Introduction to Program Analysis

Uday Khedker

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

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

Dec 2019

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.

Motivating the Need of Program Analysis

- Some representative examples
    - Classical optimizations performed by compilers
    - Optimizing heap memory usage
- 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 >= 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
```

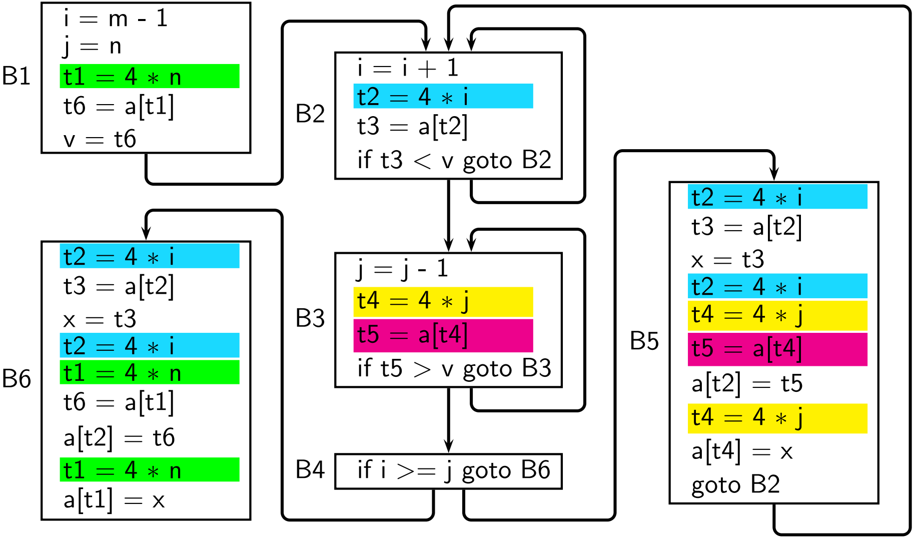
```
10. j = j - 1
11. t4 = 4 * j
12. t5 = a[t4]
13. if t5 > v goto 10
```

```
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



B1
```
i = m - 1
j = n
t1 = 4 * n
t6 = a[t1]
v = t6
```

B2
```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

B3
```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

B4
```
if i >= j goto B6
```

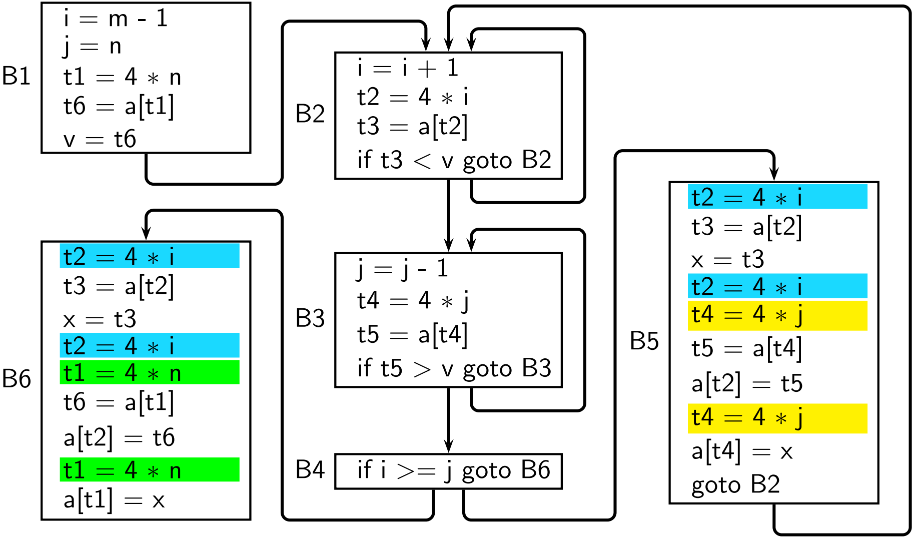
B5
```
t2 = 4 * i
t3 = a[t2]
x = t3
t2 = 4 * i
t4 = 4 * j
t5 = a[t4]
a[t2] = t5
t4 = 4 * j
a[t4] = x
goto B2
```

B6
```
t2 = 4 * i
t3 = a[t2]
x = t3
t2 = 4 * i
t1 = 4 * n
t6 = a[t1]
a[t2] = t6
t1 = 4 * n
a[t1] = x
```

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



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

```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

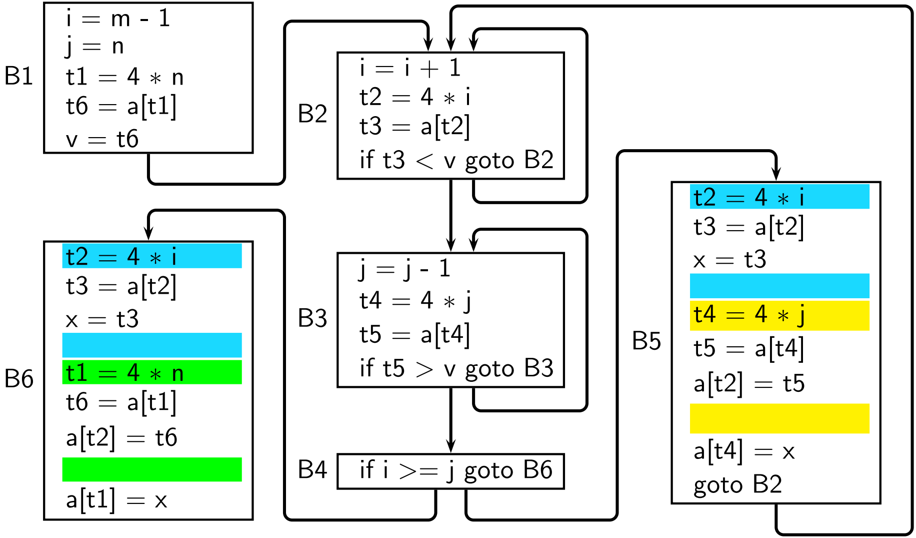
```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

```
if i >= j goto B6
```

```
t2 = 4 * i
t3 = a[t2]
x = t3
t2 = 4 * i
t4 = 4 * j
t5 = a[t4]
a[t2] = t5
t4 = 4 * j
a[t4] = x
goto B2
```

```
t2 = 4 * i
t3 = a[t2]
x = t3
t2 = 4 * i
t1 = 4 * n
t6 = a[t1]
a[t2] = t6
t1 = 4 * n
a[t1] = x
```

Local Common Subexpression Elimination



B1
```
i = m - 1
j = n
t1 = 4 * n
t6 = a[t1]
v = t6
```

B2
```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

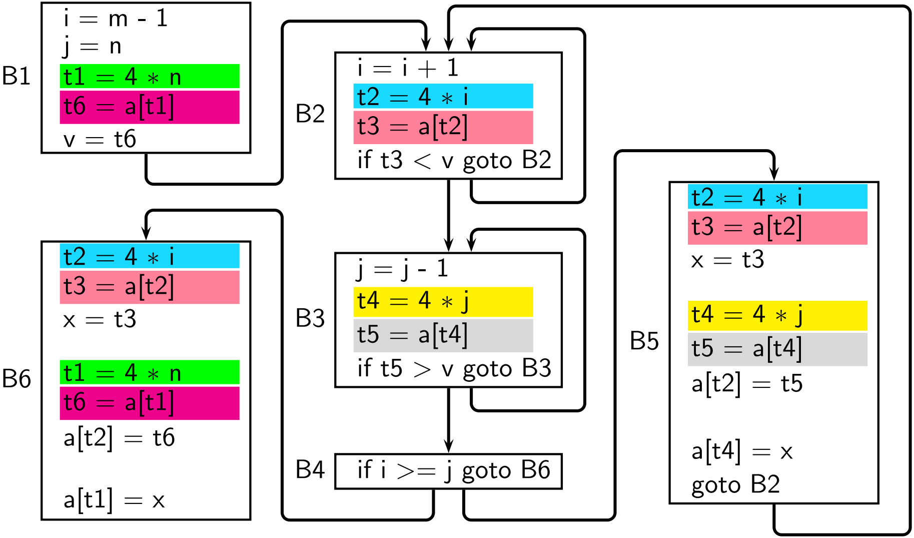
B3
```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

B4
```
if i >= j goto B6
```

B5
```
t2 = 4 * i
t3 = a[t2]
x = t3

t4 = 4 * j
t5 = a[t4]
a[t2] = t5

a[t4] = x
goto B2
```

B6
```
t2 = 4 * i
t3 = a[t2]
x = t3

t1 = 4 * n
t6 = a[t1]
a[t2] = t6

a[t1] = x
```

Global Common Subexpression Elimination



B1

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

B2

```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

B3

```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

B4

```
if i >= j goto B6
```

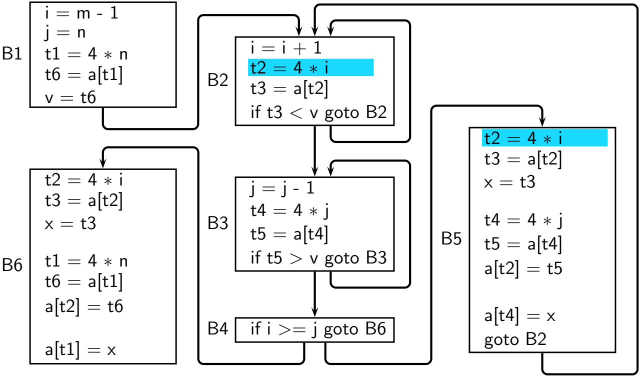
B5

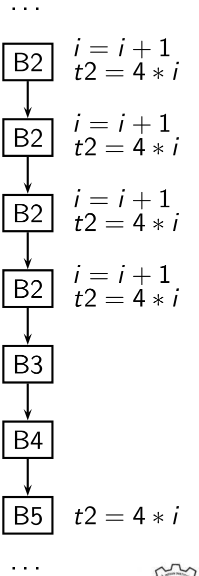
```
t2 = 4 * i
t3 = a[t2]
x = t3

t4 = 4 * j
t5 = a[t4]
a[t2] = t5

a[t4] = x
goto B2
```

B6

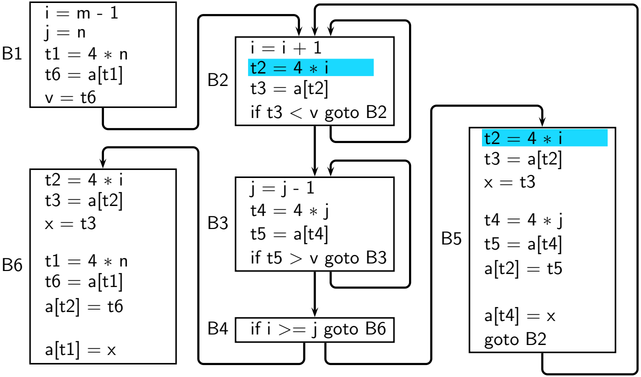
```
t2 = 4 * i
t3 = a[t2]
x = t3

t1 = 4 * n
t6 = a[t1]
a[t2] = t6

a[t1] = x
```

Global Common Subexpression Elimination





Global Common Subexpression Elimination



B1
```
i = m - 1
j = n
t1 = 4 * n
t6 = a[t1]
v = t6
```

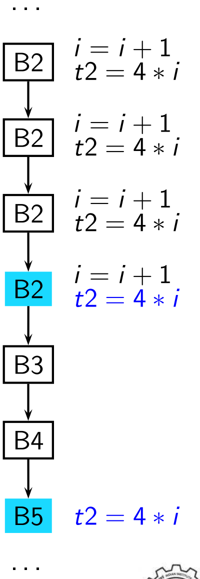
B2
```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

B3
```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

B4
```
if i >= j goto B6
```

B5
```
t2 = 4 * i
t3 = a[t2]
x = t3

t4 = 4 * j
t5 = a[t4]
a[t2] = t5

a[t4] = x
goto B2
```

B6
```
t2 = 4 * i
t3 = a[t2]
x = t3

t1 = 4 * n
t6 = a[t1]
a[t2] = t6

a[t1] = x
```



. . .

