Introduction to Data Flow Analysis

Uday Khedker

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

July 2010

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.

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
- Course details, schedule, assessment policies etc.
- Program execution model and semantics

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


Intermediate Code : Observations

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

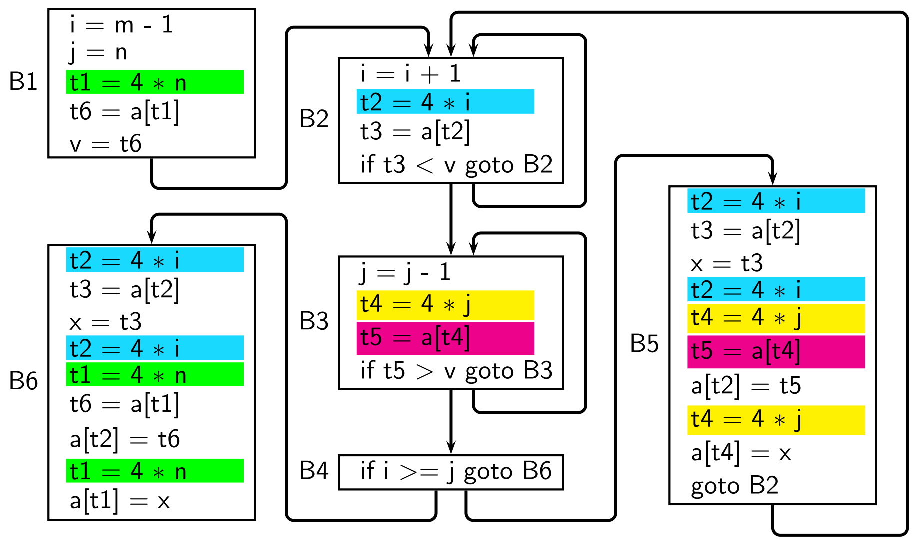
Understanding Control Flow

- 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
```

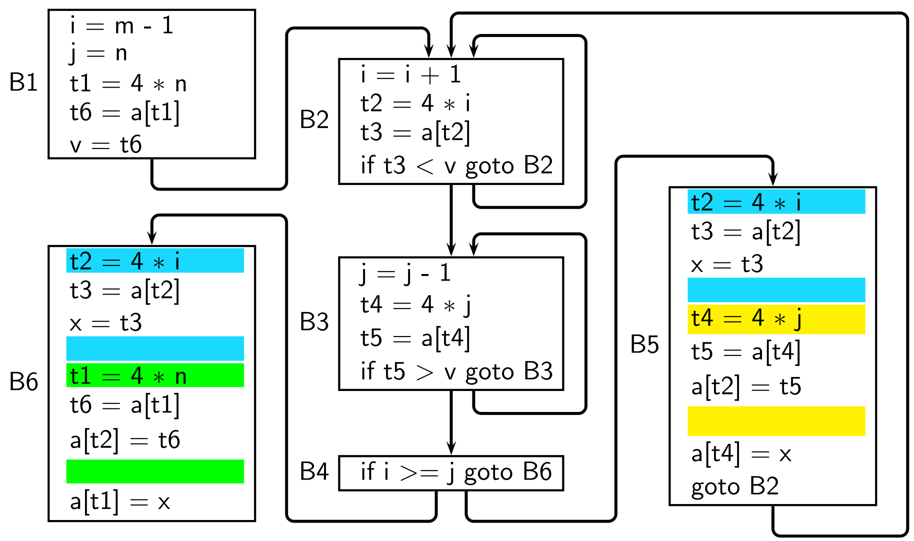

Program Flow Graph



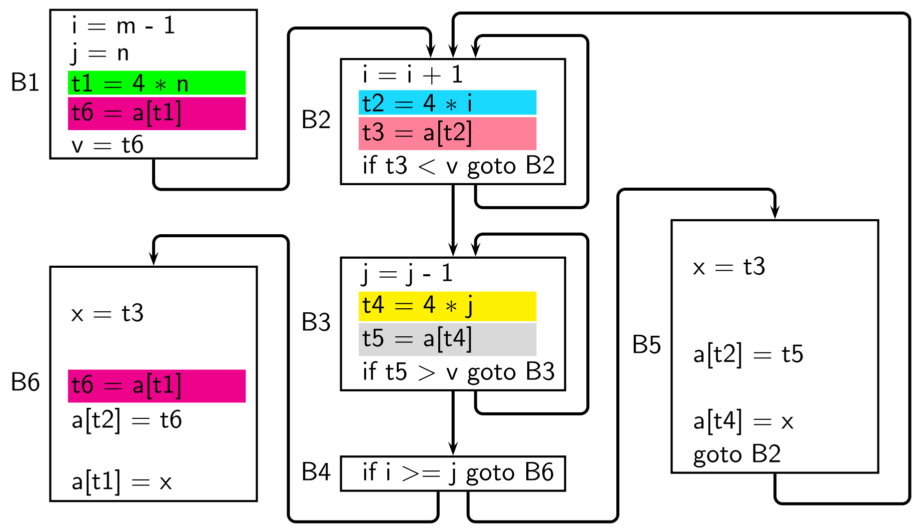
Program Flow Graph : Observations

