IIT Bombay
cs618: Program Analysis

Topic:

cs618 Introduction

Section:

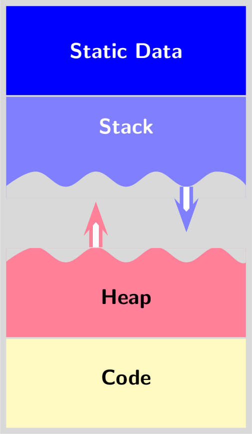
Classical Optimizations

Optimizing Heap Memory Usage

What is Program Analysis?

Course Details

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

    $\Rightarrow$ 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?

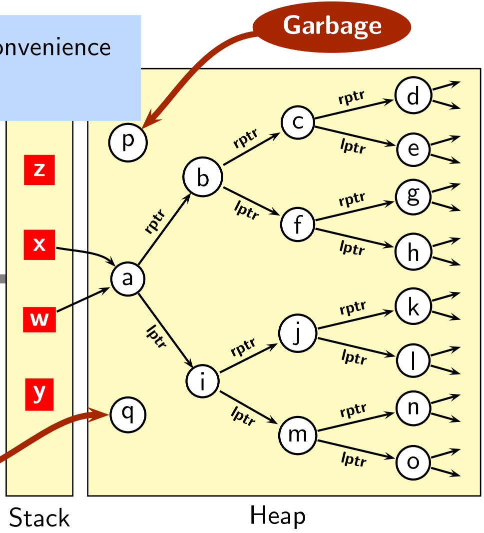
What is Garbage?

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?

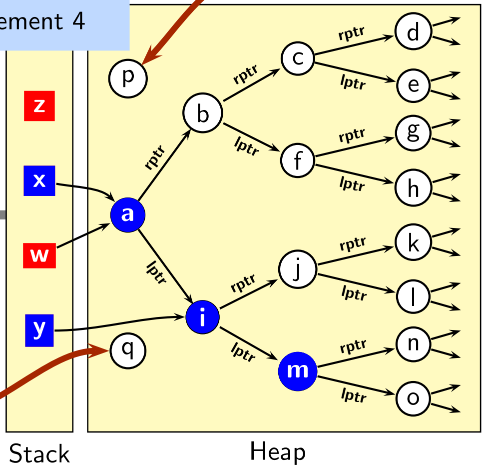
False

The blue nodes will be used after statement 4

Garbage

```
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
```



Garbage

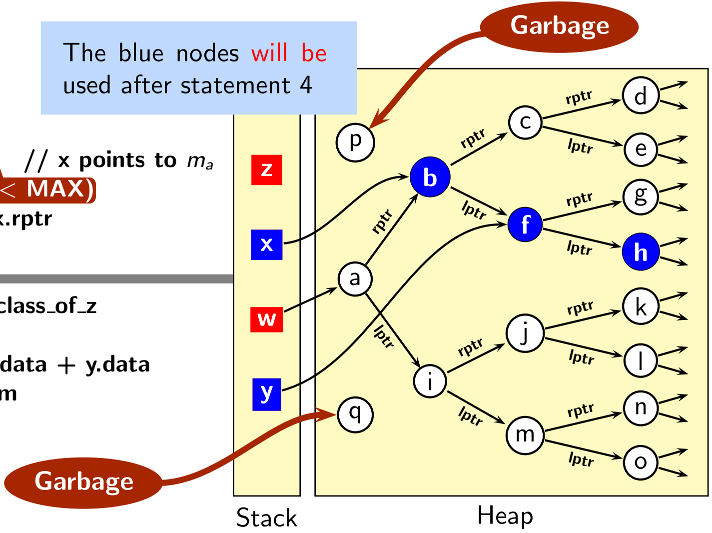
What is Garbage?

