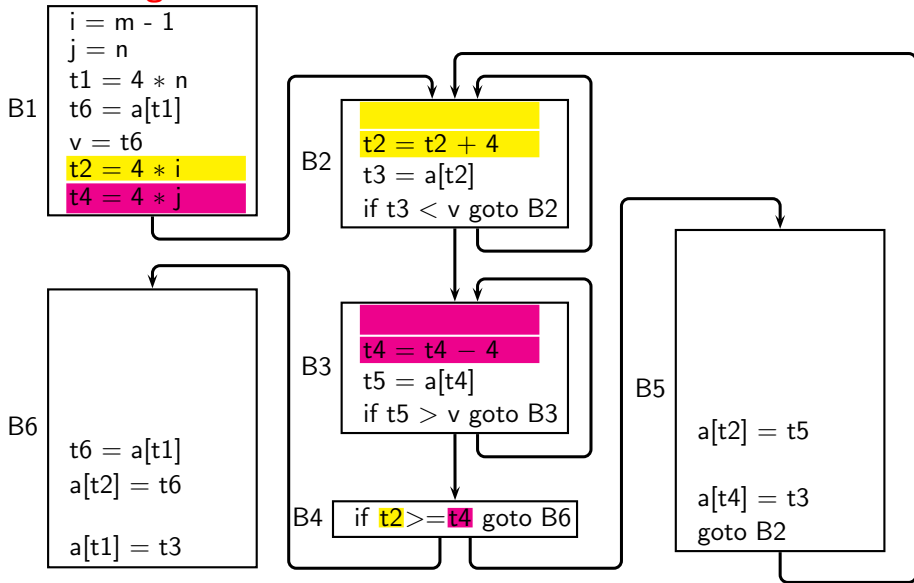
Copy Propagation and Dead Code Elimination



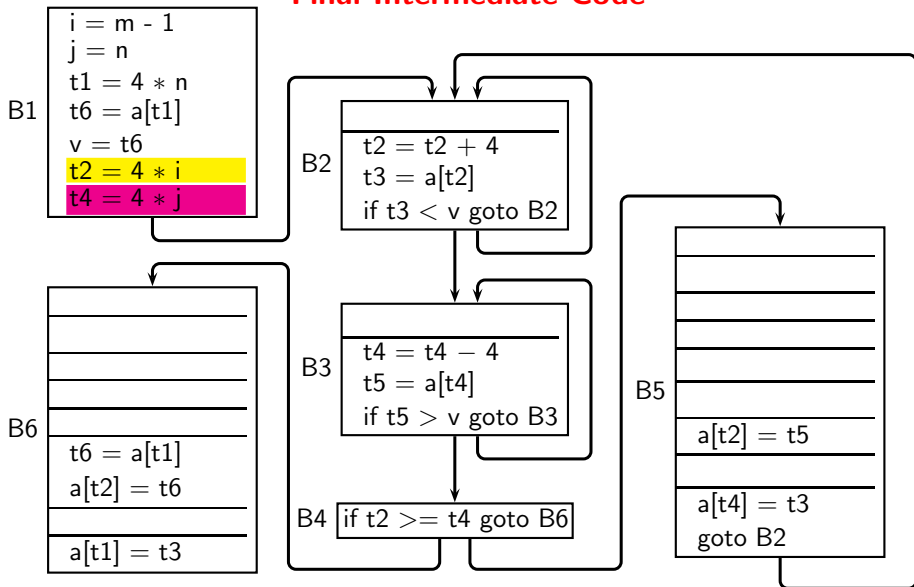
Strength Reduction and Induction Variable Elimination



Strength Reduction and Induction Variable Elimination



Final Intermediate Code



Optimized Program Flow Graph

Nesting Level	No. of Statements	
	Original	Optimized
0	14	10
1	11	4
2	8	6

If we assume that a loop is executed 10 times, then the number of computations saved at run time

$$= (14 - 10) + (11 - 4) \times 10 + (8 - 6) \times 10^2 = 4 + 70 + 200 = 274$$



Observations

- Optimizations are transformations based on some information.
- Systematic analysis required for deriving the information.
- We have looked at data flow optimizations.
Many control flow optimizations can also be performed.



Categories of Optimizing Transformations and Analyses

Code Motion Redundancy Elimination Control flow Optimization	Machine Independent	Flow Analysis (Data + Control)
Loop Transformations	Machine Dependent	Dependence Analysis (Data + Control)
Instruction Scheduling Register Allocation Peephole Optimization	Machine Dependent	Several Independent Techniques
Vectorization Parallelization	Machine Dependent	Dependence Analysis (Data + Control)



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. multicore)



Why is it Useful?

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

- 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. multicore)
- 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. multicore)
- 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

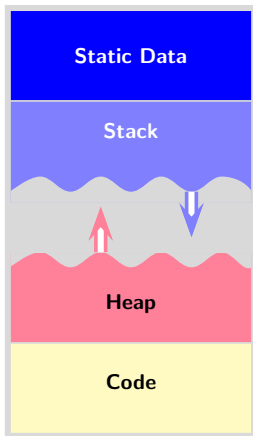
To understand the program



Part 3

Optimizing Heap Memory Usage

Standard Memory Architecture of Programs



Heap allocation provides the flexibility of

- *Variable Sizes.* Data structures can grow or shrink as desired at runtime.
(Not bound to the declarations in program.)
- *Variable Lifetimes.* Data structures can be created and destroyed as desired at runtime.
(Not bound to the activations of procedures.)

Managing Heap Memory

Decision 1: When to Allocate?

- **Explicit.** Specified in the programs. (eg. Imperative/OO languages)
- **Implicit.** Decided by the language processors. (eg. Declarative Languages)



Managing Heap Memory

Decision 1: When to Allocate?

- **Explicit.** Specified in the programs. (eg. Imperative/OO languages)
- **Implicit.** Decided by the language processors. (eg. Declarative Languages)

Decision 2: When to Deallocate?

- **Explicit.** Manual Memory Management (eg. C/C++)
- **Implicit.** Automatic Memory Management aka Garbage Collection (eg. Java/Declarative languages)



State of Art in Manual Deallocation

- Memory leaks
 - 10% to 20% of last development effort goes in plugging leaks
- Tool assisted manual plugging
 - Purify, Electric Fence, RootCause, GlowCode, yakTest, Leak Tracer, BDW Garbage Collector, mtrace, memwatch, dmalloc etc.*
- All leak detectors
 - ▶ are dynamic (and hence specific to execution instances)
 - ▶ generate massive reports to be perused by programmers
 - ▶ usually do not locate last use but only allocation escaping a call
 - ⇒ At which program point should a leak be “plugged”?



Garbage Collection \equiv Automatic Deallocation

- Retain active data structure.
Deallocate inactive data structure.
- What is an Active Data Structure?



Garbage Collection \equiv 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 \equiv 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?

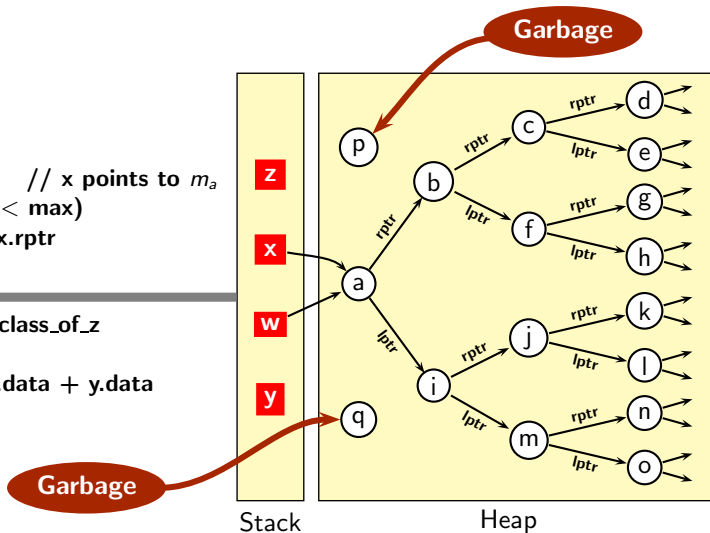


What is Garbage?

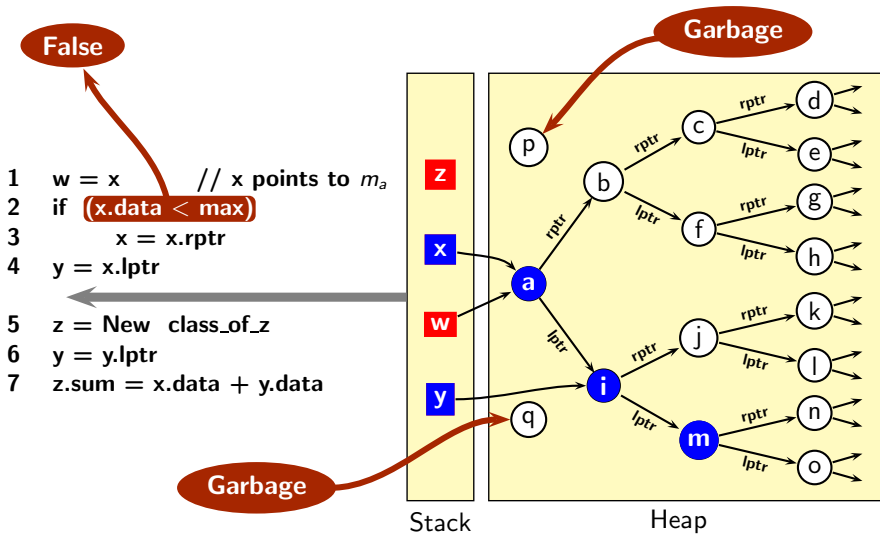
```

1  w = x      // x points to ma
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

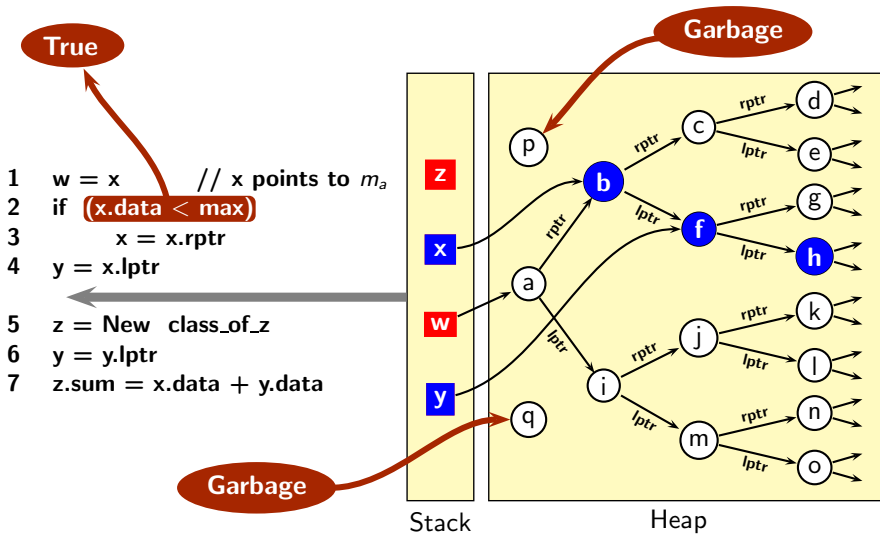
```



What is Garbage?



What is Garbage?