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

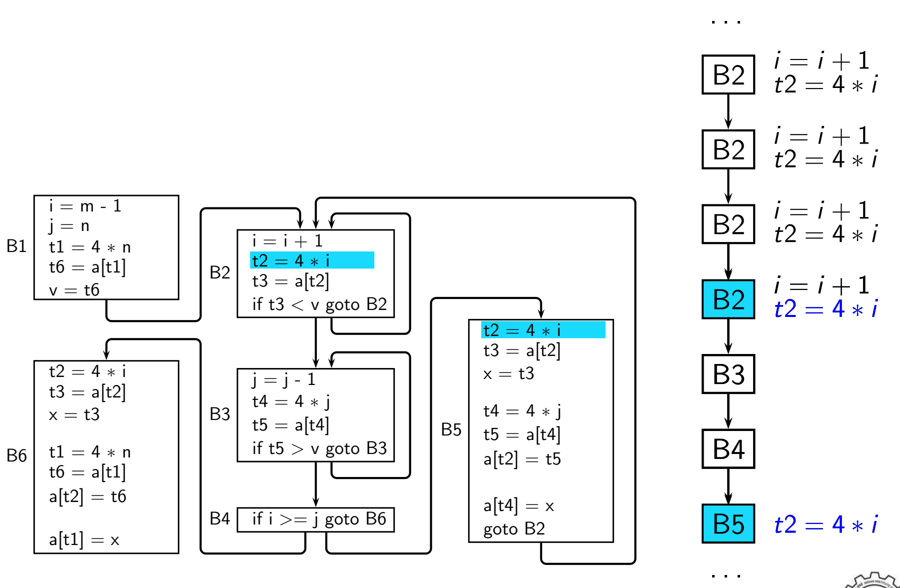
Local Common Subexpression Elimination



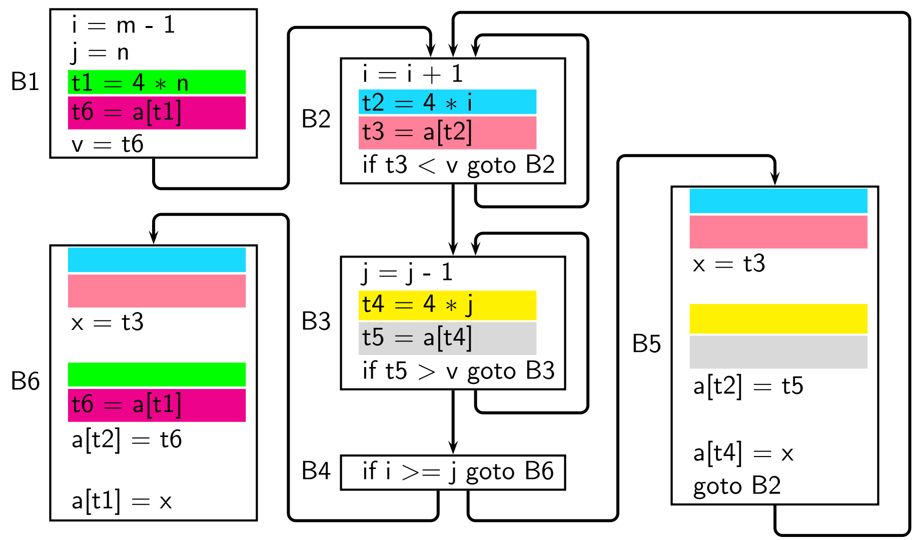
Global Common Subexpression Elimination



Global Common Subexpression Elimination



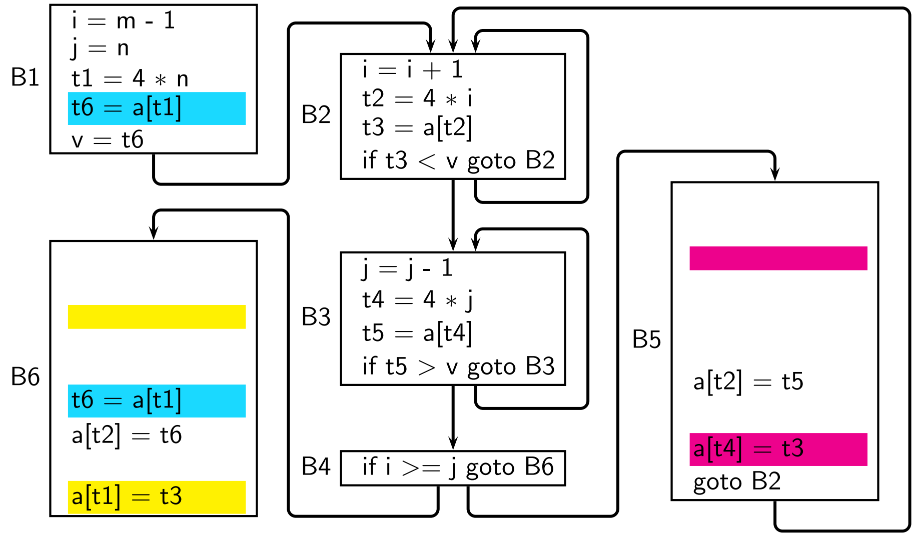
Global Common Subexpression Elimination



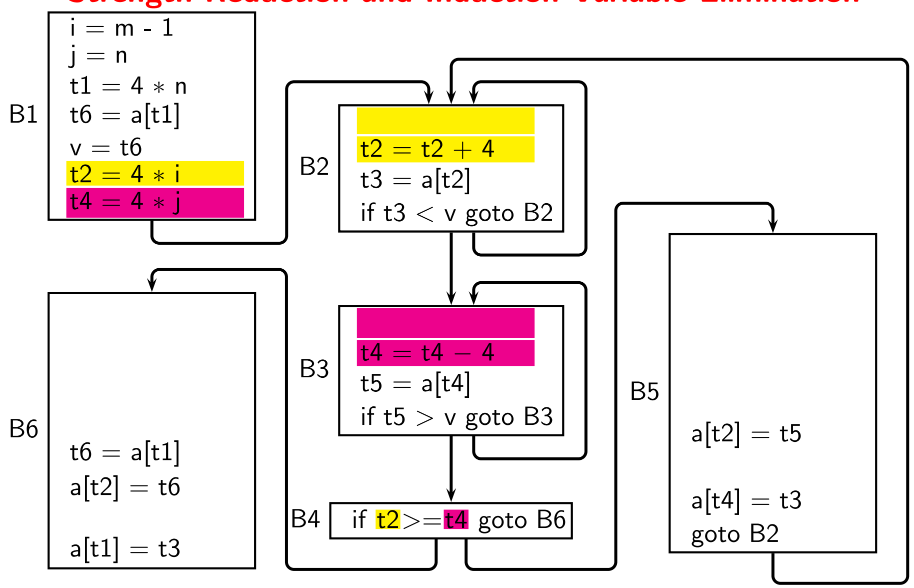
Other Classical Optimizations

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

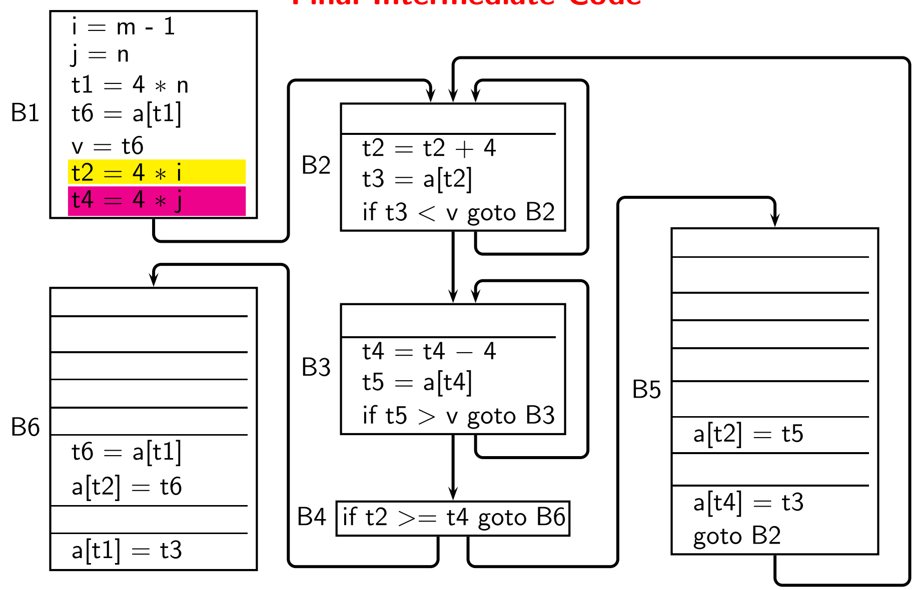
Copy Propagation and Dead Code Elimination



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)

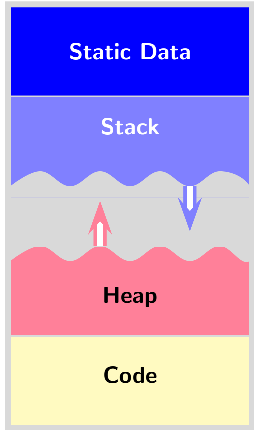
Conclusions