The blue nodes will be used after statement 4

True

```
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
```



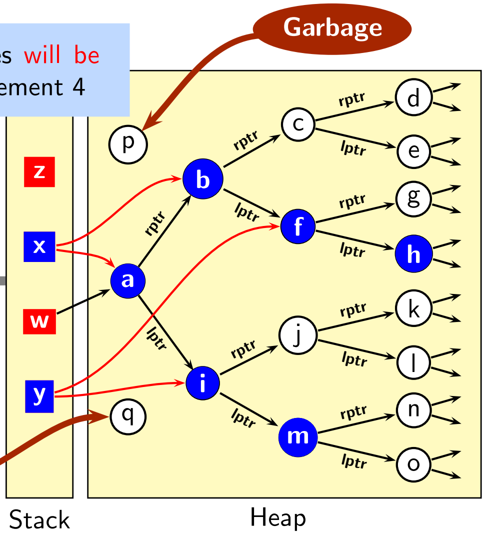
What is Garbage?

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
```

Garbage



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

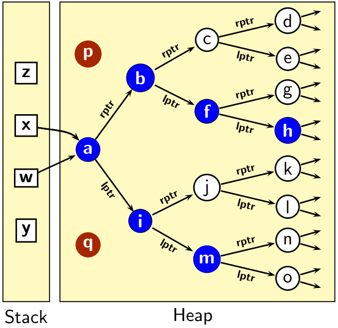
- The state of art compilers, virtual machines, garbage collectors cannot distinguish between the two

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

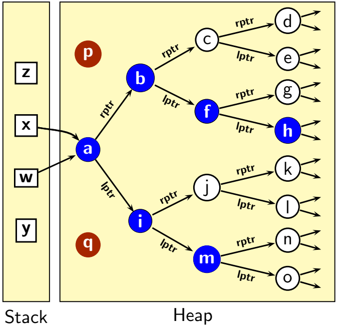


Reachability and Liveness

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

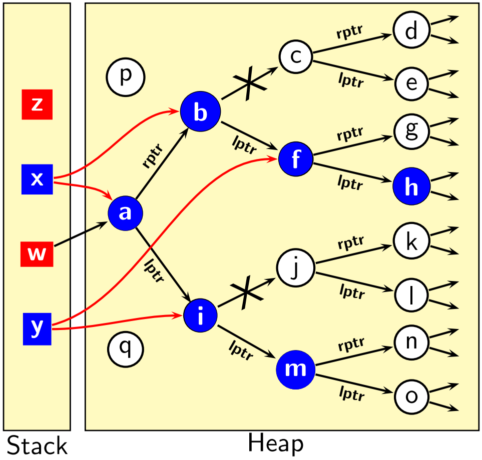


$$\text{Live} \subseteq \text{Reachable} \subseteq \text{Allocated}$$

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

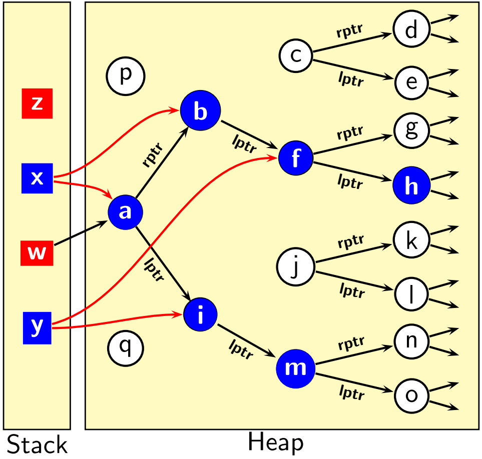
Cedar Mesa Folk Wisdom

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.
- Uses liveness data flow analysis of root variables (stack data).
- What about liveness of heap data?

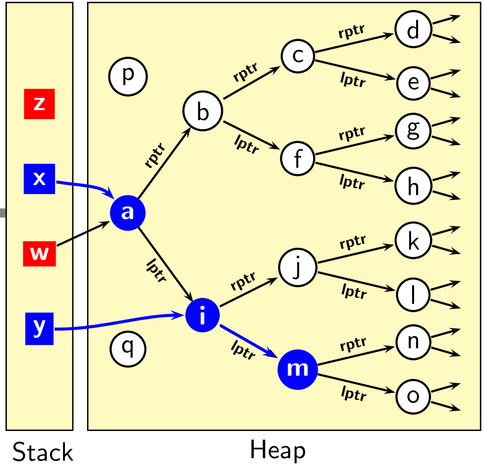
Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the while loop is not executed even once.

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

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



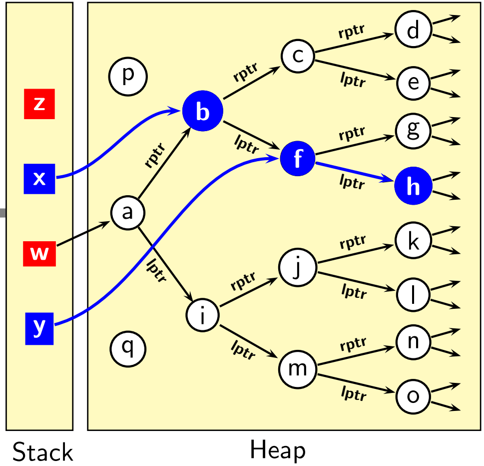
Applying Cedar Mesa Folk Wisdom to Heap Data

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
```