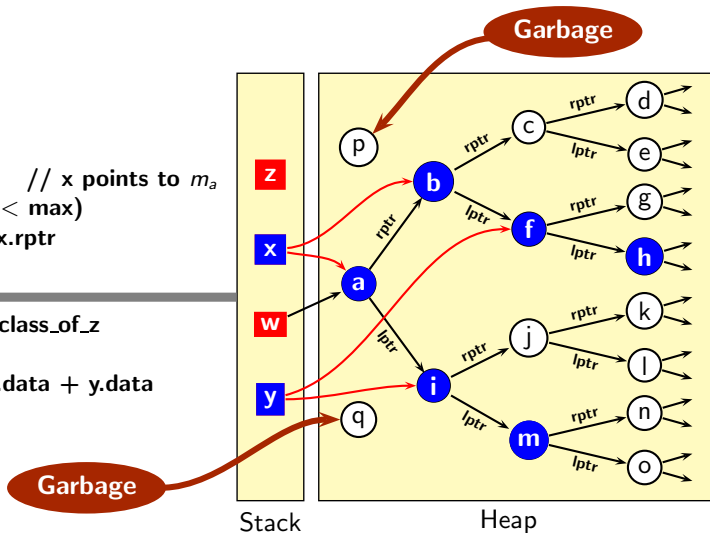


What is Garbage?

```

1  w = x      // x points to ma
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

Is Reachable Same as Live?

From www.memorymanagement.org/glossary

live (also known as alive, active) : Memory(2) or an object is live if the program will read from it in future. *The term is often used more broadly to mean reachable.*

It is not possible, in general, for garbage collectors to determine exactly which objects are still live. Instead, they use some approximation to detect objects that are provably dead, *such as those that are not reachable.*

Similar terms: reachable. Opposites: dead. See also: undead.



Is Reachable Same as Live?

- Not really. Most of us know that.

Even with the state of art of garbage collection, 24% to 76% unused memory remains unclaimed

- The state of art compilers, virtual machines, garbage collectors cannot distinguish between the two



Reachability and Liveness

Comparison between different sets of objects:

Live ? Reachable ? Allocated



Reachability and Liveness

Comparison between different sets of objects:

Live \subseteq Reachable \subseteq Allocated



Reachability and Liveness

Comparison between different sets of objects:

$$\text{Live} \subseteq \text{Reachable} \subseteq \text{Allocated}$$

The objects that are not live must be reclaimed.



Reachability and Liveness

Comparison between different sets of objects:

$$\text{Live} \subseteq \text{Reachable} \subseteq \text{Allocated}$$

The objects that are not live must be reclaimed.

$$\neg \text{Live} \quad ? \quad \neg \text{Reachable} \quad ? \quad \neg \text{Allocated}$$



Reachability and Liveness

Comparison between different sets of objects:

$$\text{Live} \subseteq \text{Reachable} \subseteq \text{Allocated}$$

The objects that are not live must be reclaimed.

$$\neg \text{Live} \supseteq \neg \text{Reachable} \supseteq \neg \text{Allocated}$$



Reachability and Liveness

Comparison between different sets of objects:

$$\text{Live} \subseteq \text{Reachable} \subseteq \text{Allocated}$$

The objects that are not live must be reclaimed.

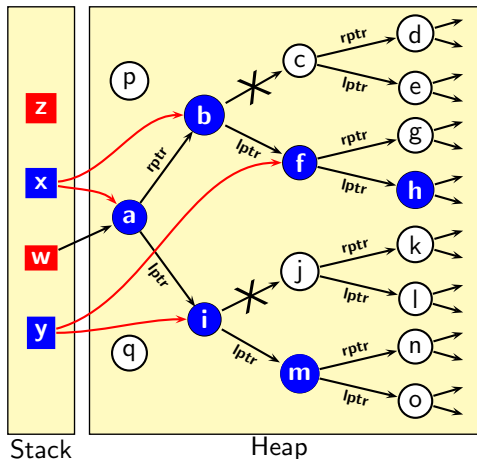
$$\neg \text{Live} \supseteq \neg \text{Reachable} \supseteq \neg \text{Allocated}$$

Garbage collectors
collect these



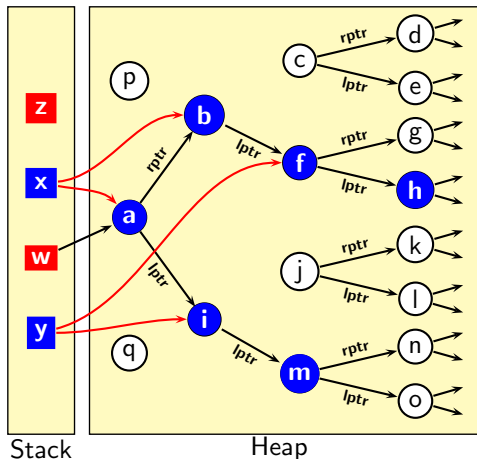
Cedar Mesa Folk Wisdom

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>)



Cedar Mesa Folk Wisdom

- Most promising, simplest to understand, yet the hardest to implement.
- Which references should be set to NULL?
 - ▶ Most approaches rely on feedback from profiling.
 - ▶ No systematic and clean solution.



Distinguishing Between Reachable and Live

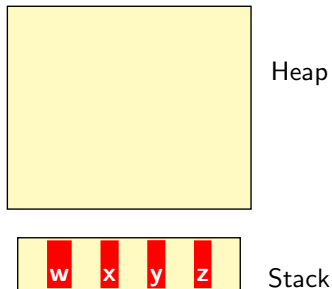
The state of art

- Eliminating objects reachable from root variables which are not live.
- Implemented in current Sun JVMs.
- Uses liveness data flow analysis of root variables (stack data).
- What about liveness of heap data?



Liveness of Stack Data: An Informal Introduction

```
1  w = x      // x points to ma
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
```

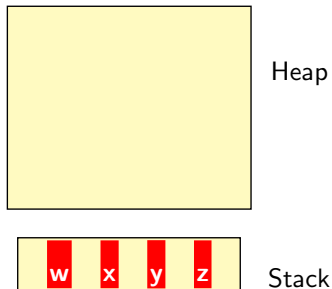


if changed to **while**



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
```

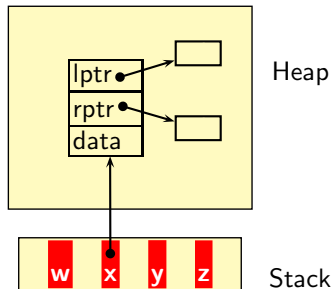


What is the
meaning of the *use*
of data?



Liveness of Stack Data: An Informal Introduction

```
1  w = x      // x points to ma
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?

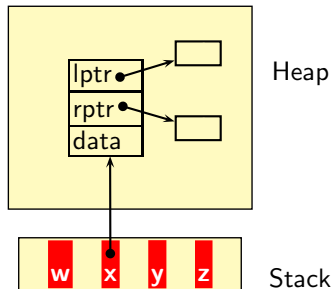


Liveness of Stack Data: An Informal Introduction

```

1  w = x      // x points to ma
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

```



Accessing the location
and reading its contents

What is the
meaning of the *use*
of data?

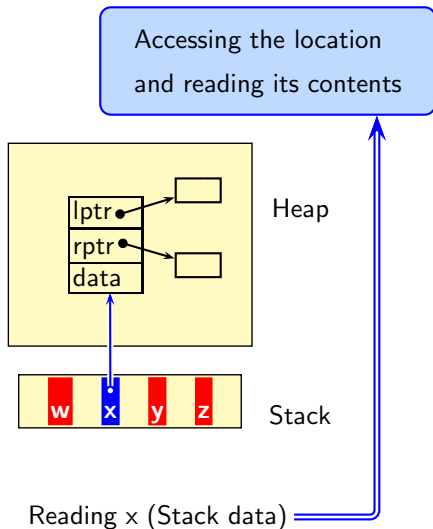


Liveness of Stack Data: An Informal Introduction

```

1  w = x      // x points to ma
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

```

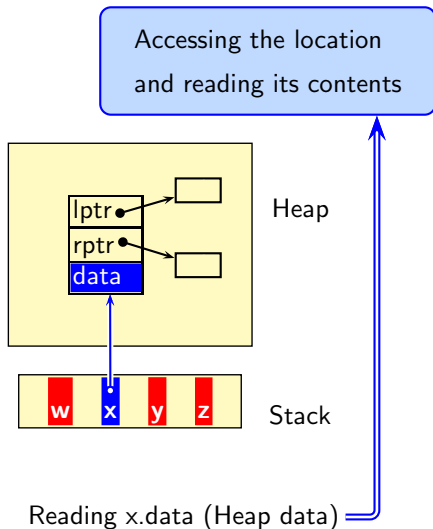


Liveness of Stack Data: An Informal Introduction

```

1  w = x      // x points to ma
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

```

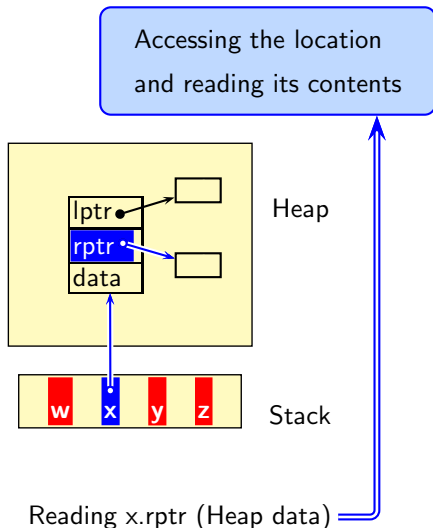


Liveness of Stack Data: An Informal Introduction

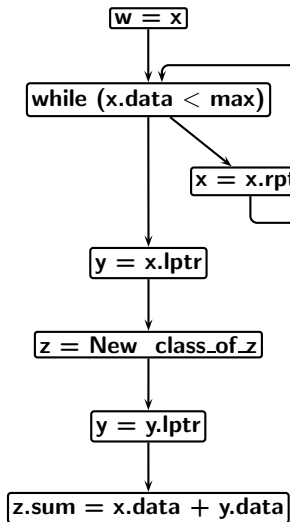
```