B2 $i = i + 1$ $t2 = 4 * i$

B2 $i = i + 1$ $t2 = 4 * i$

B2 $i = i + 1$ $t2 = 4 * i$

B2 $i = i + 1$ $t2 = 4 * i$

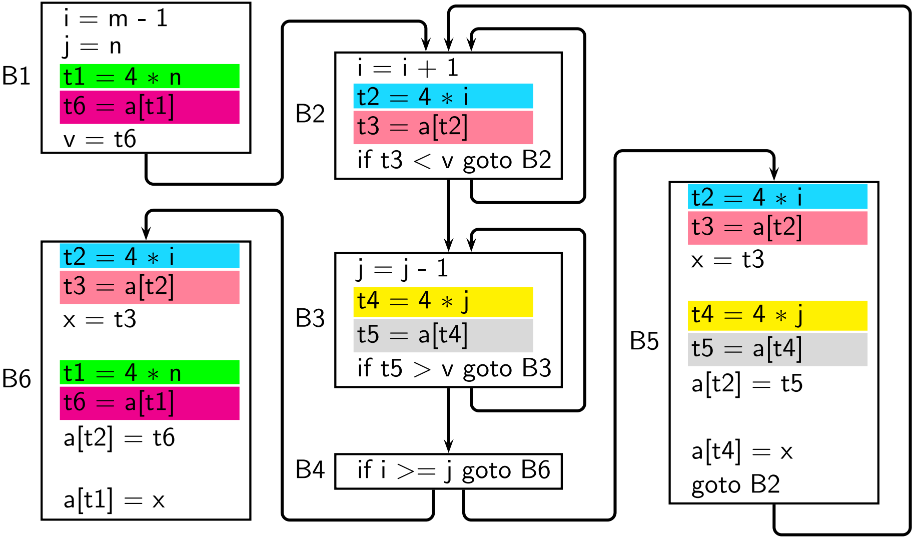
B3

B4

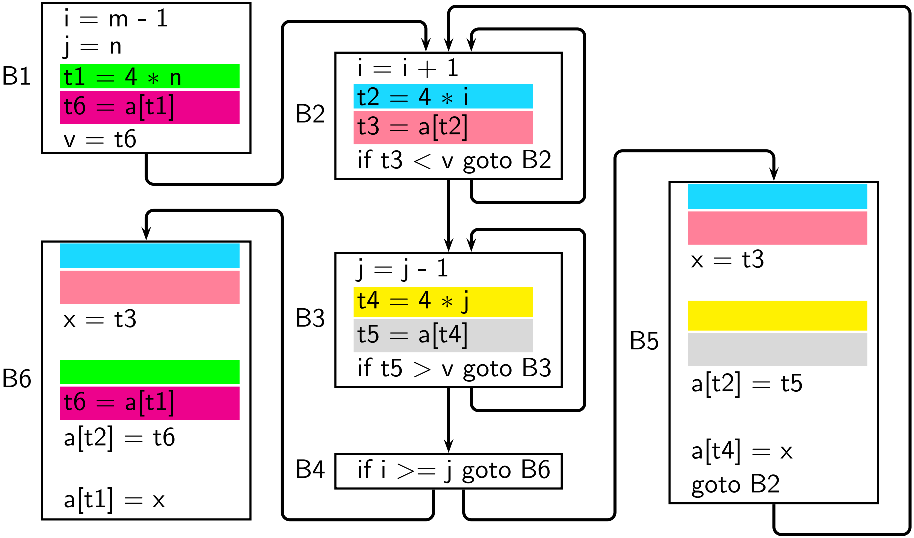
B5 $t2 = 4 * i$

. . .

Global Common Subexpression Elimination



Global Common Subexpression Elimination



B1

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

B2

```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

B3

```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

B4

```
if i >= j goto B6
```

B5

```
x = t3
a[t2] = t5
a[t4] = x
goto B2
```

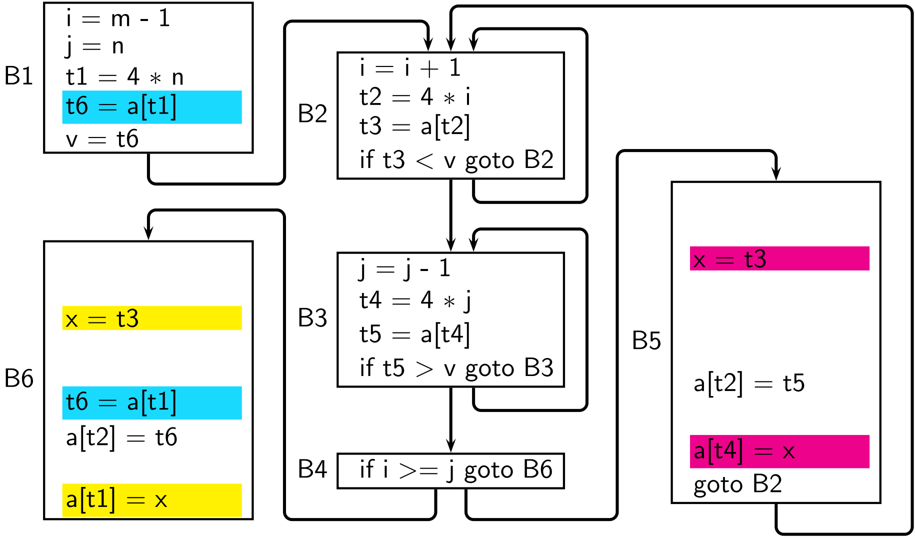
B6

```
x = t3
t6 = a[t1]
a[t2] = t6
a[t1] = x
```

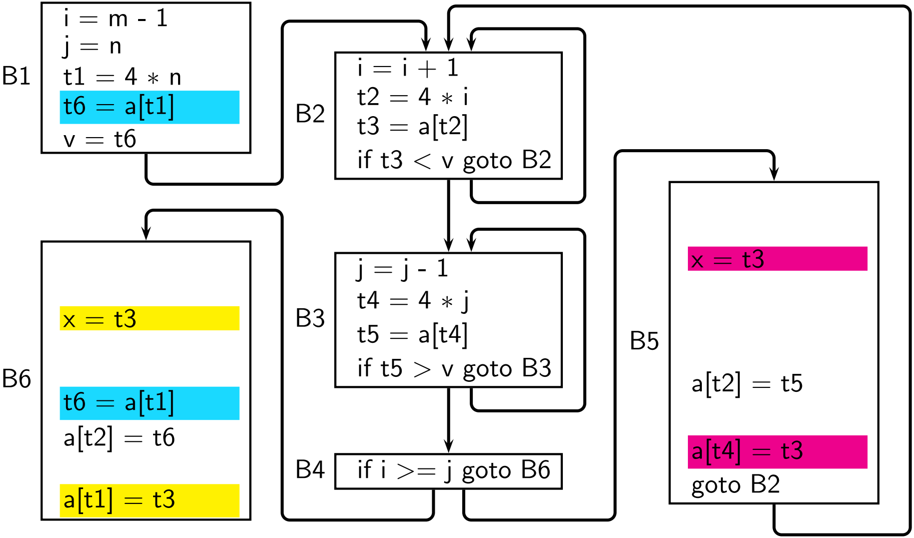
Other Classical Optimizations

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

Copy Propagation and Dead Code Elimination



Copy Propagation and Dead Code Elimination



B1

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

B2

```
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
```

B3

```
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
```

B4

```
if i >= j goto B6
```

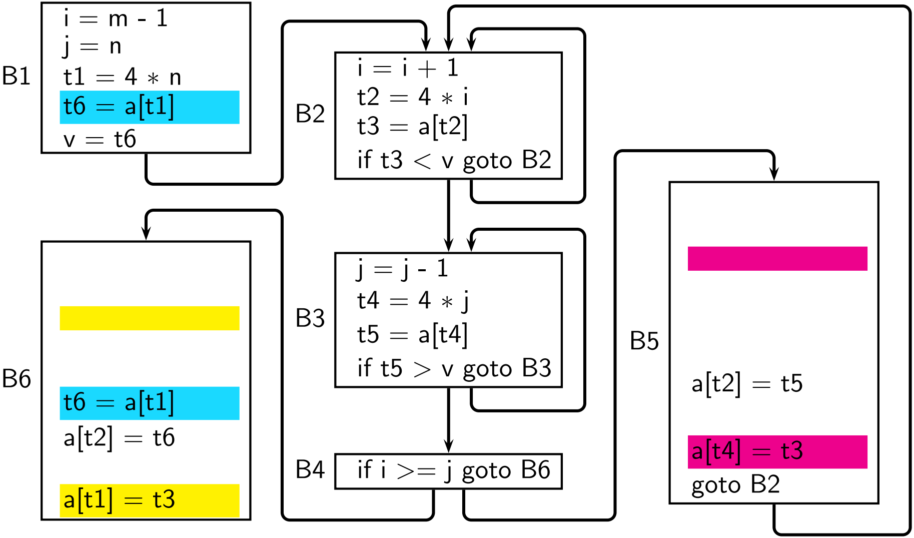
B5

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

B6

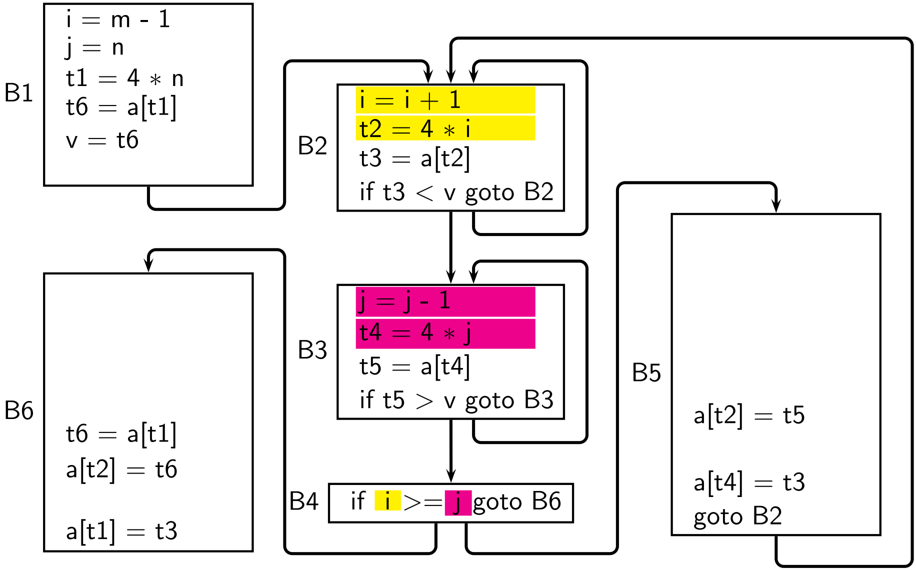
```
x = t3
t6 = a[t1]
a[t2] = t6
a[t1] = t3
```