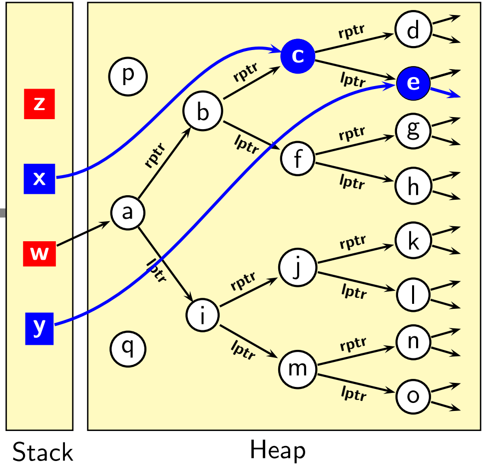
Applying Cedar Mesa Folk Wisdom to Heap Data

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?

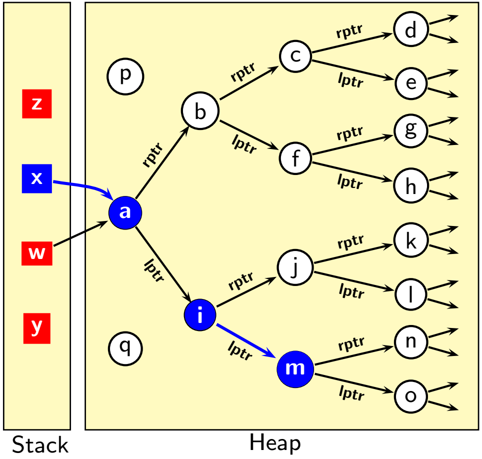
Our Solution (1)

```
                    y = z = null
w = x
                    w = null
while (x.data < MAX)
{                   x.lptr = null
    x = x.rptr      }
                    x.rptr = x.lptr.rptr = null
                    x.lptr.lptr.lptr = 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
```

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
```

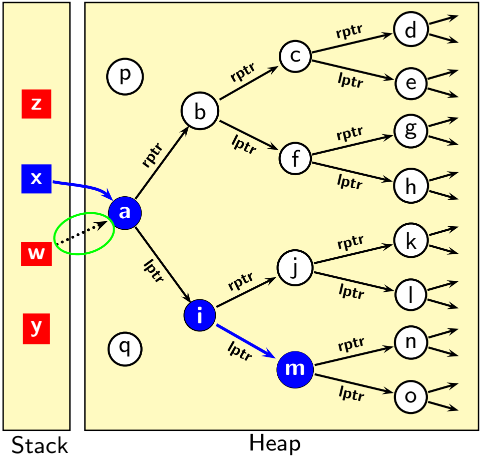
While loop is not executed even once



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
```