1  w = x      // x points to ma
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

```



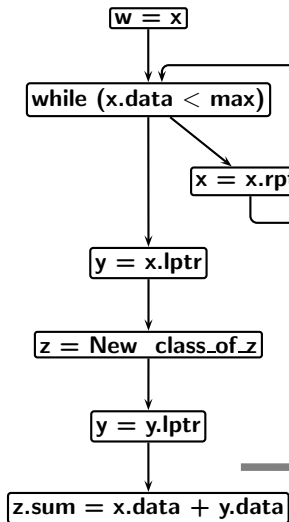
Liveness of Stack Data: An Informal Introduction



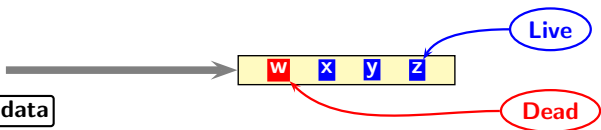
No variable is used beyond this program point



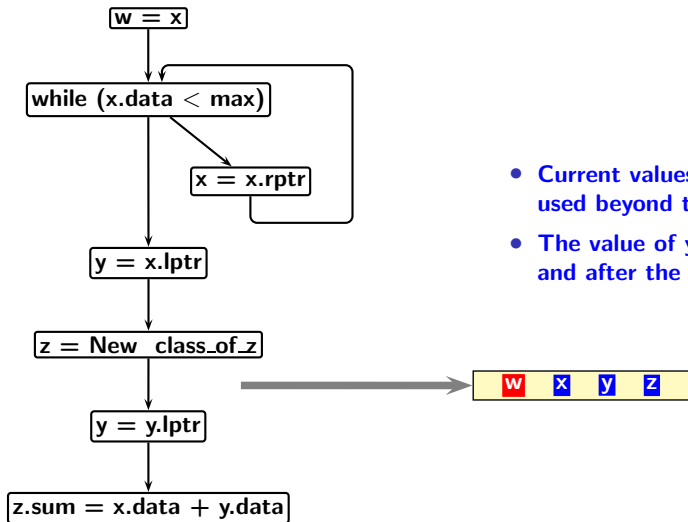
Liveness of Stack Data: An Informal Introduction



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



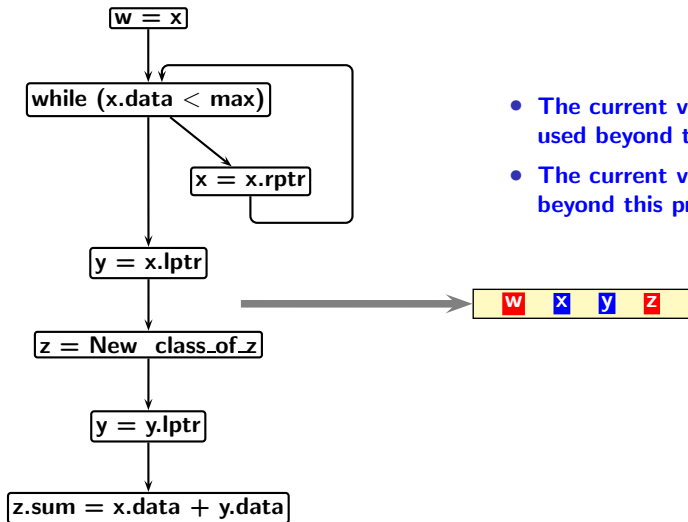
Liveness of Stack Data: An Informal Introduction



- 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`



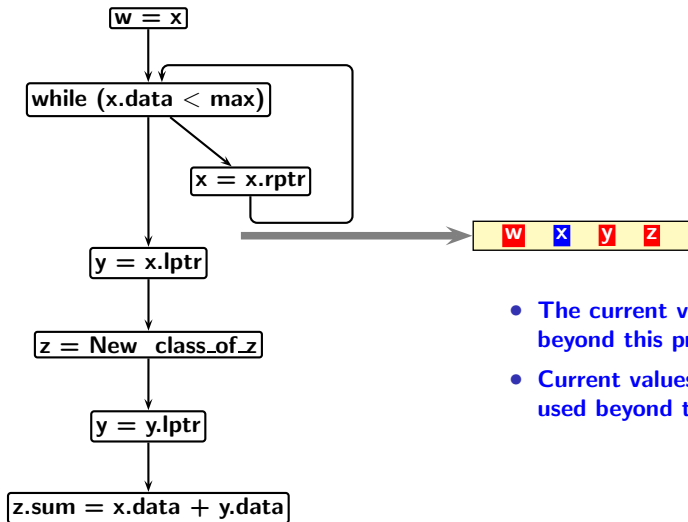
Liveness of Stack Data: An Informal Introduction



- 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



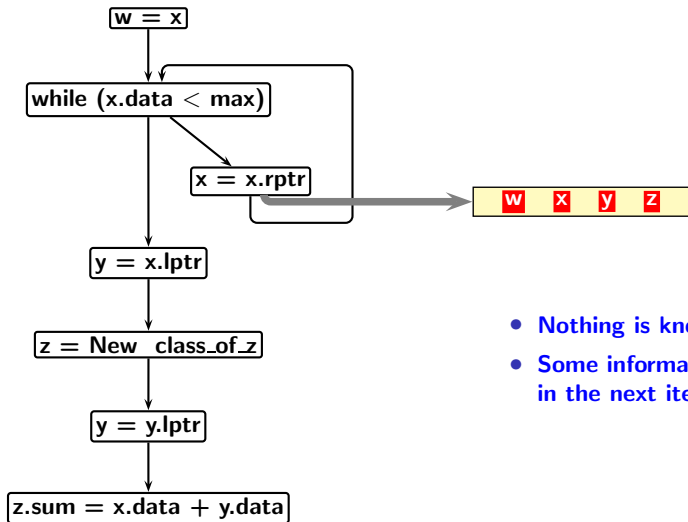
Liveness of Stack Data: An Informal Introduction



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



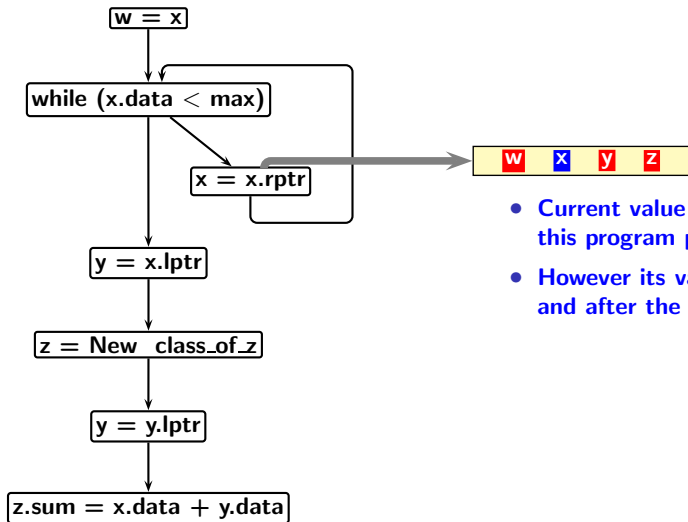
Liveness of Stack Data: An Informal Introduction



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



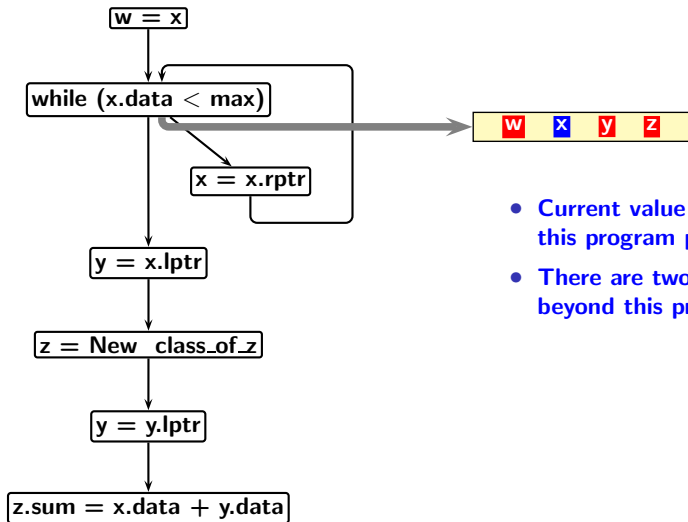
Liveness of Stack Data: An Informal Introduction



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



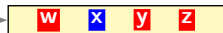
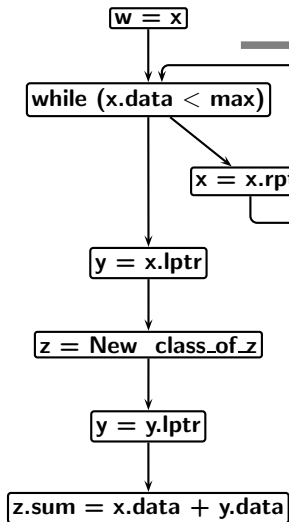
Liveness of Stack Data: An Informal Introduction



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



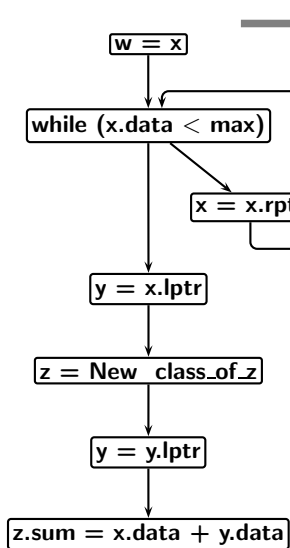
Liveness of Stack Data: An Informal Introduction



Current value of x is used beyond this program point



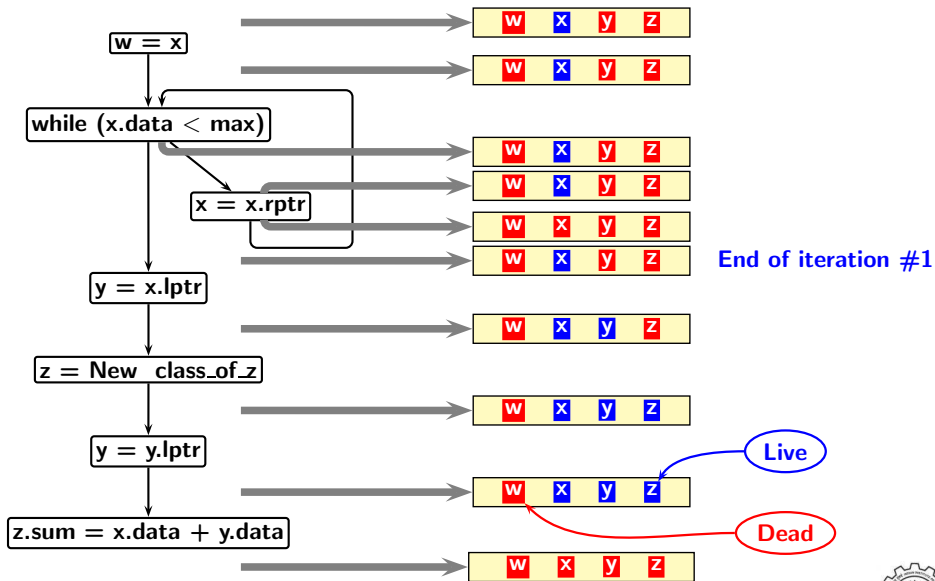
Liveness of Stack Data: An Informal Introduction



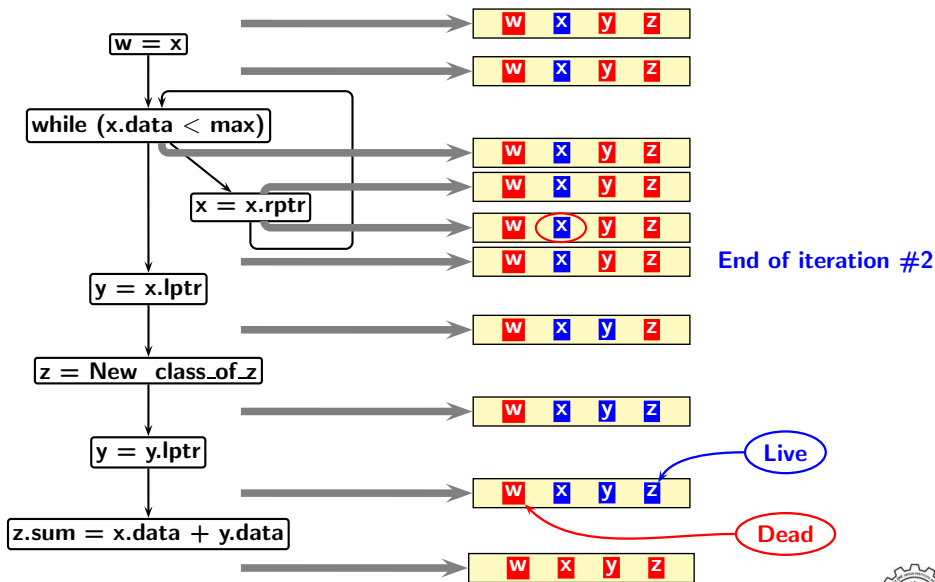
Current value of `x` is used beyond this program point



Liveness of Stack Data: An Informal Introduction



Liveness of Stack Data: An Informal Introduction



Applying Cedar Mesa Folk Wisdom to Heap Data

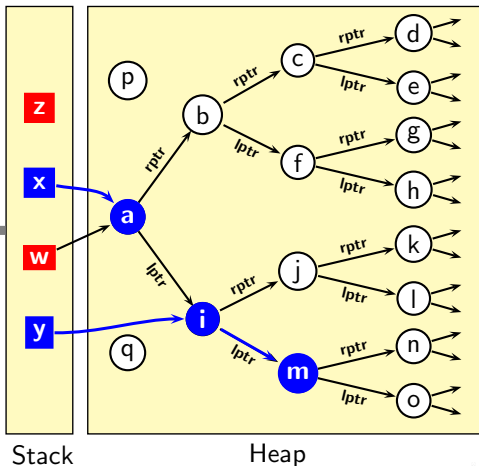
Liveness Analysis of Heap Data

If the **while** loop is not executed even once.