- Static analysis discovers useful information that represents all execution instances of the program being analysed
- This information can be used for a variety of applications such as
  - code optimization
  - verification and validation
  - reverse engineering
  - software engineering

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)

Decision 2: When to Deallocate?

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

Predictability of Lifetimes

Memory	Run Time Change		Lifetime
	Allocation	Deallocation	
Code	None	None	Predictable
Static Data	None	None	Predictable
Stack	Predictable	Predictable	Predictable
Heap	Predictable	Unpredictable	Unpredictable

Predictability $\Rightarrow$ Can be easily discovered by analysing the program text

The Importance of Predictability

- Stack and static data offer a reasonable predictability.
  - Good (static) analysis techniques.
  - Allocation/deallocation/optimization handled very well by production quality language processors.
- Heap memory does not offer the same predictability.
  - Few or no (static) production quality analysis techniques.
  - Optimization ?

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
    $\Rightarrow$ At which program point should a leak be "plugged"?

Garbage Collection $\equiv$ Automatic Deallocation

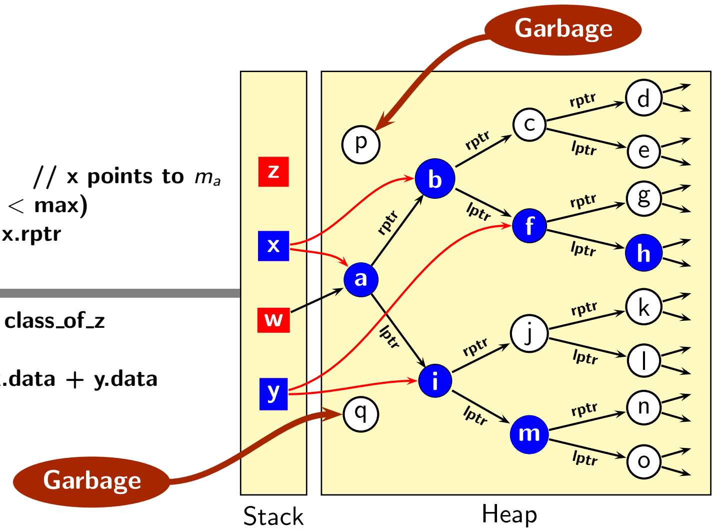
- Retain active data structure.
  Deallocate inactive data structure.
- What is an Active Data Structure?

  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?