Copy Propagation and Dead Code Elimination

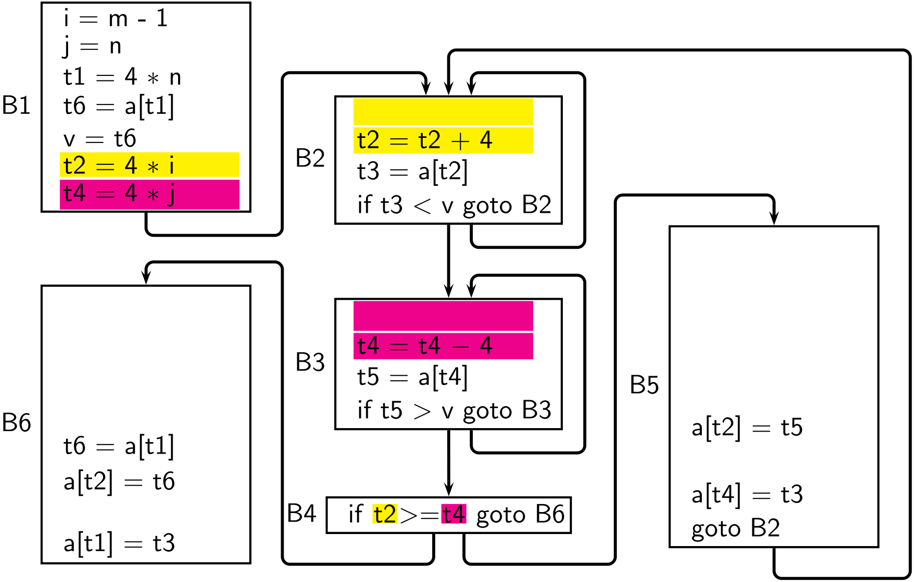


```
B1
i = m - 1
j = n
t1 = 4 * n
t6 = a[t1]
v = t6

B2
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2

B3
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3

B4
if i >= j goto B6

B5
a[t2] = t5
a[t4] = t3
goto B2

B6
t6 = a[t1]
a[t2] = t6
a[t1] = t3
```

Strength Reduction and Induction Variable Elimination



Strength Reduction and Induction Variable Elimination



B1
```
i = m - 1
j = n
t1 = 4 * n
t6 = a[t1]
v = t6
t2 = 4 * i
t4 = 4 * j
```

B2
```
t2 = t2 + 4
t3 = a[t2]
if t3 < v goto B2
```

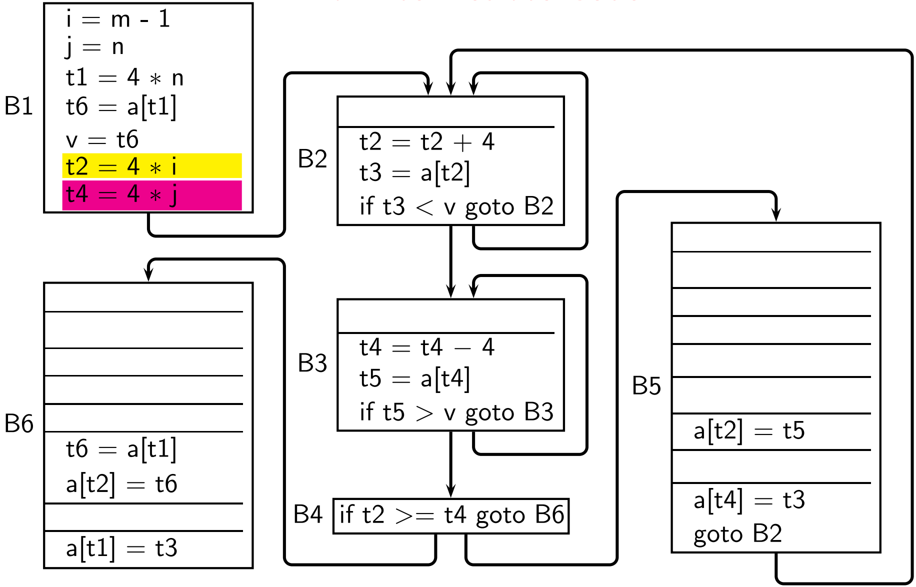
B3
```
t4 = t4 − 4
t5 = a[t4]
if t5 > v goto B3
```

B4
```
if t2>=t4 goto B6
```

B5
```
a[t2] = t5
a[t4] = t3
goto B2
```

B6
```
t6 = a[t1]
a[t2] = t6
a[t1] = t3
```

Final Intermediate Code



B1
```
i = m - 1
j = n
t1 = 4 * n
t6 = a[t1]
v = t6
t2 = 4 * i
t4 = 4 * j
```

B2
```
t2 = t2 + 4
t3 = a[t2]
if t3 < v goto B2
```

B3
```
t4 = t4 - 4
t5 = a[t4]
if t5 > v goto B3
```

B4
```
if t2 >= t4 goto B6
```

B5
```
a[t2] = t5
a[t4] = t3
goto B2
```

B6
```
t6 = a[t1]
a[t2] = t6
a[t1] = t3
```

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.

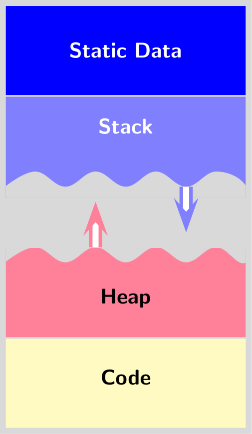
Categories of Optimizing Transformations and Analyses

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)

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 ≡ 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.

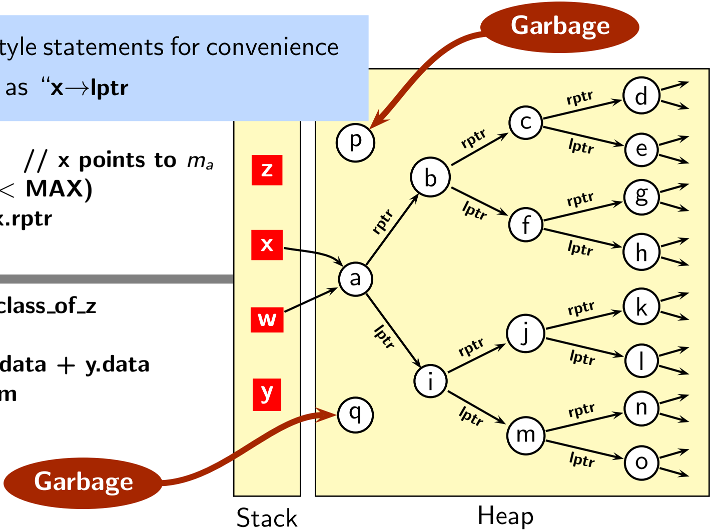
What if an object has an access path, but is not accessed after the given program point?

What is Garbage?

We use Java style statements for convenience

Read "**x.lptr**" as "**x→lptr**"

```
w = x          // x points to $m_a$
if  (x.data < MAX)
        x = x.rptr
y = x.lptr

z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```

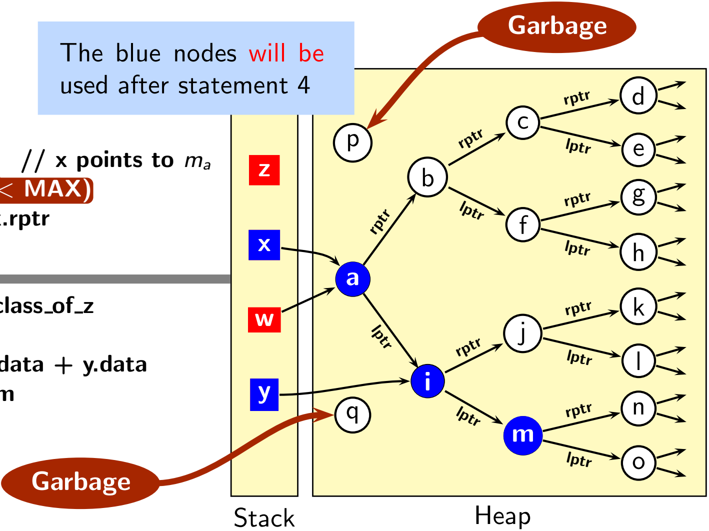


What is Garbage?

False

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
8   return z.sum
```

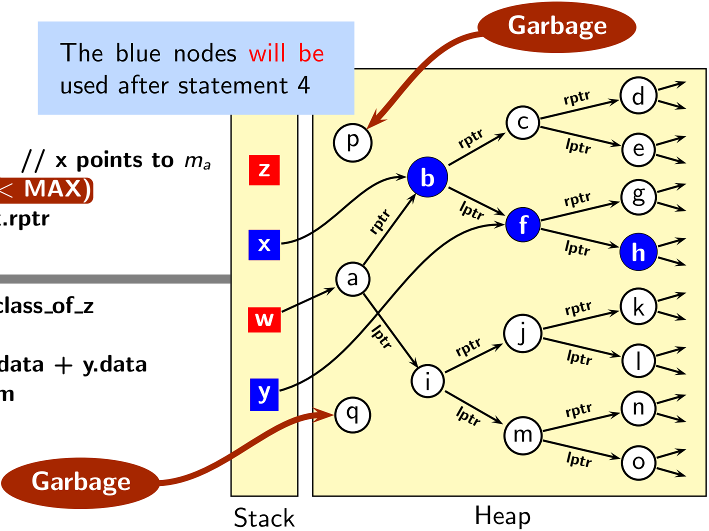


What is Garbage?