```

1  w = x      // x points to ma
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

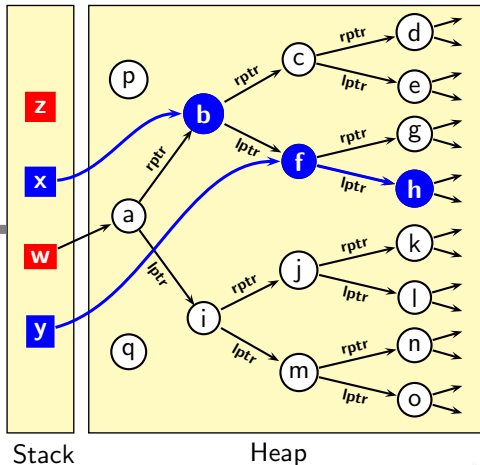
Liveness Analysis of Heap Data

If the **while** loop is executed once.

```

1  w = x      // x points to ma
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

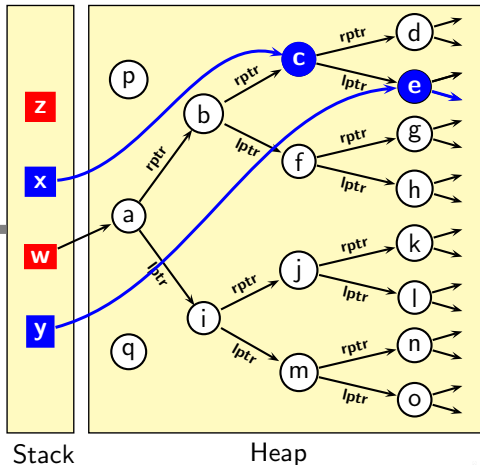
Liveness Analysis of Heap Data

If the **while** loop is executed twice.

```

1  w = x      // x points to ma
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)
    {
3      x = x.rptr                x.lptr = null
                                }
                                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
```



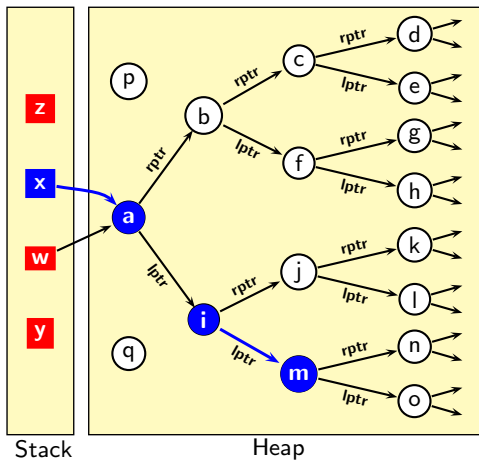
Our Solution

```

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


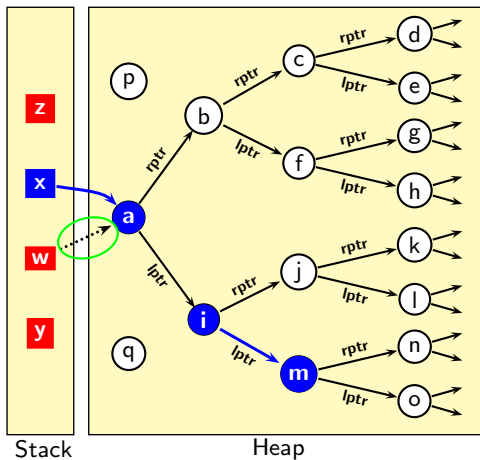
Our Solution

```

1  y = z = null
   w = x
   w = null
2  while (x.data < max)
   {
3     x.lptr = null
     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



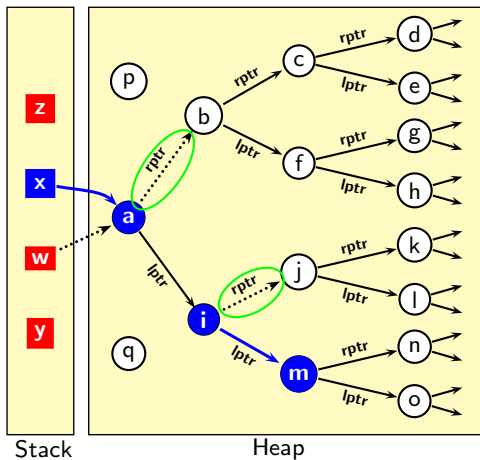
Our Solution

```

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

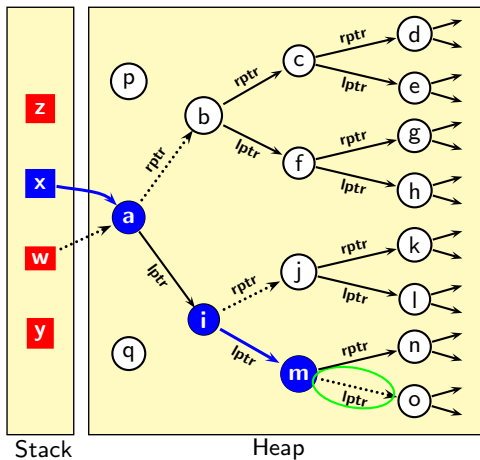


Our Solution

```

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



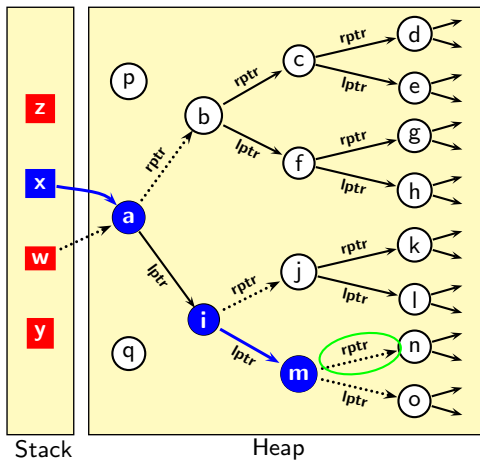
Our Solution

```

1  y = z = null
   w = x
   w = null
2  while (x.data < max)
   {
3     x.lptr = null
     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



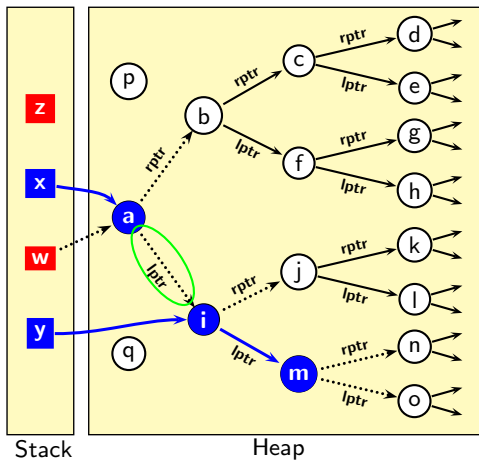
Our Solution

```

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

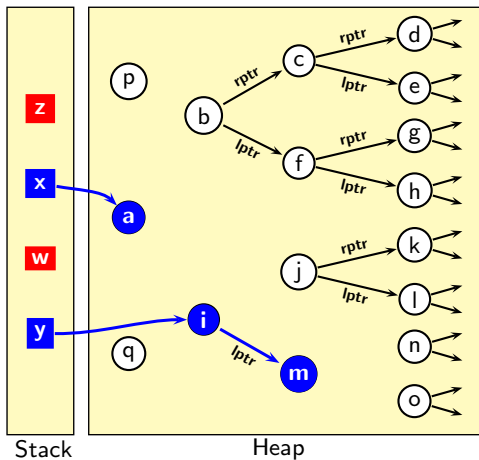


Our Solution

```

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


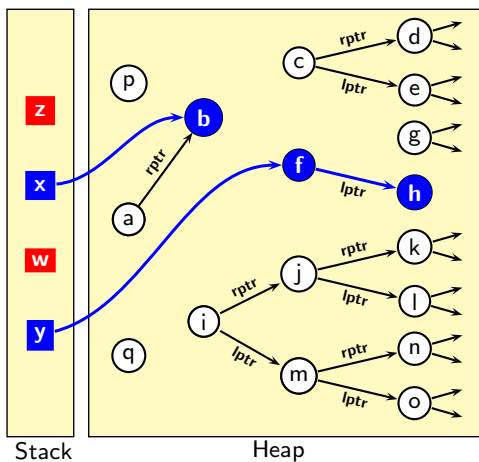
Our Solution

```

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


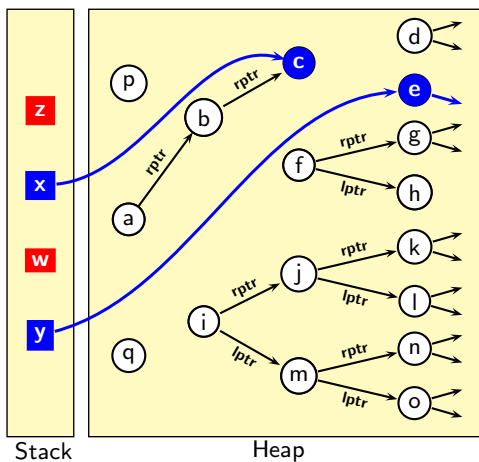
Our Solution

```

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

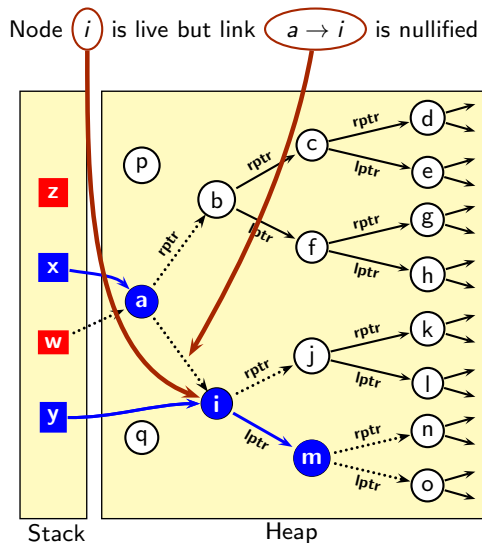


Some Observations

```

1  y = z = null
   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

```



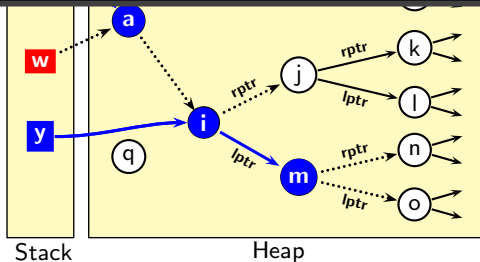
Some Observations

```

1  y = z = null
   w = x
   w = null
2  while (x.data < max)
   {
3     x.lptr = null
     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



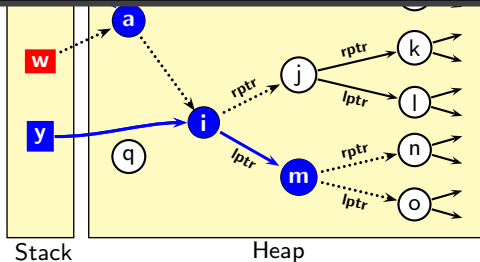
Some Observations

```

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



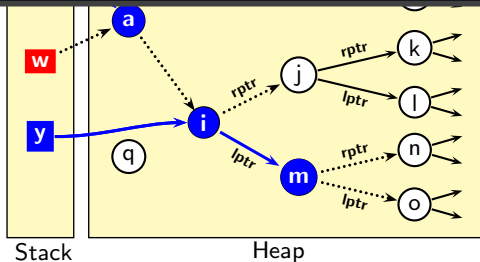
Some Observations

```

1  y = z = null
   w = x
   w = null
2  while (x.data < max)
   {
3     x.lptr = null
     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*



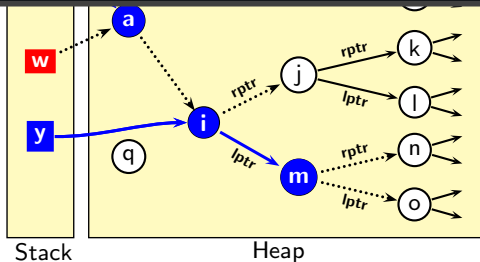
Some Observations

```

1  y = z = null
   w = x
   w = null
2  while (x.data < max)
   {
3     x.lptr = null
     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*



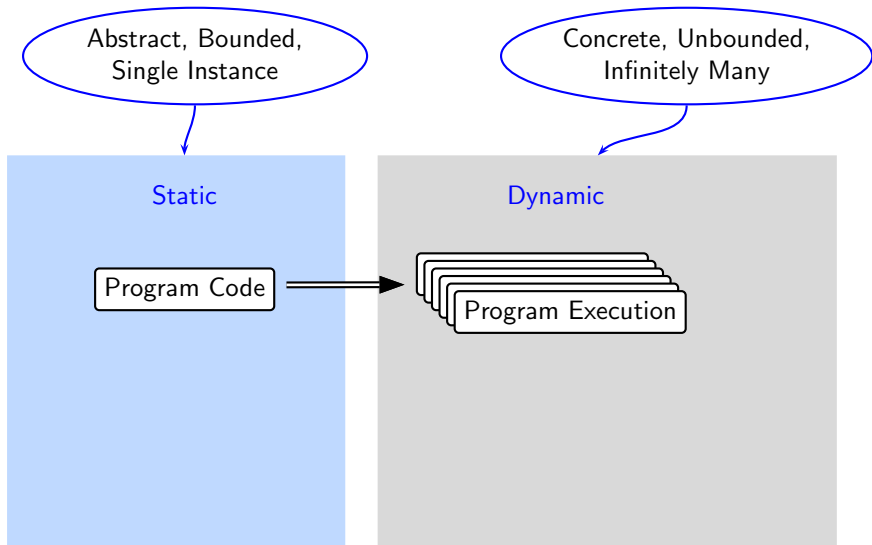
BTW, What is Static Analysis of Heap?