```
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
```



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

- Yet we have no way of distinguishing.

  Over a dozen reported approaches (since 1996), no real success.

Reachability and Liveness

Comparison between different sets of objects:

Live ? Reachable ? Allocated

The objects that are not live must be reclaimed.

Reachability and Liveness

Comparison between different sets of objects:

Live $\subseteq$ Reachable $\subseteq$ 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:

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

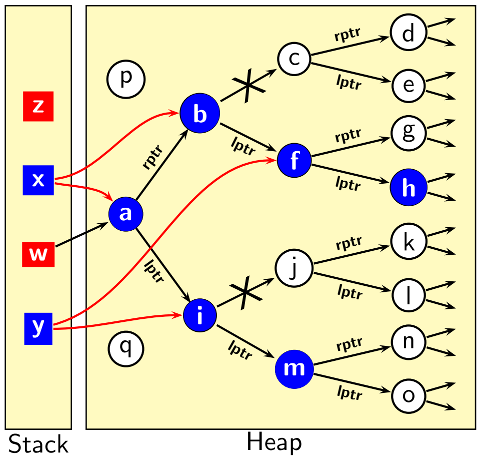
The objects that are not live must be reclaimed.

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

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

- 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.

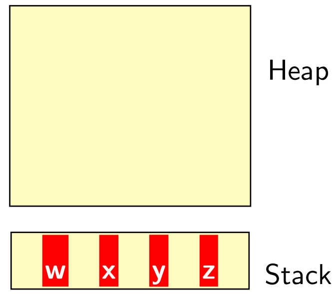
Distinguishing Between Reachable and Live

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

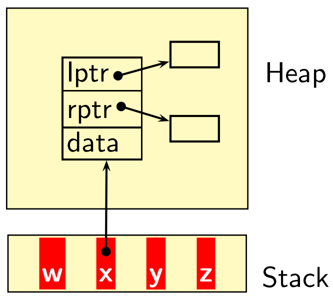
```
w = x          // x points to m_a
while  (x.data < max)
        x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
```



if changed to while

Liveness of Stack Data

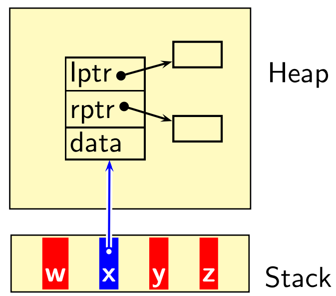
```