True

The blue nodes will be used after statement 4

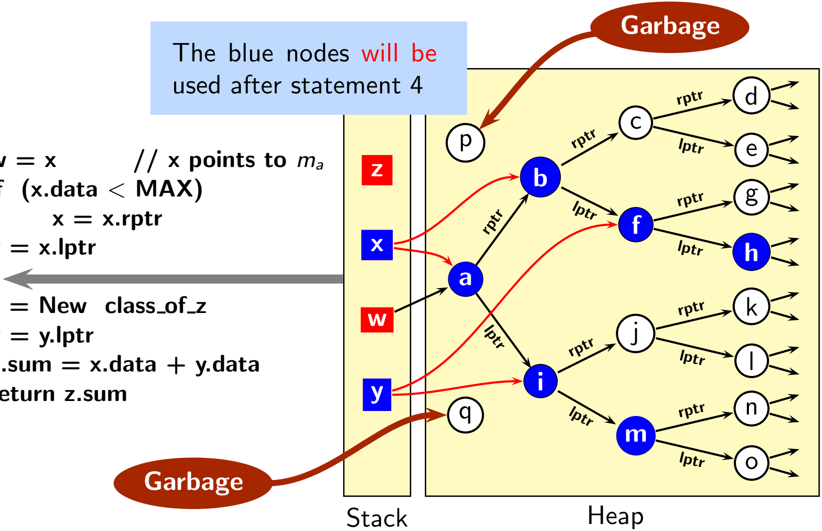
```
1    w = x              // x points to $m_a$
2    if  (x.data < MAX)
3            x = x.rptr
4    y = x.lptr

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



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
8   return z.sum
```



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

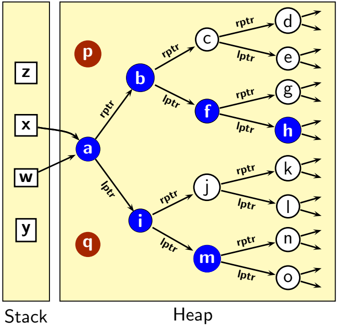
- The state of art compilers, virtual machines, garbage collectors cannot distinguish between the two

Reachability and Liveness

Some unused memory remains unclaimed because garbage collectors collect unreachable memory and not unused (i.e. non-live) memory

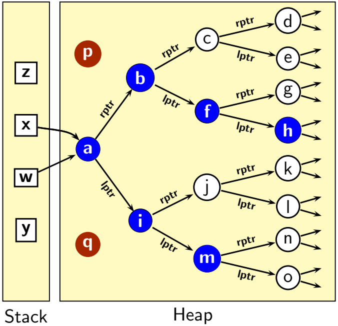
For the heap memory on the right

Allocated	White + Blue + Brown nodes
Reachable	White + Blue nodes
Live	Blue nodes



Reachability and Liveness

Some unused memory remains unclaimed because garbage collectors collect unreachable memory and not unused (i.e. non-live) memory



For the heap memory on the right

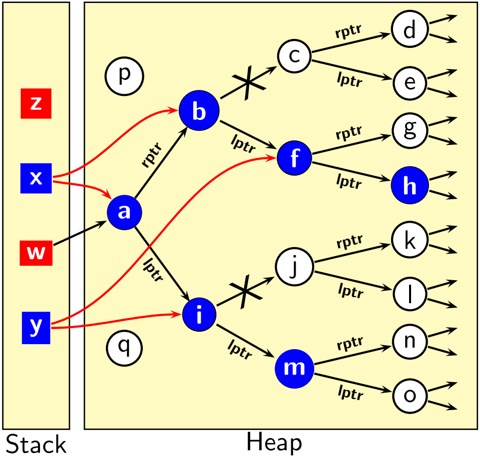
Allocated	White + Blue + Brown nodes
Reachable	White + Blue nodes
Live	Blue nodes

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

$$\text{Hence,} \quad \neg\text{Live} \supseteq \neg\text{Reachable} \supseteq \neg\text{Allocated}$$

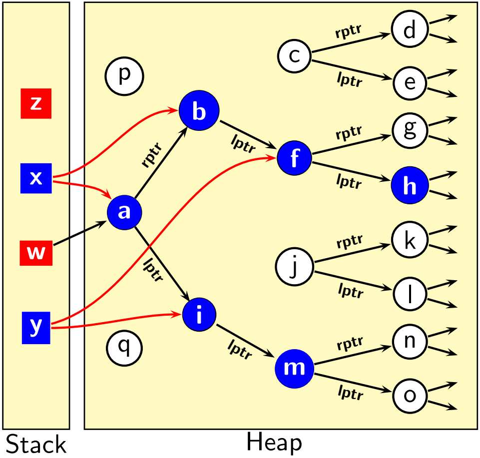
Cedar Mesa Folk Wisdom

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?

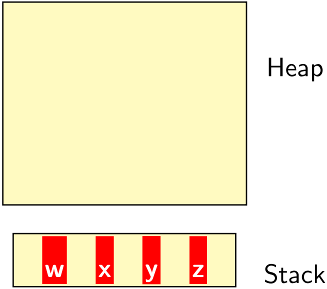
Liveness of Stack Data: An Informal Introduction (1)

We use Java style statements for convenience

Read “**x.lptr**” as “**x→lptr**

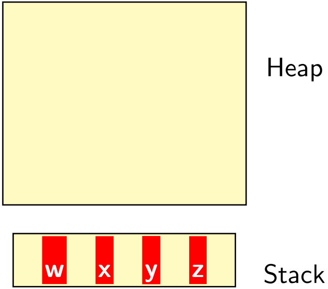
```
w = x          // x points to $m_a$
while  (x.data < MAX)
        x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```

if changed to while



Liveness of Stack Data: An Informal Introduction (1)

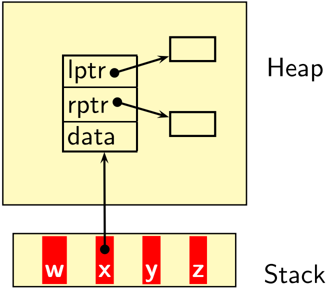
```
w = x              // x points to $m_a$
while (x.data < MAX)
        x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```



What is the meaning of the use of data?

Liveness of Stack Data: An Informal Introduction (1)

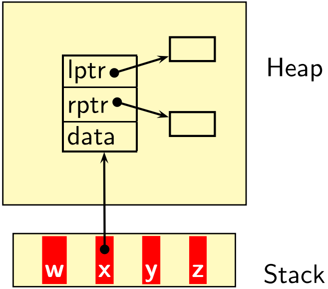
```
1   w = x            // x points to $m_a$
2   while  (x.data < MAX)
3            x = x.rptr
4   y = x.lptr
5   z = New  class_of_z
6   y = y.lptr
7   z.sum = x.data + y.data
8   return z.sum
```



What is the meaning of the use of data?

Liveness of Stack Data: An Informal Introduction (1)

```
w = x          // x points to $m_a$
while  (x.data < MAX)
        x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```

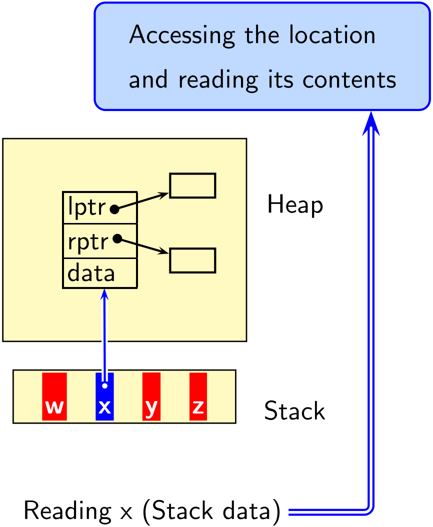


Accessing the location and reading its contents

What is the meaning of the use of data?

Liveness of Stack Data: An Informal Introduction (1)

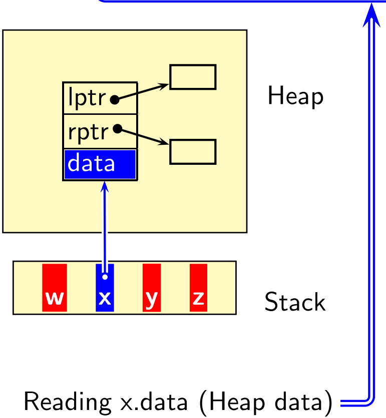
```
w = x          // x points to $m_a$
while  (x.data < MAX)
        x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```



Liveness of Stack Data: An Informal Introduction (1)

```
w = x          // x points to $m_a$
while  (x.data < MAX)
        x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```

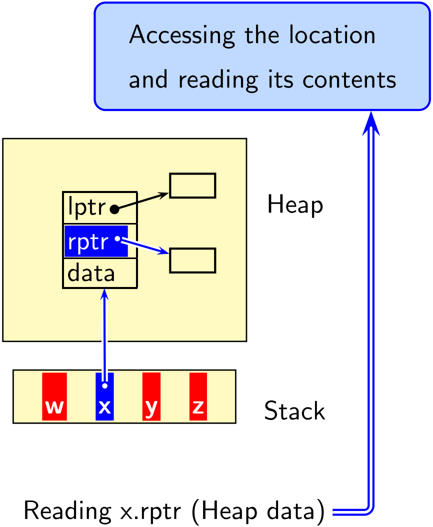
Accessing the location and reading its contents



Reading x.data (Heap data)

Liveness of Stack Data: An Informal Introduction (1)

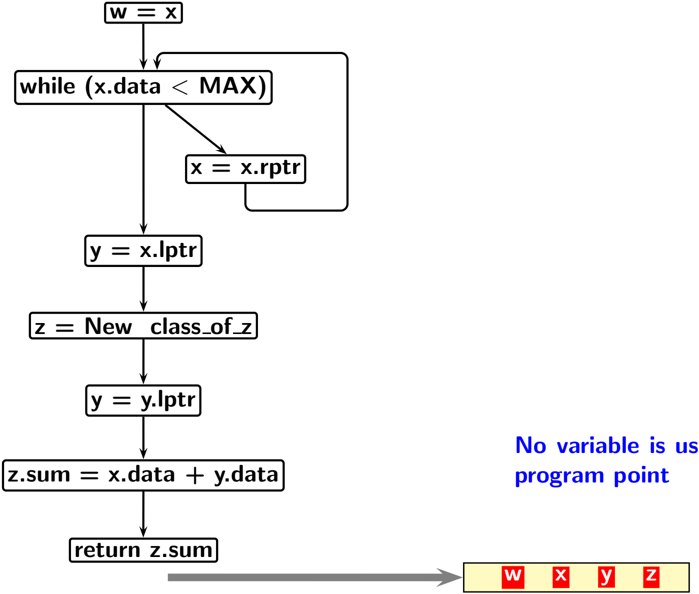
```
w = x          // x points to $m_a$
while  (x.data < MAX)
        x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```



Accessing the location and reading its contents

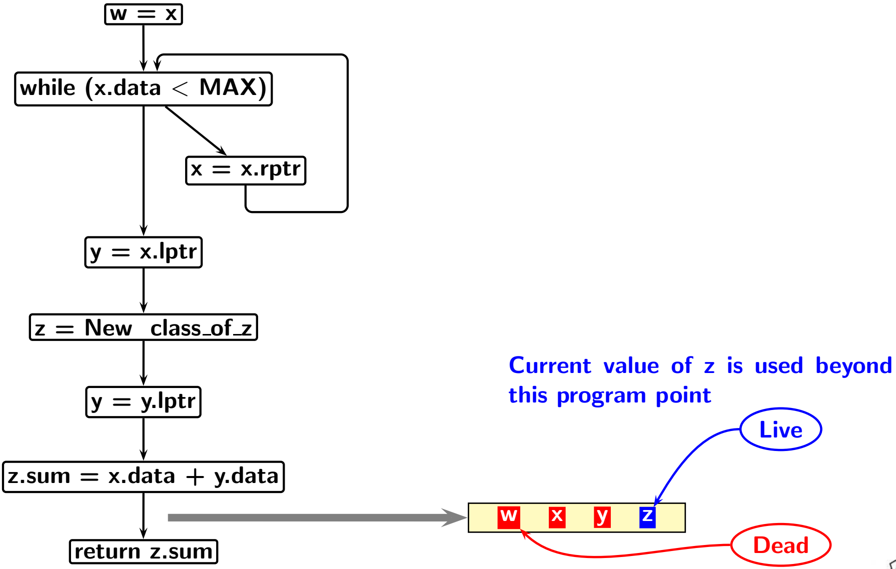
Reading x.rptr (Heap data)

Liveness of Stack Data: An Informal Introduction (2)