Static

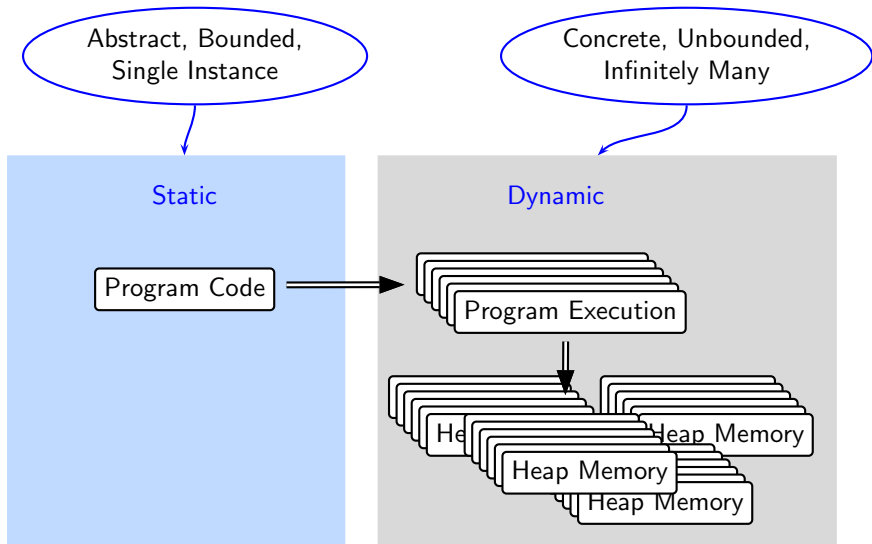
Dynamic



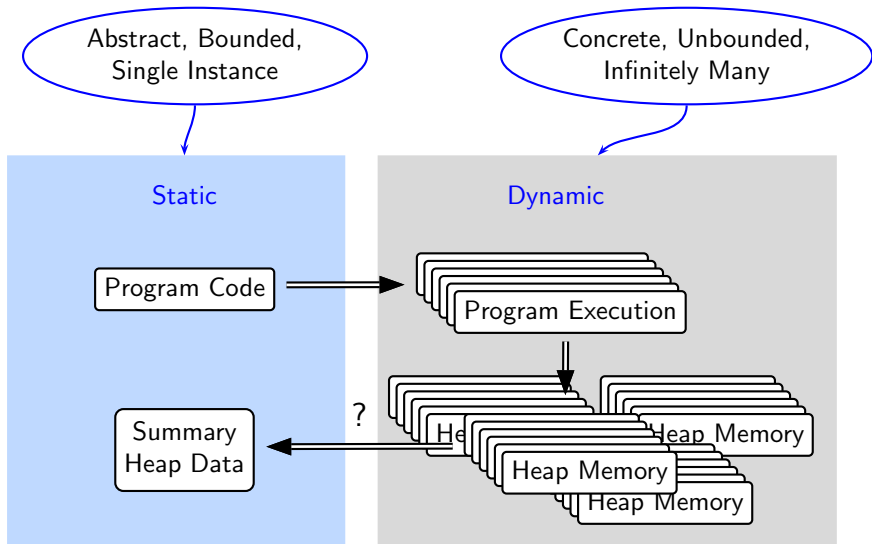
BTW, What is Static Analysis of Heap?



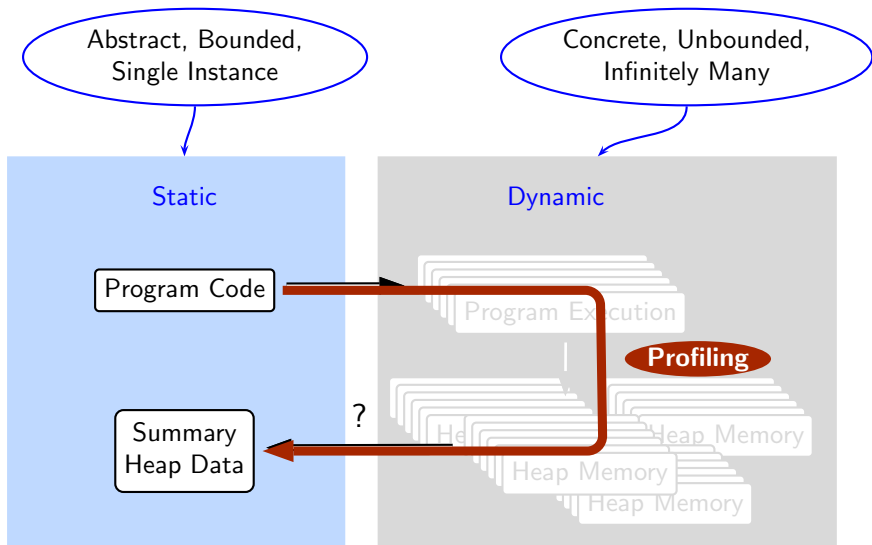
BTW, What is Static Analysis of Heap?



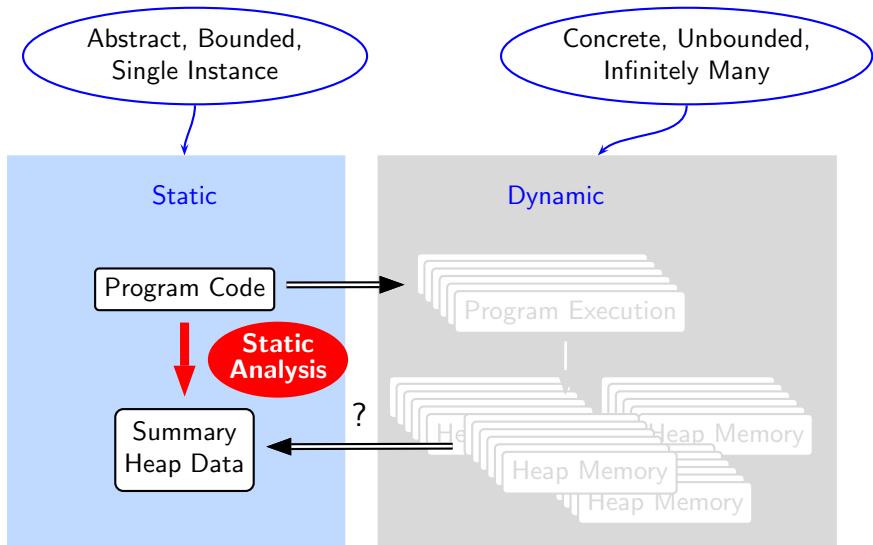
BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



Part 4

Course Details

The Main Theme of the Course

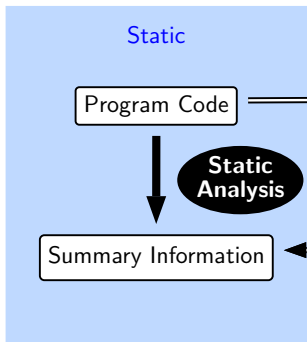
Constructing *suitable abstractions* for
sound & precise modelling of
runtime behaviour of programs
efficiently



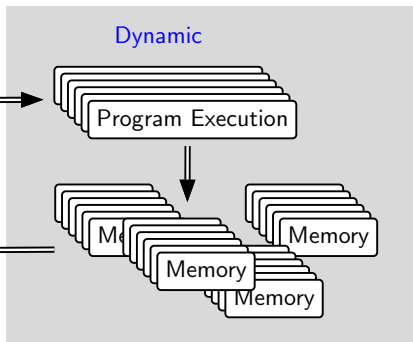
The Main Theme of the Course

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

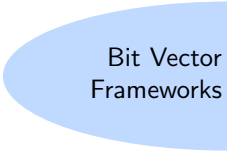
Abstract, Bounded, Single Instance



Concrete, Unbounded, Infinitely Many



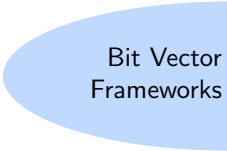
Sequence of Generalizations in the Course Modules




Bit Vector
Frameworks



Sequence of Generalizations in the Course Modules



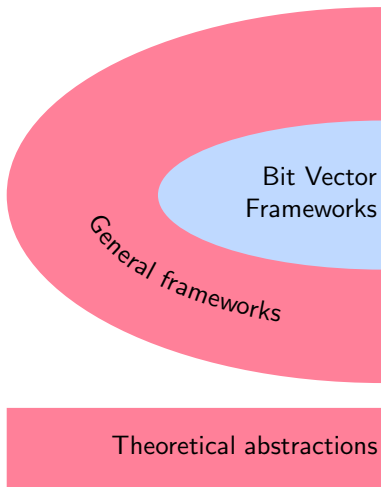
Bit Vector
Frameworks



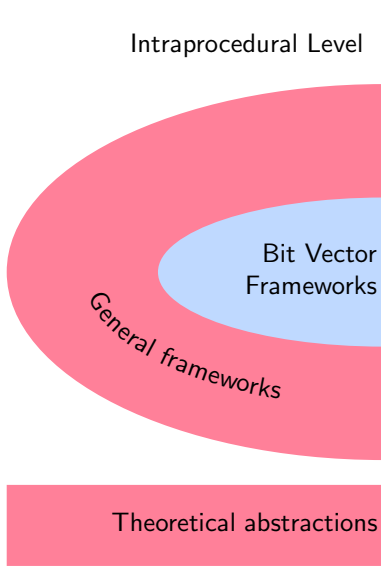
Theoretical abstractions



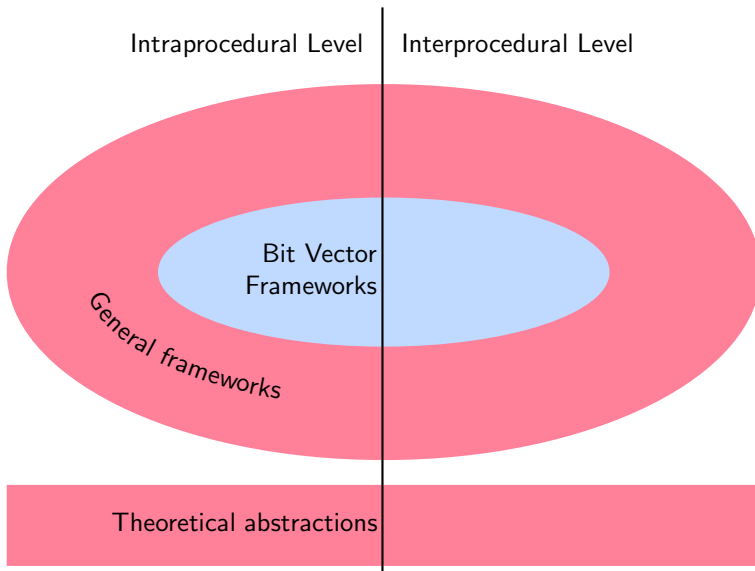
Sequence of Generalizations in the Course Modules



Sequence of Generalizations in the Course Modules



Sequence of Generalizations in the Course Modules



Course Pedagogy

- Interleaved lectures and tutorials
- Plenty of problem solving
- Practice problems will be provided,
 - ▶ Readymade solutions will not be provided
 - ▶ Your solutions will be checked
- Detailed course plan can be found at the course page:
<http://www.cse.iitb.ac.in/~uday/courses/cs618-17/>
- Moodle will be used extensively for announcements and discussions



Assessment Scheme

- Tentative plan

Mid Semester Examination	30%
End Semester Examination	40%
Two Quizzes	10%
Project	20%
Total	100%

- Can be fine tuned based on the class feedback



Course Strength and Selection Criteria

- Unavailability of TAs forces restricting the strength
Less than 30 is preferable, 40 is tolerable
- Course primarily aimed at M.Tech. 1 students
Follow up course and MTPs
- If the number is large, selection will be based on a test
 - ▶ Separate selection for M.Tech.1 and other students
 - ▶ Preference to M.Tech.1 students
 - ▶ May allow a reasonable number of audits
 - Attending all lectures is sufficient
 - No need to appear in examinations or do projects
 - ▶ Need to finalize the logistics of the test



Questions ??