w = x          // x points to m_a
while  (x.data < max)
        x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
```



What is the meaning of use of data?

Liveness of Stack Data

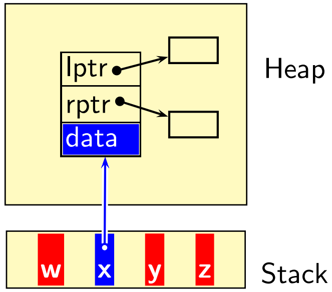
```
w = x          // x points to m_a
while  (x.data < max)
        x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
```



Reading x (Stack data)

Liveness of Stack Data

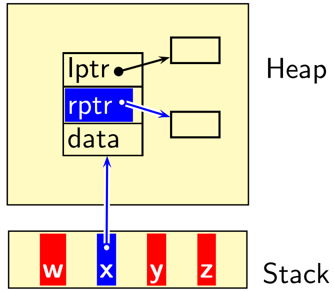
```
w = x          // x points to m_a
while (x.data < max)
       x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
```



Reading x.data (Heap data)

Liveness of Stack Data

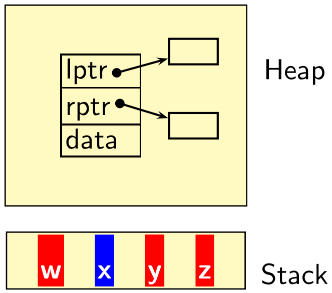
```
w = x          // x points to m_a
while (x.data < max)
       x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
```



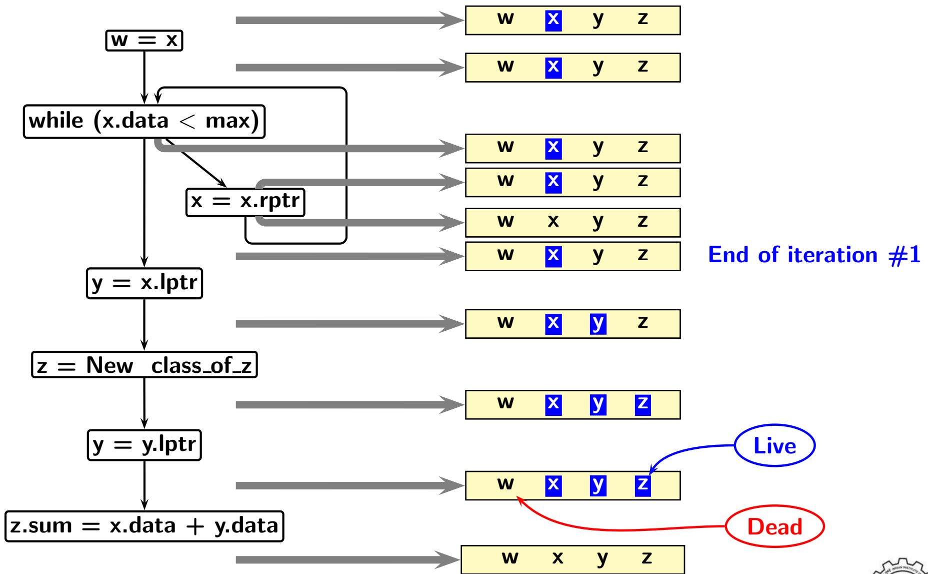
Reading x.rptr (Heap data)

Liveness of Stack Data

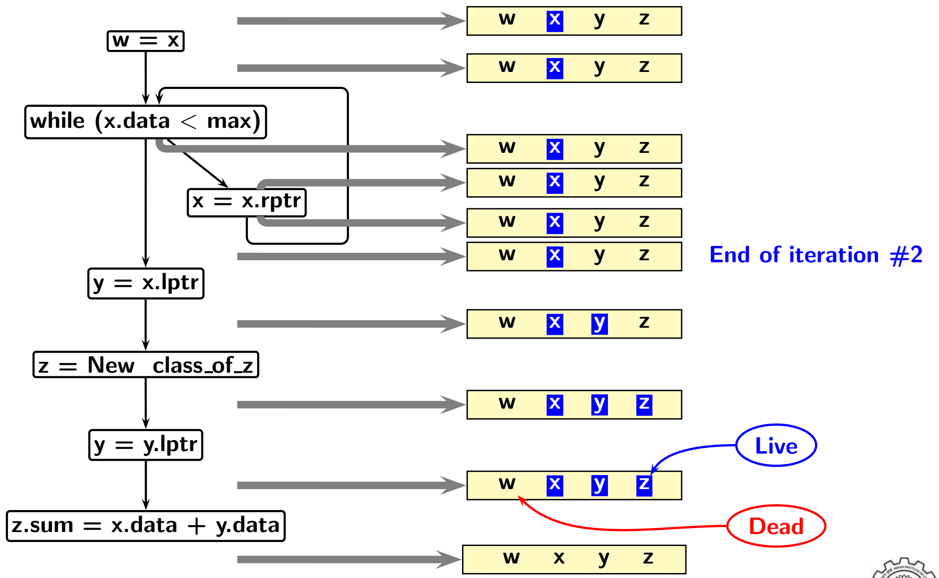
```
w = x          // x points to m_a
while (x.data < max)
       x = x.rptr
y = x.lptr
z = New  class_of_z
y = y.lptr
z.sum = x.data + y.data
```



Liveness of Stack Data



Liveness of Stack Data

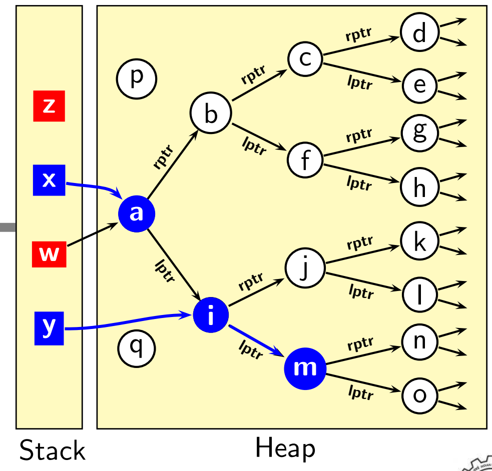


Applying Cedar Mesa Folk Wisdom to Heap Data

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
```