No variable is used beyond this program point

Liveness of Stack Data: An Informal Introduction (2)



w = x

while (x.data < MAX)

x = x.rptr

y = x.lptr

z = New class_of_z

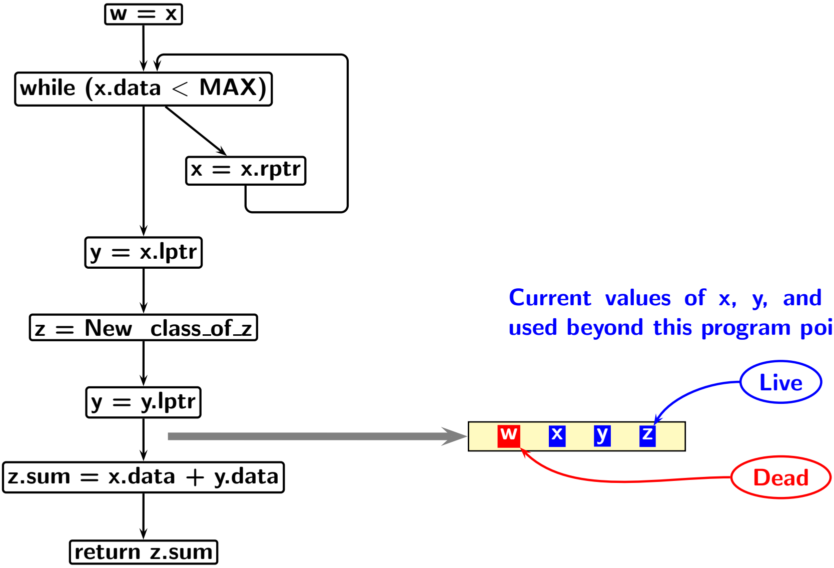
y = y.lptr

z.sum = x.data + y.data

return z.sum

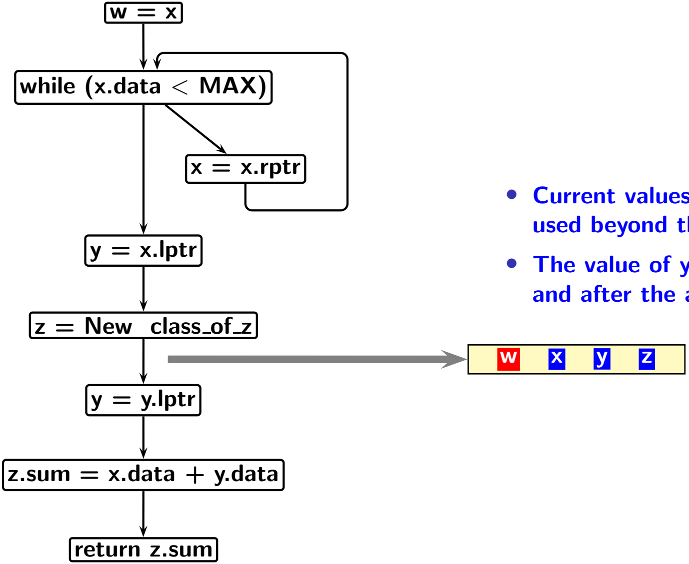
Current value of z is used beyond this program point

Liveness of Stack Data: An Informal Introduction (2)



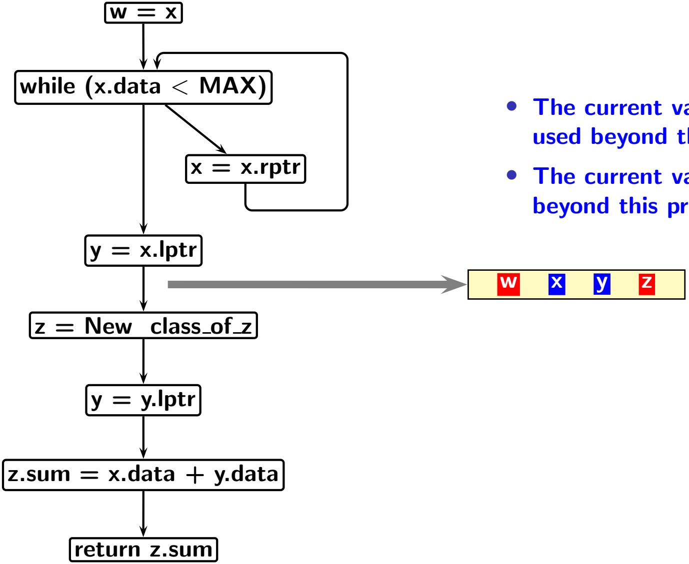
Current values of x, y, and z are used beyond this program point

Liveness of Stack Data: An Informal Introduction (2)



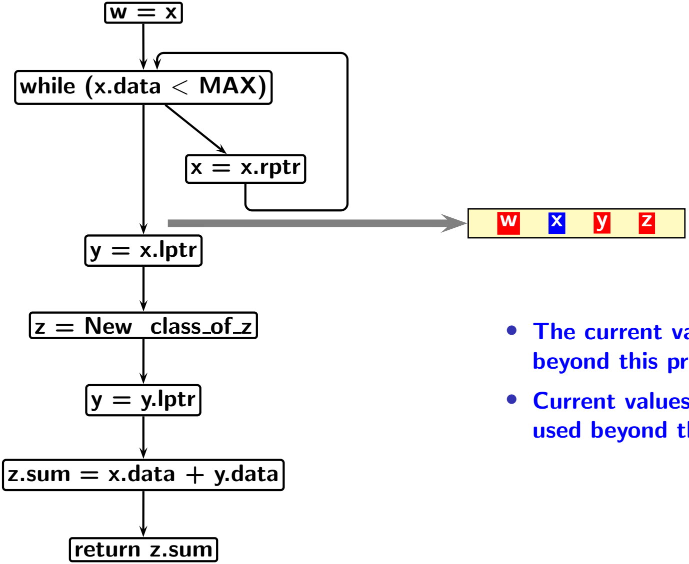
- Current values of x, y, and z are used beyond this program point
- The value of y is different before and after the assignment to y

Liveness of Stack Data: An Informal Introduction (2)



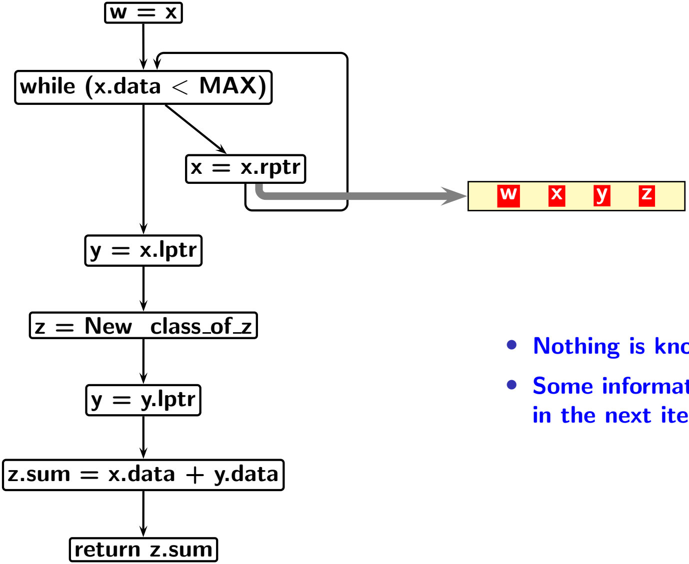
- The current values of x and y are used beyond this program point
- The current value of z is not used beyond this program point

Liveness of Stack Data: An Informal Introduction (2)



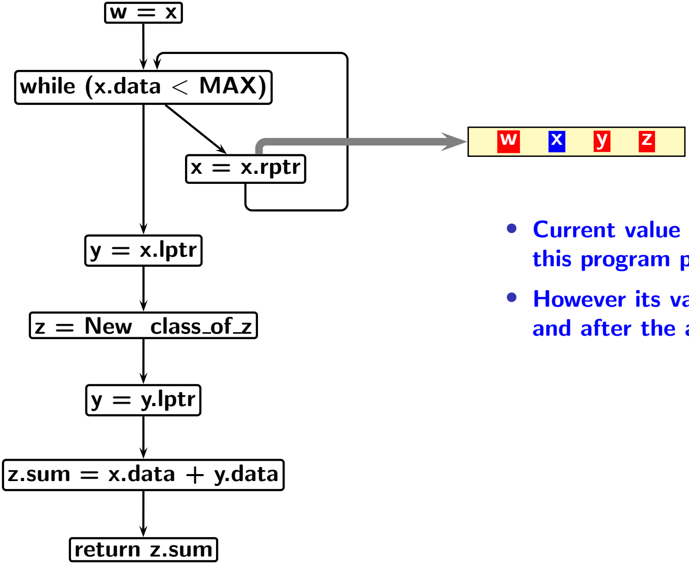
- The current values of x is used beyond this program point
- Current values of y and z are not used beyond this program point

Liveness of Stack Data: An Informal Introduction (2)



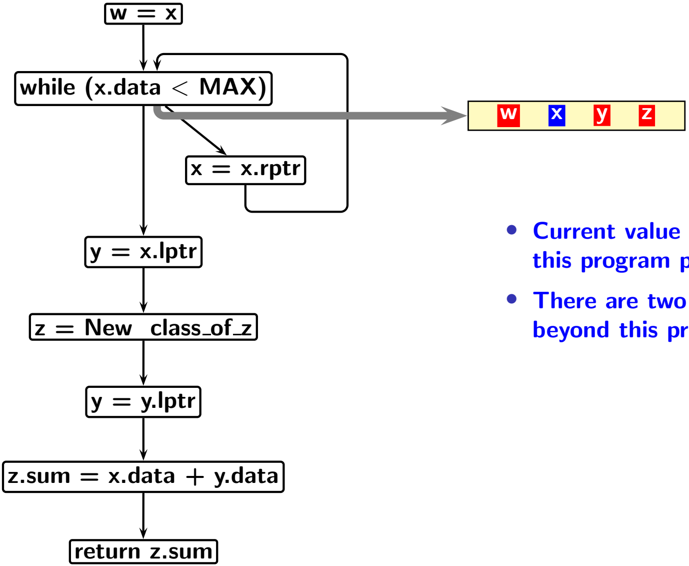
- Nothing is known as of now
- Some information will be available in the next iteration point

Liveness of Stack Data: An Informal Introduction (2)



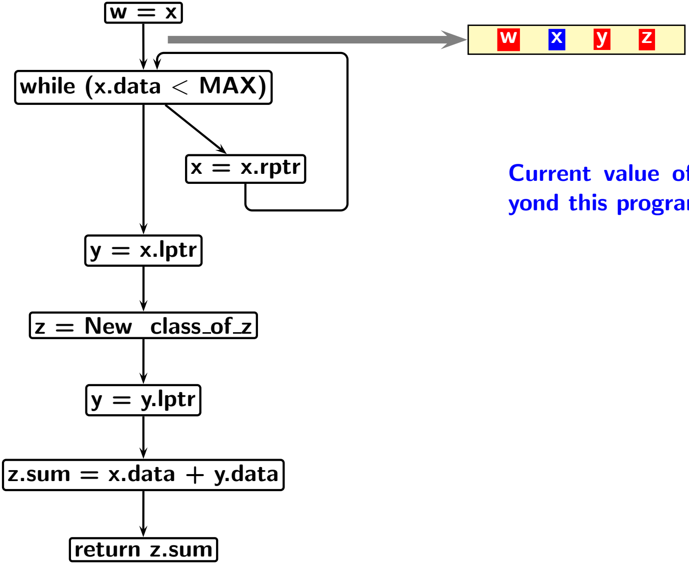
- **Current value of x is used beyond this program point**
- **However its value is different before and after the assignment**

Liveness of Stack Data: An Informal Introduction (2)