Part 5

Program Model

Program Representation

- Three address code statements
 - ▶ Result, operator, operand1, operand2
 - ▶ Assignments, expressions, conditional jumps
 - ▶ Initially only scalars
Pointers, structures, arrays modelled later
- 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



An Example Program

```
int main()
{ int a, b, c, n;

  a = 4;
  b = 2;
  c = 3;
  n = c*2;
  while (a <= n)
  {
    a = a+1;
  }
  if (a < 12)
    a = a+b+c;
  return a;
}
```



An Example Program

```
int main()
{ int a, b, c, n;

  a = 4;
  b = 2;
  c = 3;
  n = c*2;
  while (a <= n)
  {
    a = a+1;
  }
  if (a < 12)
    a = a+b+c;
  return a;
}
```

1. a = 4
2. b = 2
3. c = 3
4. n = c*2
5. if (a > n)
 goto 8
6. a = a + 1
7. goto 5
8. if (a ≥ 12)
 goto 11
9. t1 = a+b
10. a = t1+c
11. return a



An Example Program

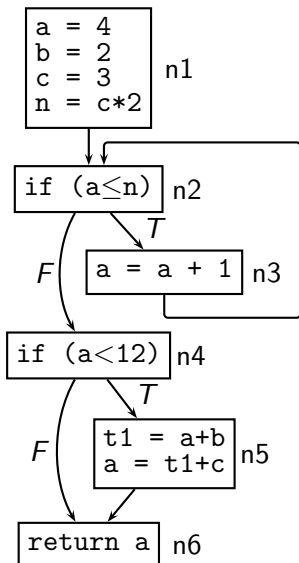
```

int main()
{ int a, b, c, n;

  a = 4;
  b = 2;
  c = 3;
  n = c*2;
  while (a <= n)
  {
    a = a+1;
  }
  if (a < 12)
    a = a+b+c;
  return a;
}

```

1. a = 4
2. b = 2
3. c = 3
4. n = c*2
5. if (a > n)
 - goto 8
6. a = a + 1
7. goto 5
8. if (a ≥ 12)
 - goto 11
9. t1 = a+b
10. a = t1+c
11. return a