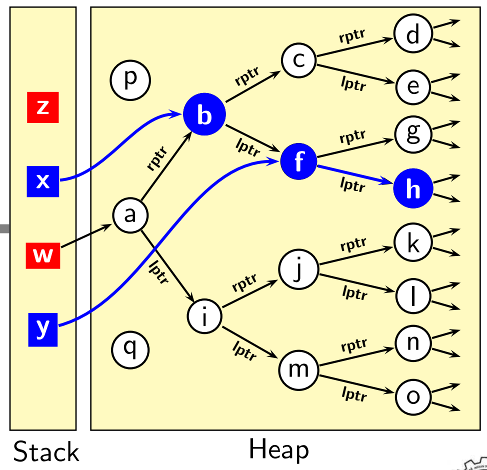


Applying Cedar Mesa Folk Wisdom to Heap Data

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
```

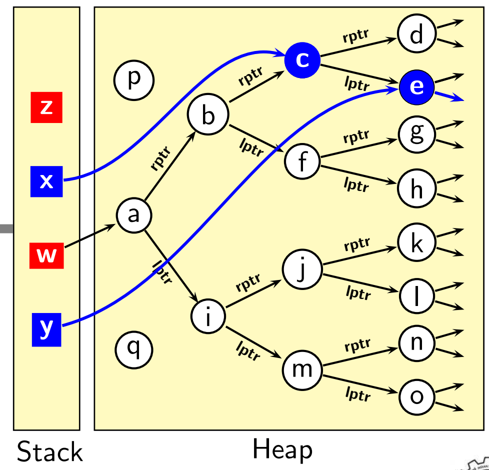


Applying Cedar Mesa Folk Wisdom to Heap Data

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
```



The Moral of the Story

- Mappings between access expressions and l-values keep changing
- This is a *rule* for heap data
  For stack and static data, it is an *exception*!
- Static analysis of programs has made significant progress for stack and static data.