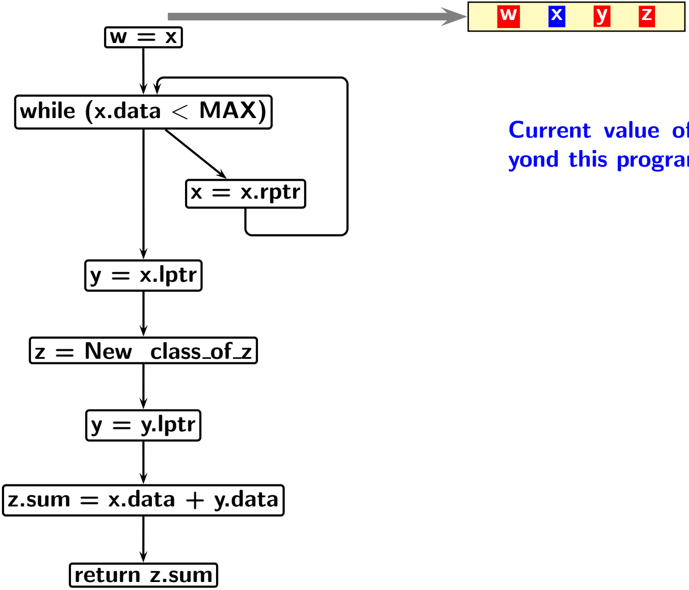
- Current value of x is used beyond this program point
- There are two control flow paths beyond this program point

Liveness of Stack Data: An Informal Introduction (2)



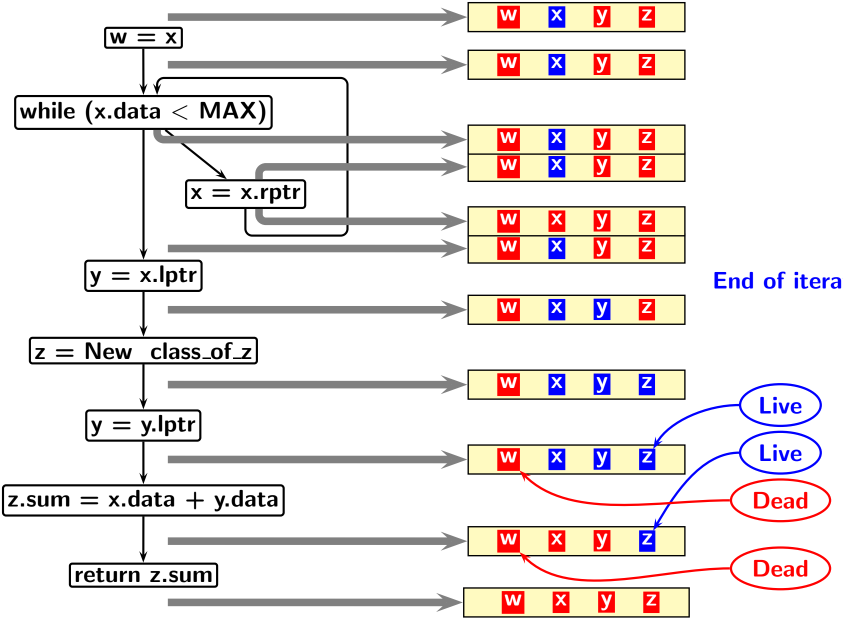
Current value of x is used beyond this program point

Liveness of Stack Data: An Informal Introduction (2)



Current value of x is used beyond this program point

Liveness of Stack Data: An Informal Introduction (2)



w = x

while (x.data < MAX)

x = x.rptr

y = x.lptr

z = New class_of_z

y = y.lptr

z.sum = x.data + y.data

return z.sum

End of iteration #1

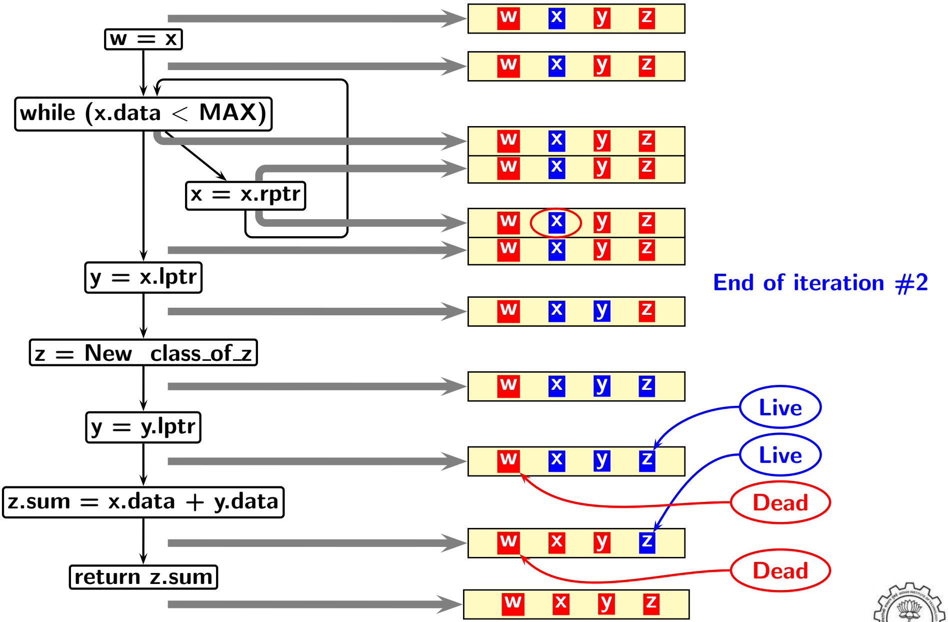
Live

Live

Dead

Dead

Liveness of Stack Data: An Informal Introduction (2)



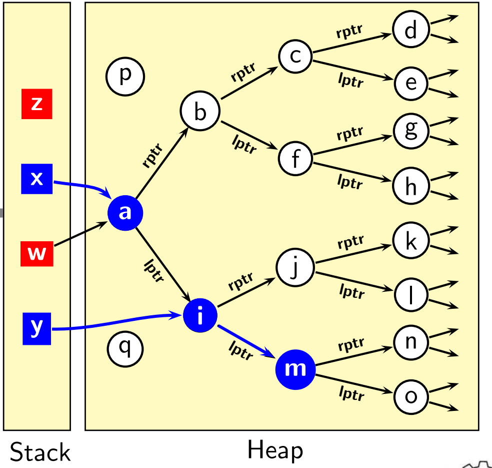
End of iteration #2

Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the while loop is not executed even once.

```
w = x            // x points to $m_a$
while  (x.data < MAX)
        x = x.rptr
y = x.lptr

z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```

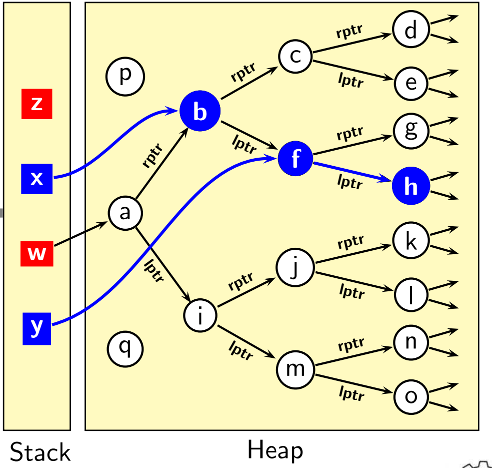


Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the while loop is executed once.

```
w = x            // x points to $m_a$
while  (x.data < MAX)
        x = x.rptr
y = x.lptr

z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```

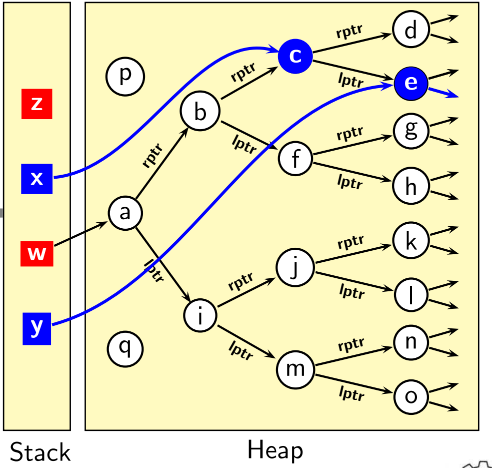


Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the while loop is executed twice.

```
w = x              // x points to $m_a$
while  (x.data < MAX)
        x = x.rptr
y = x.lptr

z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
return z.sum
```



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?

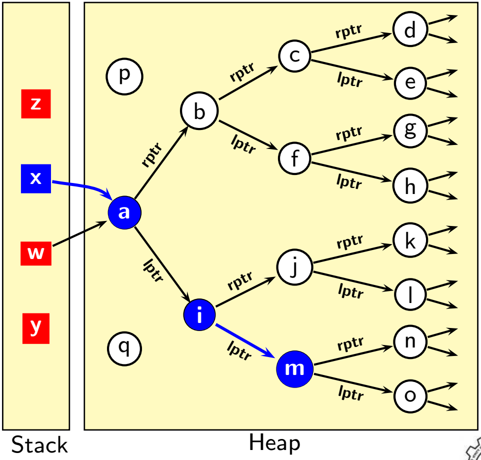
Our Solution (1)

```
                              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 = 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 = null
8  return z.sum
                              z = null
```

Our Solution (2)

```
   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 = null
8  return z.sum
   z = null
```

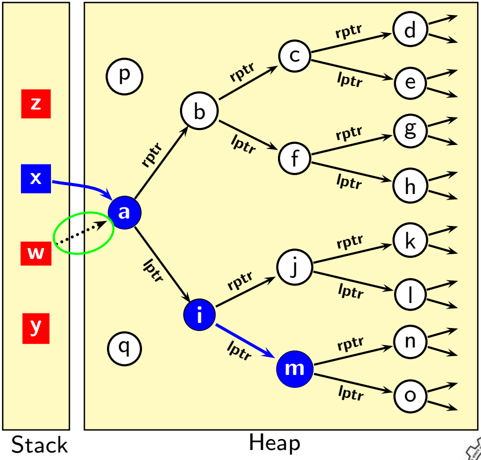
While loop is not executed even once



Our Solution (2)

```
  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 = null
8 return z.sum
  z = null
```

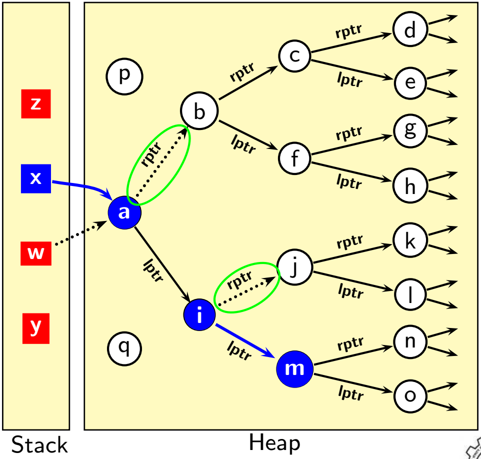
While loop is not executed even once



Our Solution (2)

```
  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 = null
8 return z.sum
  z = null
```