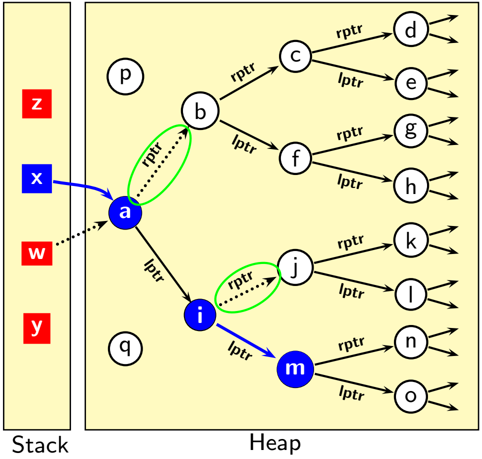
While loop is not executed even once



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
```

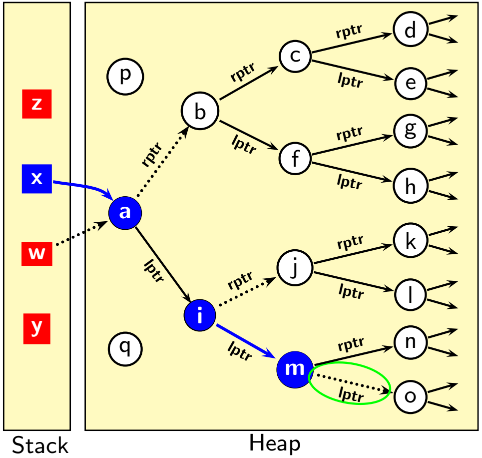
While loop is not executed even once



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
```

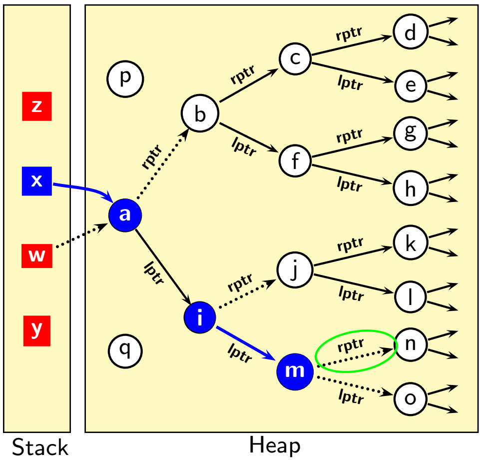
While loop is not executed even once



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
```

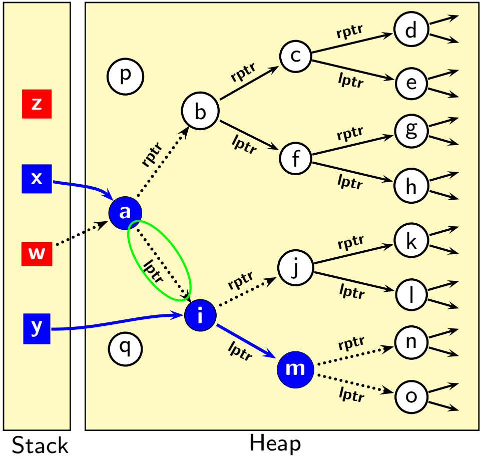
While loop is not executed even once



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
```

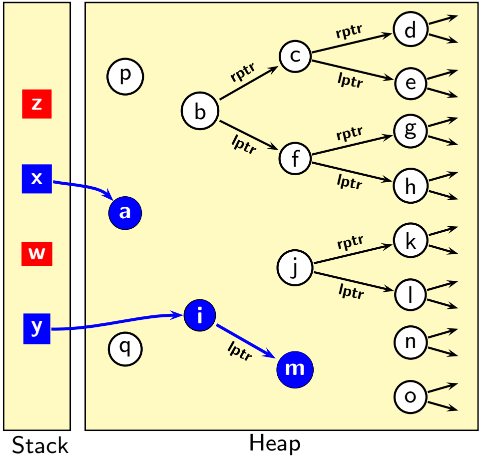
While loop is not executed even once



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
```

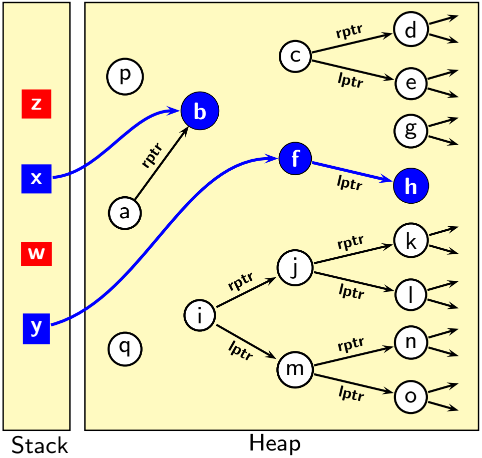
While loop is not executed even once



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
```