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

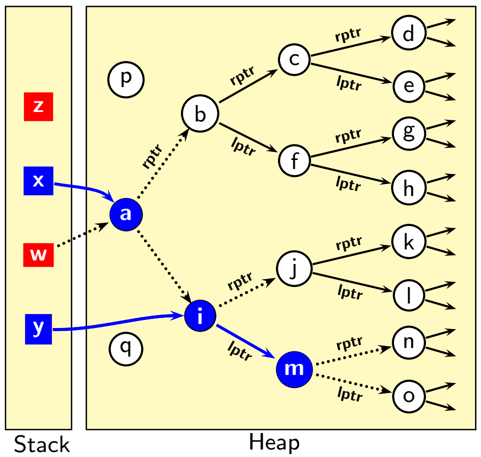
Our Solution

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

Our Solution

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

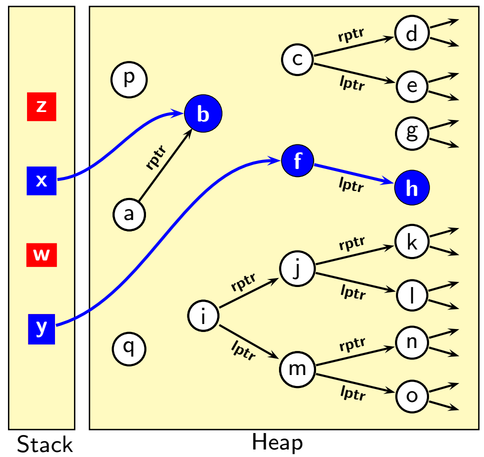
While loop is not executed even once



Our Solution

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

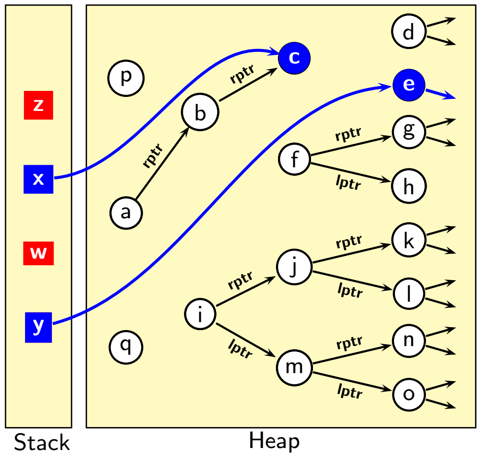
While loop is executed once



Our Solution

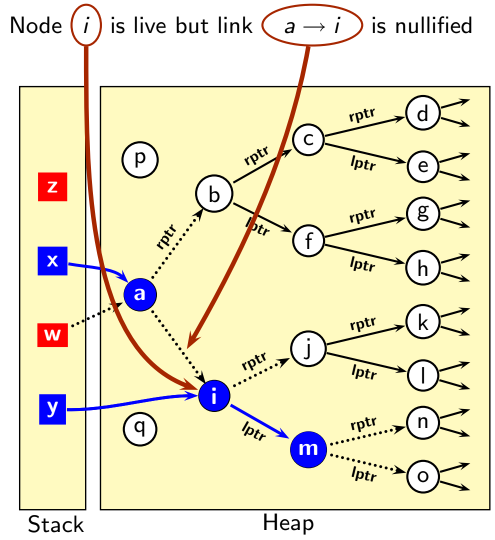
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {     x.lptr = null
3         x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New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

While loop is executed twice



Some Observations

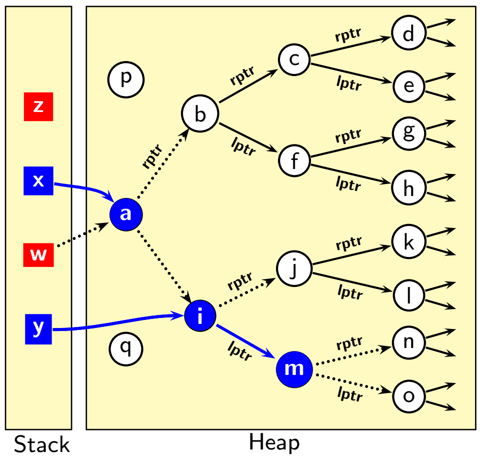
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {     x.lptr = null
3         x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = New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



Some Observations

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

New access expressions are created.
Can they cause exceptions?



BTW, What is Static Analysis of Heap?