Part 6

Soundness and Precision

Soundness and Precision of Static Analysis

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



Soundness and Precision of Static Analysis

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;
}
```

A blue box labeled "Absolute" is positioned above the `abs(b)` expression in the code. A blue arrow points from the box to the `abs(b)` expression.

Control Flow Graph

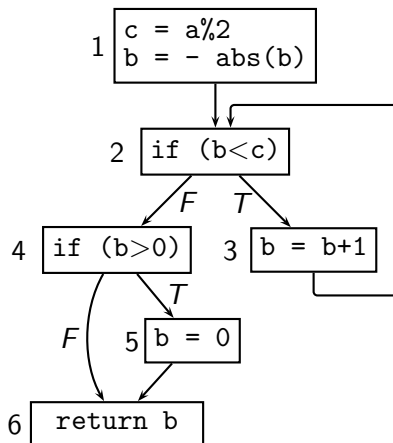


Soundness and Precision of Static Analysis

Example Program

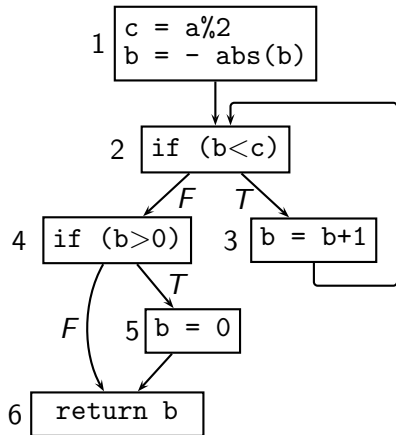
```
int a; Absolute
int f(int b)
{  int c;
   c = a%2;
   b = - abs(b);
   while (b < c)
       b = b+1;
   if (b > 0)
       b = 0;
   return b;
}
```

Control Flow Graph



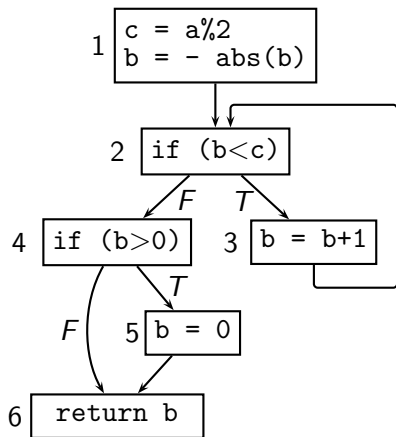
Execution Traces for Concrete Semantics

- A state: (Program Point, Variables \rightarrow Values)
- A trace: a valid sequence of states starting with a given initial state



Execution Traces for Concrete Semantics

- A state: (Program Point, Variables \rightarrow Values)
- A trace: a valid sequence of states starting with a given initial state



Trace 1

a b c

Entry₁, (5, 2, 7)

Entry₂, (5, -2, 1)

Entry₃, (5, -2, 1)

Entry₂, (5, -1, 1)

Entry₃, (5, -1, 1)

Entry₂, (5, 0, 1)

Entry₃, (5, 0, 1)

Entry₂, (5, 1, 1)

Entry₄, (5, 1, 1)

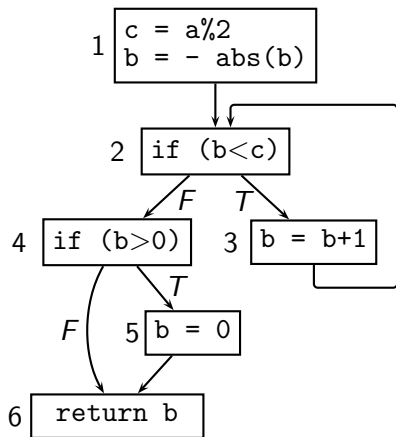
Entry₅, (5, 1, 1)

Entry₆, (5, 0, 1)



Execution Traces for Concrete Semantics

- A state: (Program Point, Variables \rightarrow Values)
- A trace: a valid sequence of states starting with a given initial state



Trace 1

a b c

Entry₁, (5, 2, 7)

Entry₂, (5, -2, 1)

Entry₃, (5, -2, 1)

Entry₂, (5, -1, 1)

Entry₃, (5, -1, 1)

Entry₂, (5, 0, 1)

Entry₃, (5, 0, 1)

Entry₂, (5, 1, 1)

Entry₄, (5, 1, 1)

Entry₅, (5, 1, 1)

Entry₆, (5, 0, 1)

Trace 1

a b c

Entry₁, (-5, -2, 8)

Entry₂, (-5, -2, -1)

Entry₃, (-5, -2, -1)

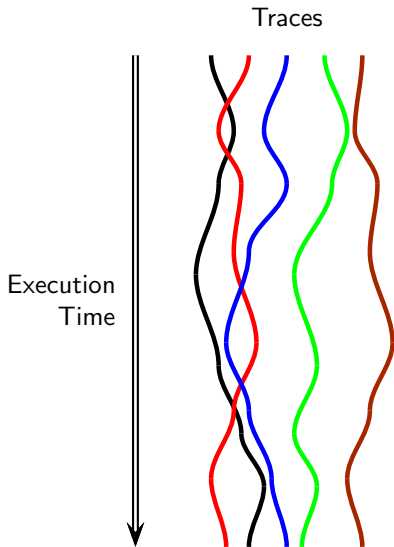
Entry₂, (-5, -1, -1)

Entry₄, (-5, -1, -1)

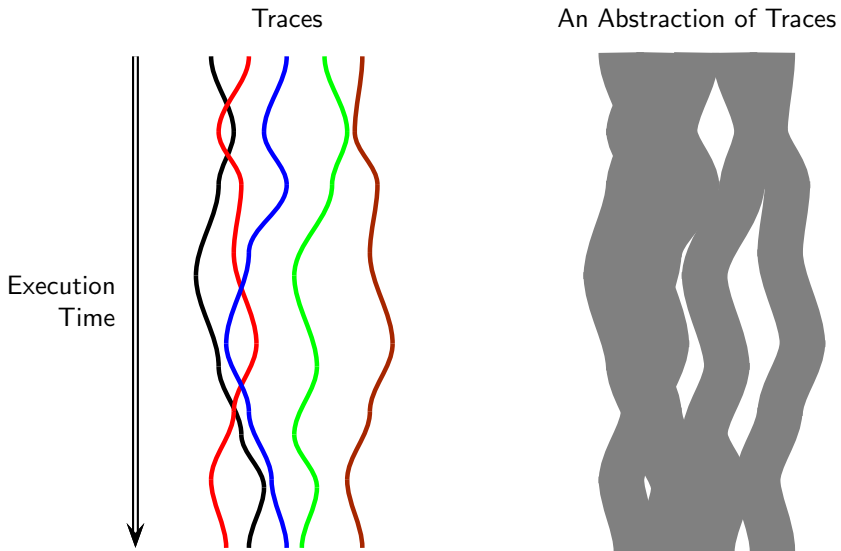
Entry₆, (-5, -1, -1)



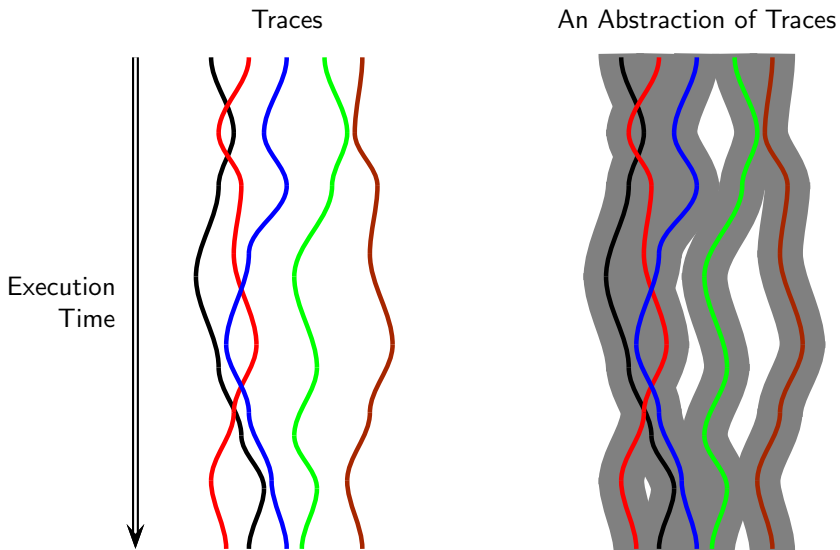
Static Analysis Computes Abstractions of Traces (1)



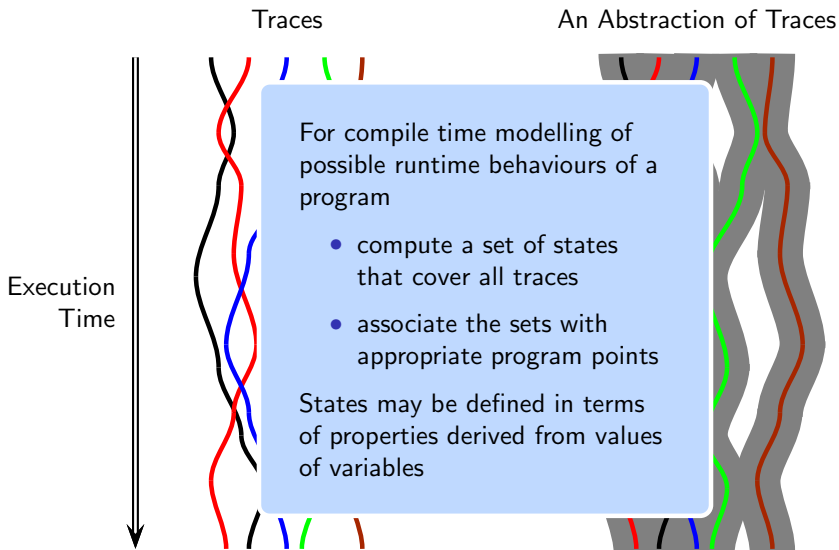
Static Analysis Computes Abstractions of Traces (1)



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

Entry₁, (5, 2, 7)

Entry₂, (5, -2, 1)

Entry₃, (5, -2, 1)

Entry₂, (5, -1, 1)

Entry₃, (5, -1, 1)

Entry₂, (5, 0, 1)

Entry₃, (5, 0, 1)

Entry₂, (5, 1, 1)

Entry₄, (5, 1, 1)

Entry₅, (5, 1, 1)

Entry₆, (5, 0, 1)

Trace 1

a b c

Entry₁, (-5, -2, 8)

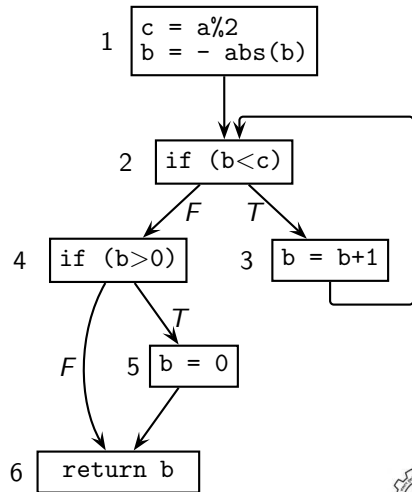
Entry₂, (-5, -2, -1)

Entry₃, (-5, -2, -1)

Entry₂, (-5, -1, -1)

Entry₄, (-5, -1, -1)

Entry₆, (-5, -1, -1)



Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

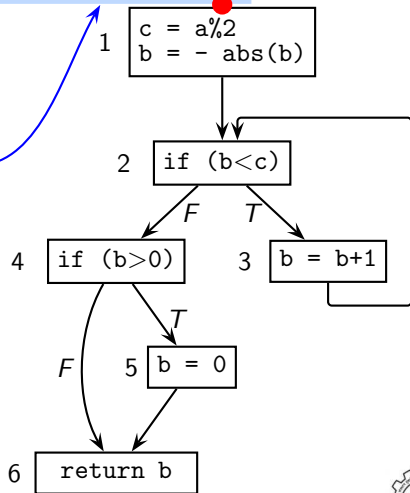
Trace 1

	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 1

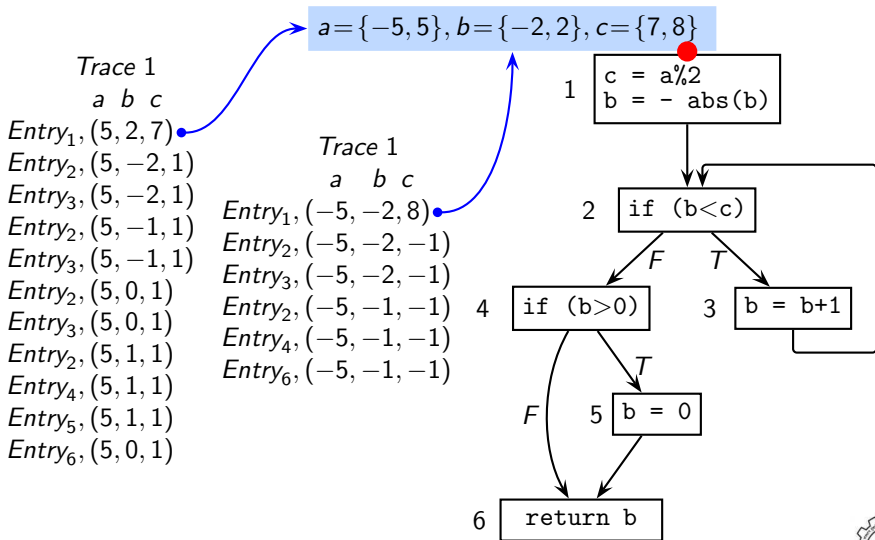
	a	b	c
Entry ₁	-5	-2	8
Entry ₂	-5	-2	-1
Entry ₃	-5	-2	-1
Entry ₂	-5	-1	-1
Entry ₄	-5	-1	-1
Entry ₆	-5	-1	-1

$\{(5, 2, 7), (-5, -2, 8)\}$



Static Analysis Computes Abstractions of Traces (2)

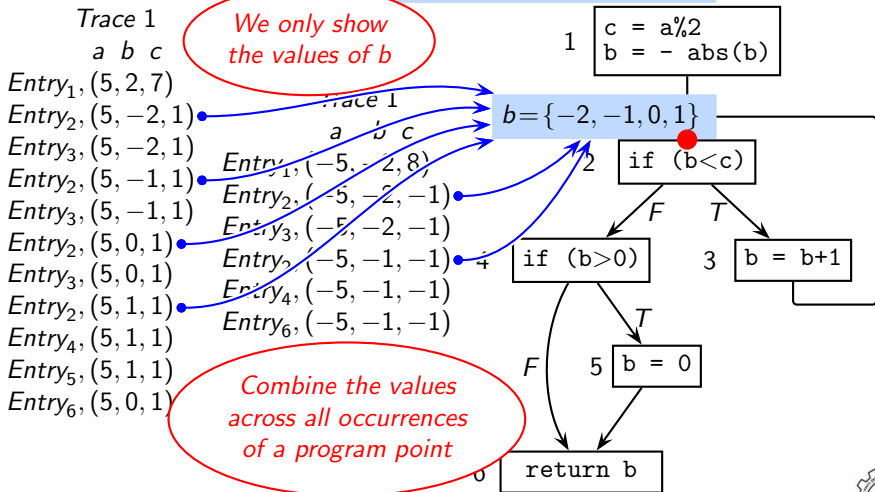
A possible static abstraction using sets



Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

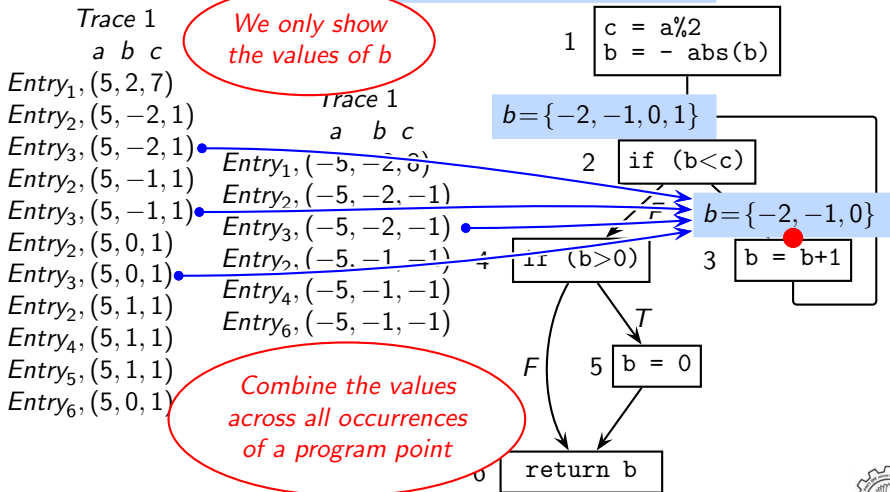
$$a = \{-5, 5\}, b = \{-2, 2\}, c = \{7, 8\}$$



Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

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



Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

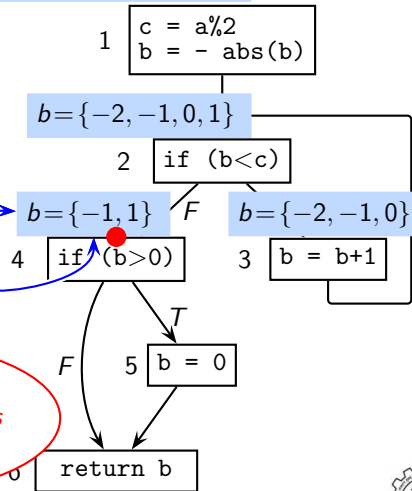
$$a = \{-5, 5\}, b = \{-2, 2\}, c = \{7, 8\}$$

Trace 1
 a b c
 Entry₁, (5, 2, 7)
 Entry₂, (5, -2, 1)
 Entry₃, (5, -2, 1)
 Entry₂, (5, -1, 1)
 Entry₃, (5, -1, 1)
 Entry₂, (5, 0, 1)
 Entry₃, (5, 0, 1)
 Entry₂, (5, 1, 1)
 Entry₄, (5, 1, 1)
 Entry₅, (5, 1, 1)
 Entry₆, (5, 0, 1)

We only show
the values of b

Trace 1
 a b c
 Entry₁, (-5, -2, 8)
 Entry₂, (-5, -2, -1)
 Entry₃, (-5, -2, -1)
 Entry₂, (-5, -1, -1)
 Entry₄, (-5, -1, -1)
 Entry₆, (-5, -1, -1)