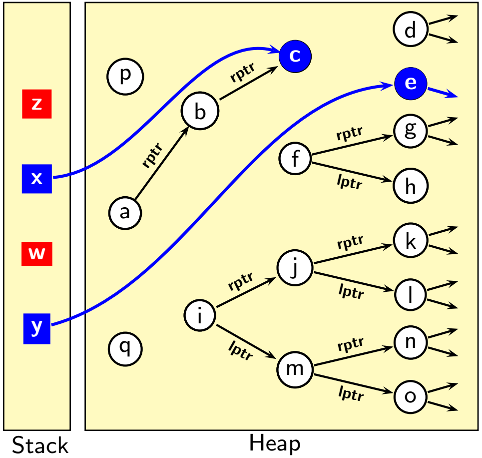
While loop is executed once



Our Solution (2)

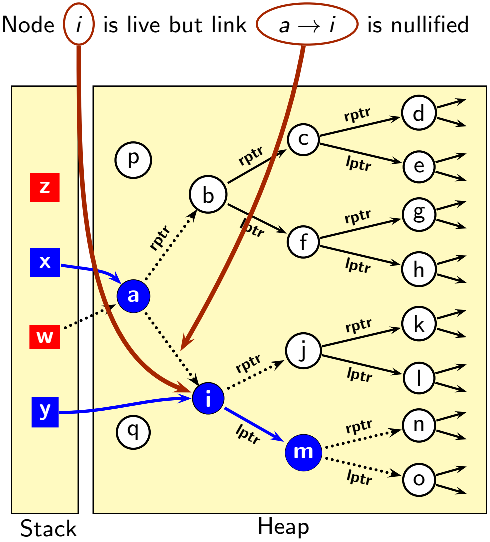
```
y = z = null
w = x
w = null
while (x.data < MAX)
{       x.lptr = null
        x = x.rptr       }
x.rptr = x.lptr.rptr = null
x.lptr.lptr.lptr = null
x.lptr.lptr.rptr = null
y = x.lptr
x.lptr = y.rptr = null
y.lptr.lptr = y.lptr.rptr = null
z = New class_of_z
z.lptr = z.rptr = null
y = y.lptr
y.lptr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

While loop is executed twice



Some Observations

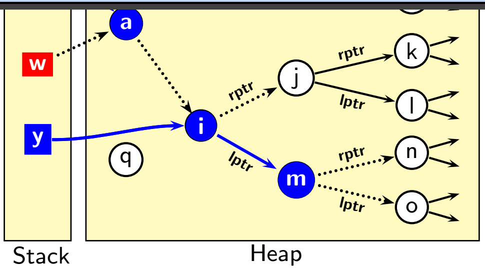
```
    y = z = null
1   w = x
    w = null
2   while (x.data < MAX)
    {       x.lptr = null
3           x = x.rptr        }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z = 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
```



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
```

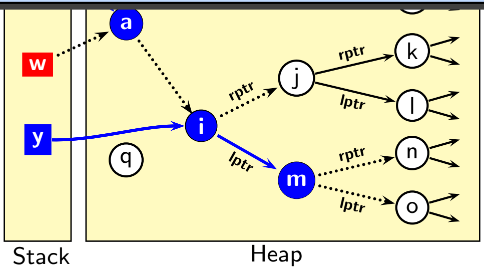
- The memory address that x holds when the execution reaches a given program point is not an invariant of program execution



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
```

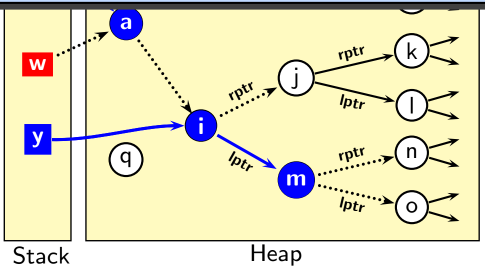
- The memory address that *x* holds when the execution reaches a given program point is not an invariant of program execution
- Whether we dereference lptr out of *x* or rptr out of *x* at a given program point is an invariant of program execution



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
```

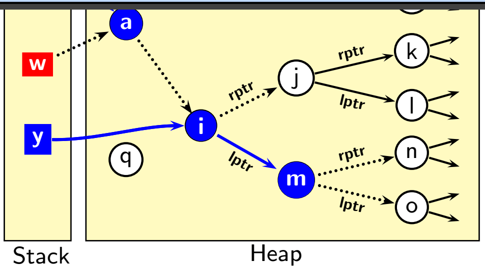
- The memory address that *x* holds when the execution reaches a given program point is not an invariant of program execution
- Whether we dereference lptr out of *x* or rptr out of *x* at a given program point is an invariant of program execution
- *A static analysis can discover only invariants*



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*



An Overview of Heap Reference Analysis

- A reference (called a *link*) can be represented by an *access path*.