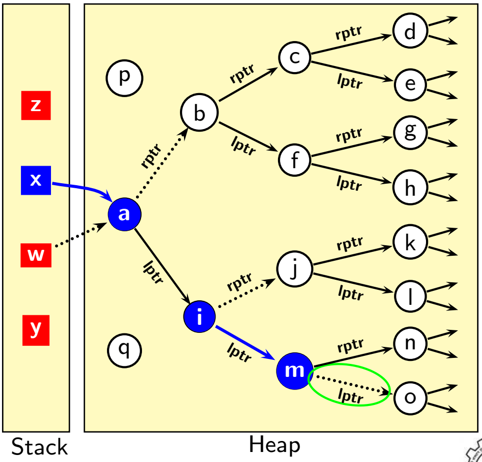
While loop is not executed even once



Our Solution (2)

```
     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 = null
8    return z.sum
     z = null
```

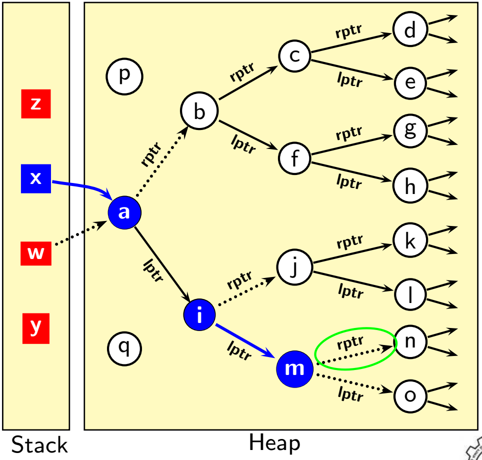
While loop is not executed even once



Our Solution (2)

```
  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 = null
8 return z.sum
  z = null
```

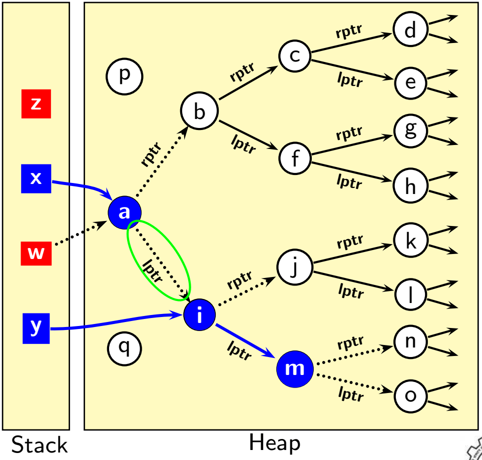
While loop is not executed even once



Our Solution (2)

```
     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 = null
8    return z.sum
     z = null
```

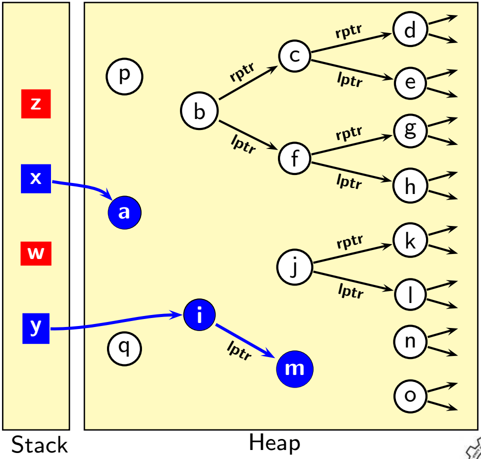
While loop is not executed even once



Our Solution (2)

```
y = z = null
w = x
w = null
while (x.data < MAX)
{       x.lptr = null
        x = x.rptr       }
x.rptr = x.lptr.rptr = null
x.lptr.lptr.lptr = null
x.lptr.lptr.rptr = null
y = x.lptr
x.lptr = y.rptr = null
y.lptr.lptr = y.lptr.rptr = null
z = New class_of_z
z.lptr = z.rptr = null
y = y.lptr
y.lptr = y.rptr = null
z.sum = x.data + y.data
x = y = null
return z.sum
z = null
```

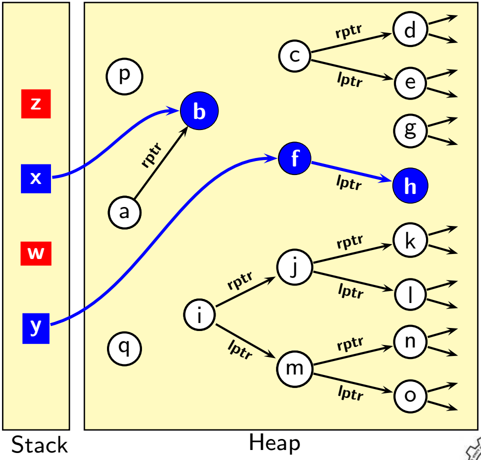
While loop is not executed even once



Our Solution (2)

```
    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 = null
8   return z.sum
    z = null
```

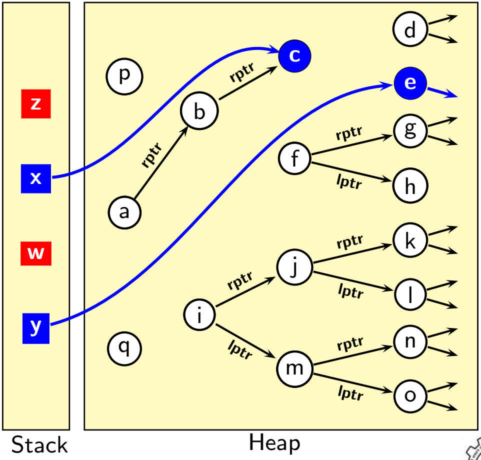
While loop is executed once



Our Solution (2)

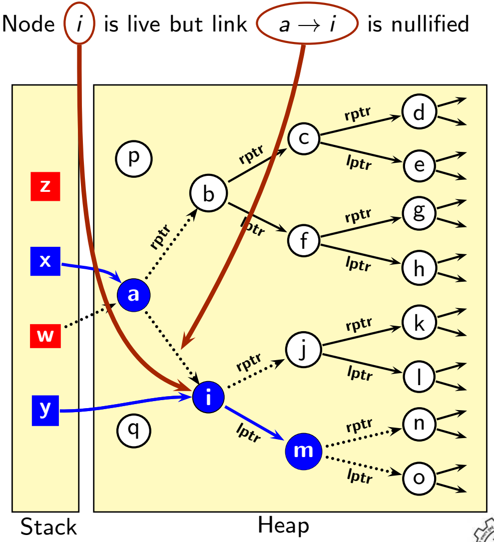
```
y = z = null
w = x
w = null
while (x.data < MAX)
{       x.lptr = null
        x = x.rptr        }
x.rptr = x.lptr.rptr = null
x.lptr.lptr.lptr = null
x.lptr.lptr.rptr = null
y = x.lptr
x.lptr = y.rptr = null
y.lptr.lptr = y.lptr.rptr = null
z = New class_of_z
z.lptr = z.rptr = null
y = y.lptr
y.lptr = y.rptr = null
z.sum = x.data + y.data
x = y = null
return z.sum
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 = null
8  return z.sum
   z = null
```

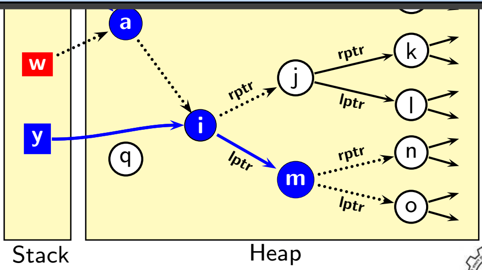


Node i is live but link $a \to i$ is nullified

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 = null
8   return z.sum
    z = null
```

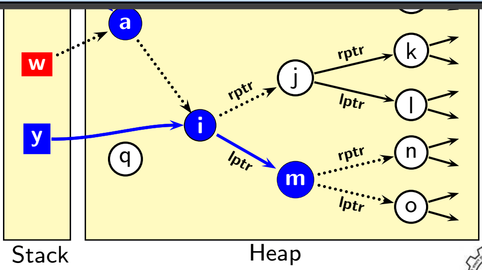
- The memory address that *x* holds when the execution reaches a given program point is not an invariant of program execution



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 = null
8    return z.sum
     z = null
```

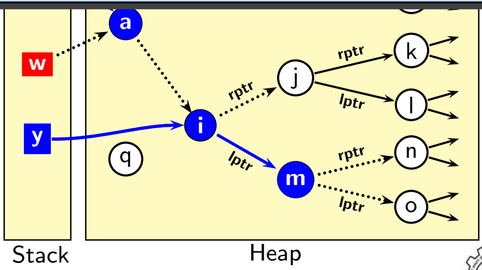
- The memory address that x holds when the execution reaches a given program point is not an invariant of program execution
- Whether we dereference lptr out of x or rptr out of x at a given program point is an invariant of program execution