Combine the values
across all occurrences
of a program point



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

Entry₁, (5, 2, 7)

Entry₂, (5, -2, 1)

Entry₃, (5, -2, 1)

Entry₂, (5, -1, 1)

Entry₃, (5, -1, 1)

Entry₂, (5, 0, 1)

Entry₃, (5, 0, 1)

Entry₂, (5, 1, 1)

Entry₄, (5, 1, 1)

Entry₅, (5, 1, 1)

Entry₆, (5, 0, 1)

We only show
the values of b

Trace 1

a b c

Entry₁, (-5, -2, 8)

Entry₂, (-5, -2, -1)

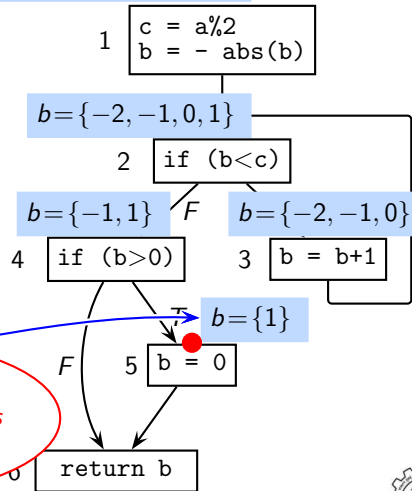
Entry₃, (-5, -2, -1)

Entry₂, (-5, -1, -1)

Entry₄, (-5, -1, -1)

Entry₆, (-5, -1, -1)

Combine the values
across all occurrences
of a program point



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

Entry₁, (5, 2, 7)

Entry₂, (5, -2, 1)

Entry₃, (5, -2, 1)

Entry₂, (5, -1, 1)

Entry₃, (5, -1, 1)

Entry₂, (5, 0, 1)

Entry₃, (5, 0, 1)

Entry₂, (5, 1, 1)

Entry₄, (5, 1, 1)

Entry₅, (5, 1, 1)

Entry₆, (5, 0, 1)

We only show
the values of b

Trace 1

a b c

Entry₁, (-5, -2, 8)

Entry₂, (-5, -2, -1)

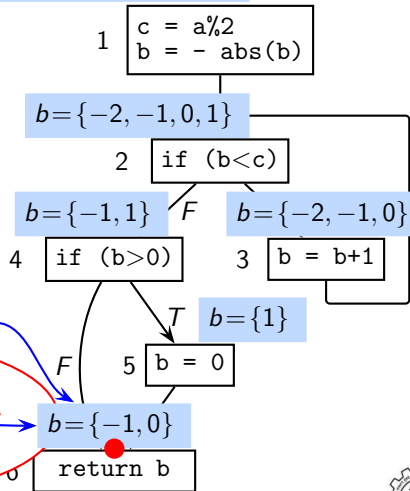
Entry₃, (-5, -2, -1)

Entry₂, (-5, -1, -1)

Entry₄, (-5, -1, -1)

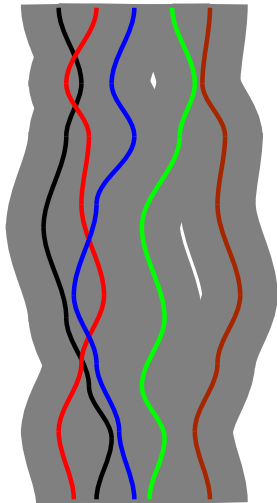
Entry₆, (-5, -1, -1)

Combine the values
across all occurrences
of a program point



Soundness of Abstractions (1)

Sound

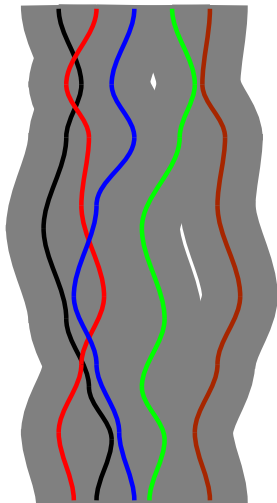


- An over-approximation of traces is sound



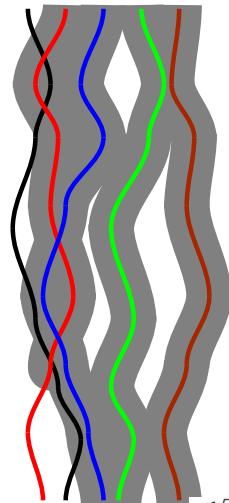
Soundness of Abstractions (1)

Sound



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

Unsound



Soundness of Abstractions (2)

An unsound abstraction

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

1 $c = a \% 2$
 $b = -\text{abs}(b)$

$b = \{-2, -1, 0, 1\}$

2 if ($b < c$)

$b = \{-1, 1\}$ F

$b = \{-2, -1, 0\}$

4 if ($b > 0$)

3 $b = b + 1$

$b = \{1\}$ T

5 $b = 0$ F

$b = \{-1, 0\}$

6 return b

All variables can have arbitrary values at the start.

b can have many more values at the entry of

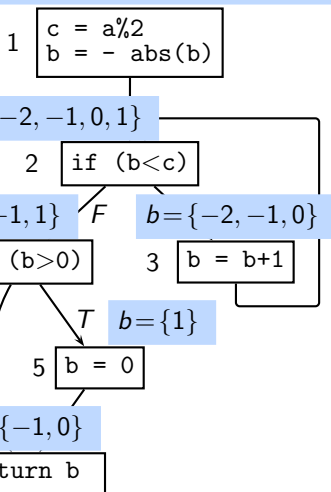
- blocks 2 and 3 (e.g. -3, -8, ...)
- block 4 (e.g. 0)



Soundness of Abstractions (2)

An unsound abstraction

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



A sound abstraction using intervals

- Overapproximated range of values denoted by

$$\left[\text{low_limit}, \text{high_limit} \right]$$

- Inclusive limits with

$$\text{low_limit} \leq \text{high_limit}$$

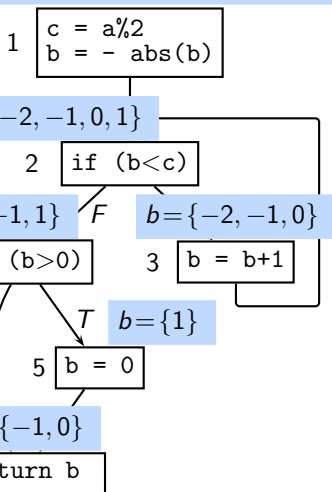
- One continuous range per variable with no “holes”



Soundness of Abstractions (2)

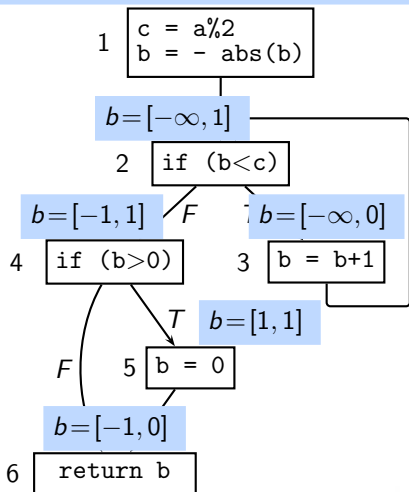
An unsound abstraction

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



A sound abstraction using intervals

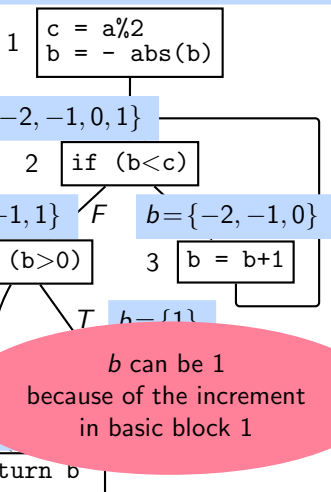
$a = [-\infty, \infty], b = [-\infty, \infty], c = [-\infty, \infty]$



Soundness of Abstractions (2)

An unsound abstraction

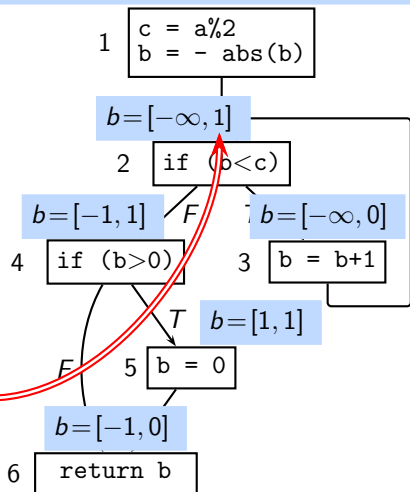
$a = \{-5, 5\}, b = \{-2, 2\}, c = \{7, 8\}$



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

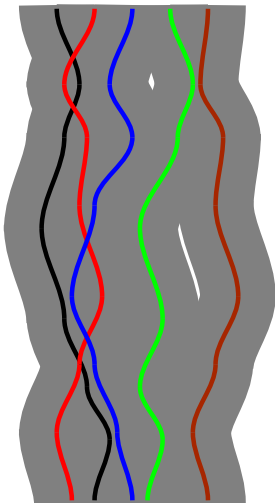
A sound abstraction using intervals

$a = [-\infty, \infty], b = [-\infty, \infty], c = [-\infty, \infty]$



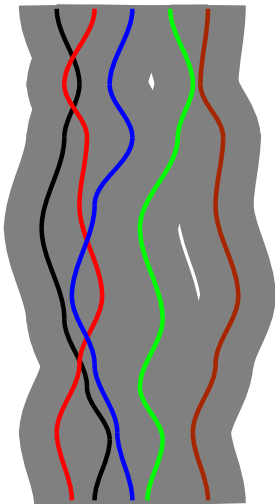
Precision of Sound Abstractions(1)

Sound but imprecise

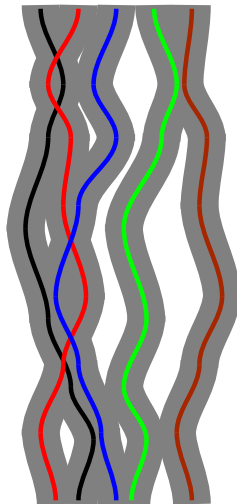


Precision of Sound Abstractions(1)

Sound but imprecise

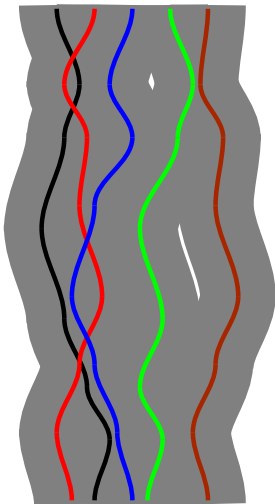


Sound and more precise

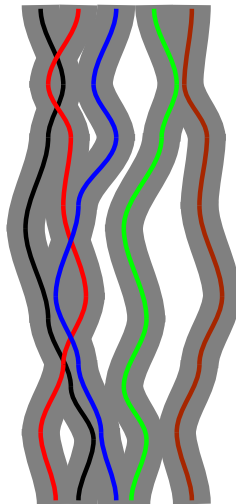


Precision of Sound Abstractions(1)

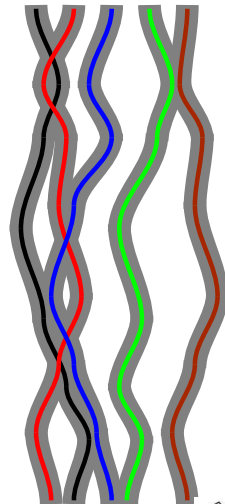
Sound but imprecise



Sound and more precise



Sound and even more precise

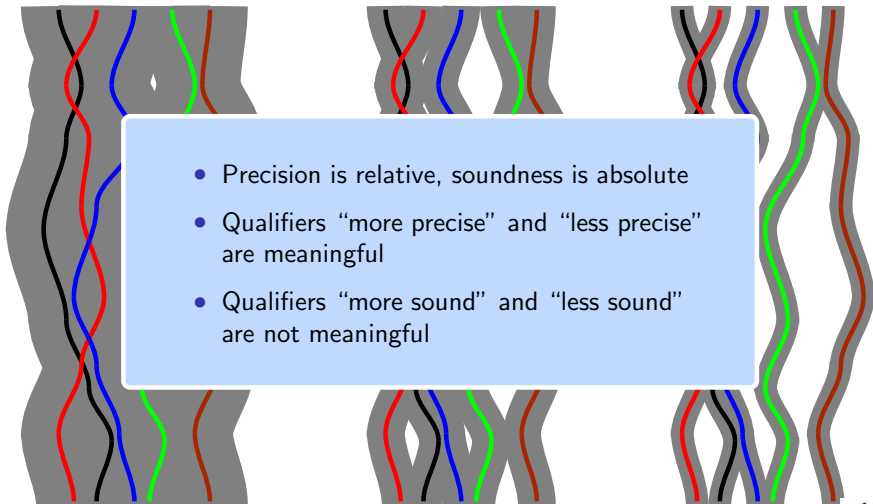


Precision of Sound Abstractions(1)

Sound but imprecise

Sound and more precise

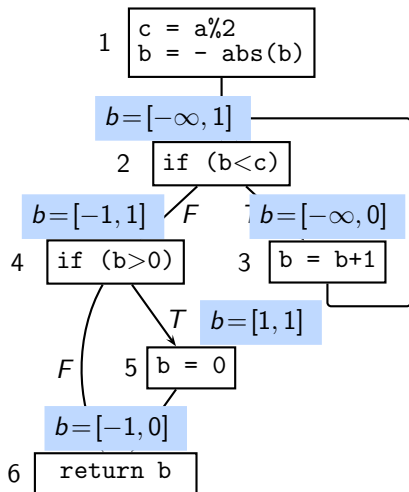
Sound and even more precise



Precision of Sound Abstractions(2)

A precise abstraction using intervals

$a = [-\infty, \infty], b = [-\infty, \infty], c = [-\infty, \infty]$



An imprecise abstraction using intervals

$a = [-\infty, \infty], b = [-\infty, \infty], c = [-\infty, \infty]$