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 = null
8 return z.sum
  z = null
```

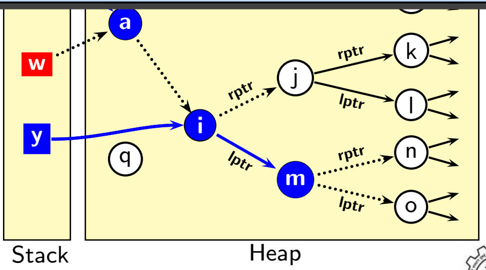
- The memory address that *x* holds when the execution reaches a given program point is not an invariant of program execution
- Whether we dereference lptr out of *x* or rptr out of *x* at a given program point is an invariant of program execution
- *A static analysis can discover only invariants*



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 = null
8   return z.sum
    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*

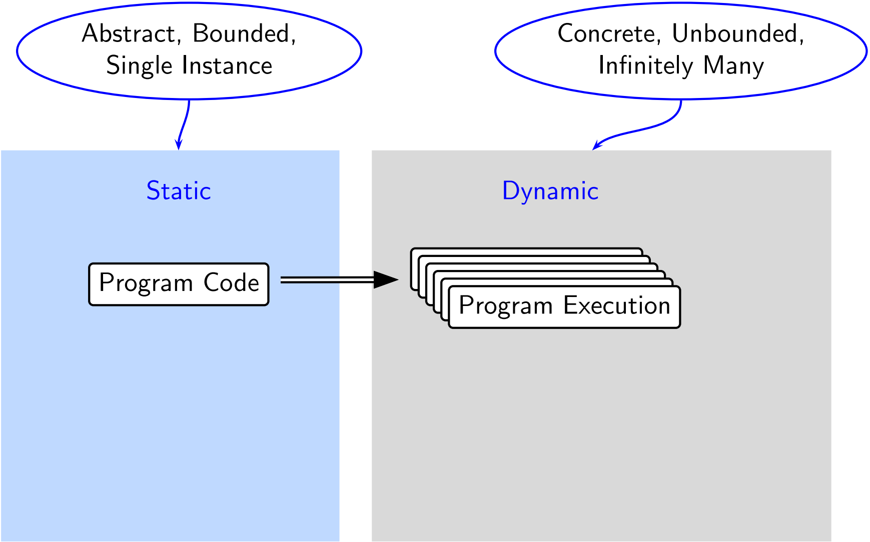


BTW, What is Static Analysis of Heap?

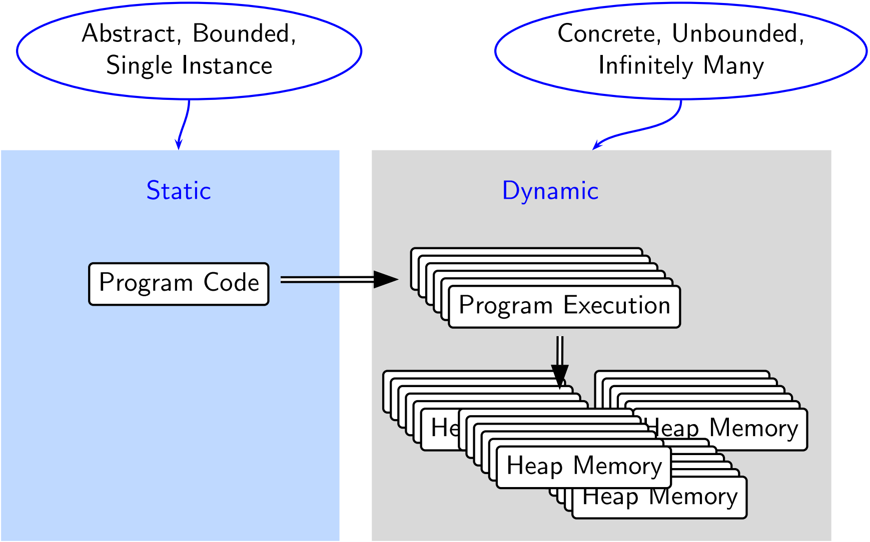
Static

Dynamic

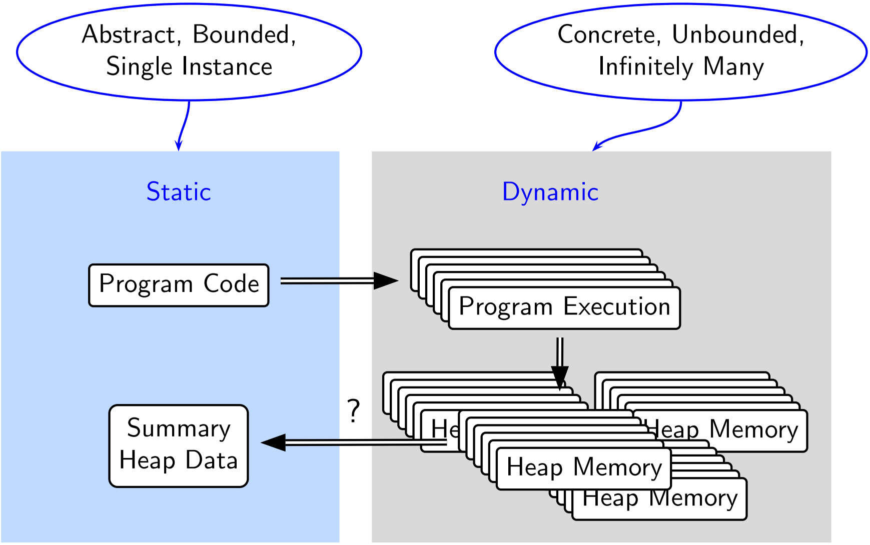
BTW, What is Static Analysis of Heap?



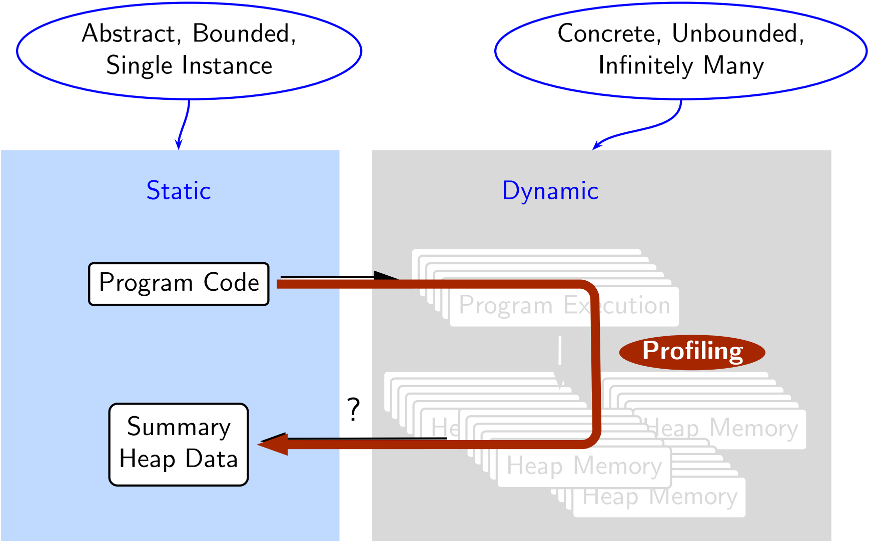
BTW, What is Static Analysis of Heap?



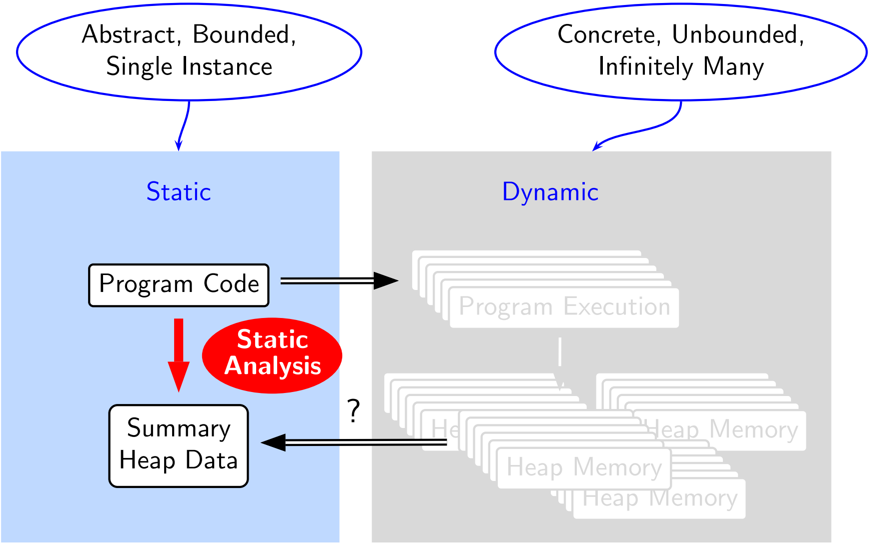
BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



Abstract, Bounded,
Single Instance

Concrete, Unbounded,
Infinitely Many

Static

Dynamic

Program Code

Program Execution

Static Analysis

Summary
Heap Data

?

Heap Memory

Heap Memory

Heap Memory

Part 4

What is 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

Important Requirements of Static Analysis

- We discuss the following important requirements
    - Soundness
    - Precision
    - Efficiency
    - Scalability
- Soundness and precision are described more formally later in module 2

Inexactness of Static Analysis Results

- Static analysis predicts run time behaviour of programs
- Static analysis is undecidable

  there cannot exist an algorithm that can compute
  exact result for every program

- Possible reasons of undecidability
    - Values of variables not known
    - Branch outcomes not known
    - Infinitely many paths in the presence of loops or recursion
    - Infinitely many values
- Static analysis predictions may not match the actual run time behaviour

Possible Errors in Static Analysis Predictions

- Some predictions may be erroneous because the predicted behaviour
  - may not be found in some execution instances, or
  - may not be found in any execution instance

  (Error ≡ Mismatch between run time behaviour and predicted behaviour)
- Some of these errors may be harmless whereas some may be harmful
- Some of these errors may be unavoidable (recall undecidability)
- How do we characterize, identify, and minimize, these errors?

Examples of Harmless and Harmful Errors in Predictions (1)

- For security check at an airport,
    - Frisking a person more than others on mere suspicion may be an error but it is harmless from the view point of security
    - Not frisking a person much even after a suspicion is an error and it could be a harmful from the view point of security

- For stopping smuggling of contraband goods
    - Not checking every passenger may be erroneous but is harmless
    - Checking every passenger may be right but is harmful

- Weather prediction during rainy season
    - A doubtful prediction of "*heavy to very heavy rain*" is harmless
    - Not predicting "*heavy to very heavy rain*" could be harmful

Examples of Harmless and Harmful Errors in Predictions (2)

- For medical diagnosis
    - Subjecting a person to further investigations may be erroneous but in most cases it is harmless
    - Avoiding further investigations even after some suspicions could be harmful
- For establishing justice in criminal courts
    - Starting with the assumption that an accused is innocent may be erroneous but is harmless
    - Starting with the assumption that an accused is guilty may be harmful

Harmless Errors and Harmful Errors in Static Analysis

- For a static analysis,
    - Harmless errors can be tolerated but should be minimized **Precision**
    - Harmful errors MUST be avoided **Soundness**
- Some behaviours concluded by a static analysis are
    - uncertain and cannot be guaranteed to occur at run time,
      (This uncertainty is harmless and hence is conservative)
    - certain and can be guaranteed to occur at run time
      (The absence of this certainty for these behaviours may be harmful)

Examples of Conservative and Definite Information

- Liveness is uncertain (also called conservative)

  If a variable is declared live at a program point, it may or may not be used beyond that program point at run time

  (Why is it harmless if the variable is not actually used?)

- Deadness (i.e. absence of liveness) is certain (also called definite)

  If a variable is declared to be dead at a program point, it is guaranteed to be not used beyond that program point at run time

  (Why is it harmful if the variable is not actually dead?)

Efficiency and Scalability

- Efficiency
    - How well are resources used
    - Measured in terms of work done per unit resource
    - Resources: time, memory, power, energy, processors, network etc.
    - Example: Strike rate of a batter in cricket
- Scalability
    - How large inputs can be handled
    - Measured in terms of size of the input
    - Example: Total runs scored by a batter in cricket
- Efficiency and scalability are orthogonal
    - Efficiency does not necessarily imply scalability
    - Scalability does not necessarily imply efficiency

Efficiency and Scalability May be Unrelated

Examples of the combinations of efficiency and scalability from sorting algorithms

	Efficient	Inefficient
Scalable	Merge Sort	Selection Sort
Non-scalable	Quicksort	Bubble Sort

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 be sound

  Ex: Traffic police catching people for traffic violations
- 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

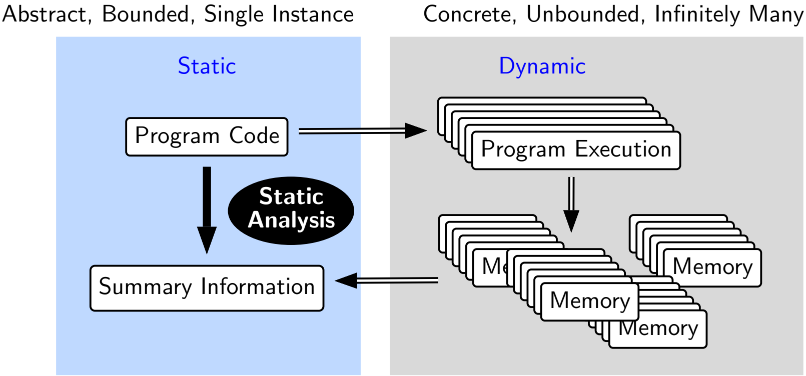
  Tolerate imprecision for complete soundness

The Goal of Program Analysis

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

The Goal of Program Analysis

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



Part 5

An Overview of the Tutorial

Topics Covered

- Live variables analysis (including strong liveness analysis)
- Constant propagation
- Pointer Analysis

We start with the basics but will reach research frontiers and discuss research being done at IIT Bombay

Pedagogy

Interleaving of interactive

- Lectures
- Demos
- Paper exercises
- Computer exercises using SPAN tool

Questions ??

Part 6

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
```