  Eg. "$x \rightarrow$ lptr $\rightarrow$ rptr"

- A link may be accessed in multiple ways

- Setting links to null

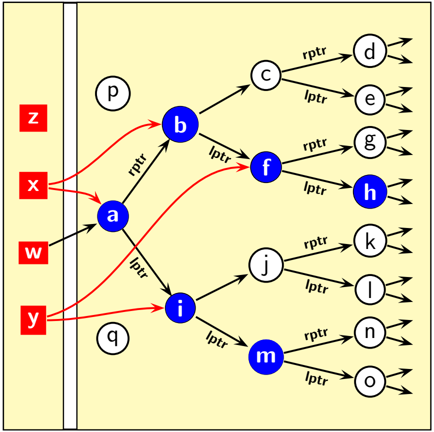
  - *Alias Analysis*. Identify all possible ways of accessing a link
  - *Liveness Analysis*. For each program point, identify "dead" links (i.e. links which are not accessed after that program point)
  - *Availability and Anticipability Analyses*. Dead links should be reachable for making null assignment.
  - *Code Transformation*. Set "dead" links to null

Assumptions

For simplicity of exposition

- Java model of heap access
    - Root variables are on stack and represent references to memory in heap.
    - Root variables cannot be pointed to by any reference.
- Simple extensions for C++
    - Root variables can be pointed to by other pointers.
    - Pointer arithmetic is not handled.

Key Idea #1 : Access Paths Denote Links



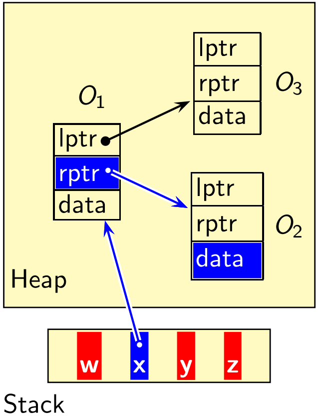
- Root variables : x, y, z
- Field names : rptr, lptr
- Access path : x→rptr→lptr
  Semantically, sequence of "links"
- Frontier : name of the last link
- Live access path : If the link corresponding to its frontier is used in future

What Makes a Link Live?

Assuming that a statement must be executed, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for accessing the contents of the corresponding target object:

Example	Objects read	Live access paths
sum = x.rptr.data	x, O_1, O_2	$x, x\rightarrow$rptr
if (x.rptr.data < sum)	x, O_1, O_2	$x, x\rightarrow$rptr

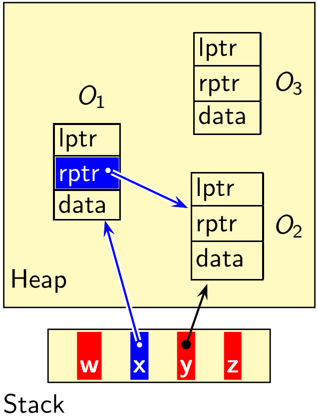


What Makes a Link Live?

Assuming that a statement must be executed, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for copying the contents of the corresponding target object:

Example	Objects read	Live access paths
$y = x.\text{rptr}$	x, O_1	$x, x.\text{rptr}$

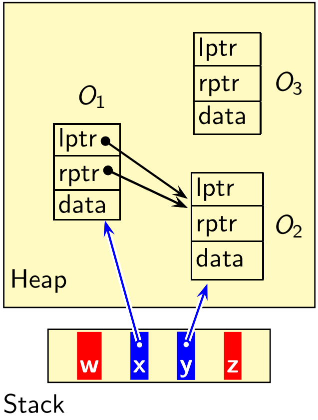


What Makes a Link Live?

Assuming that a statement must be executed, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for copying the contents of the corresponding target object:

Example	Objects read	Live access paths
$y = x.\text{rptr}$	x, O_1	$x, x.\text{rptr}$
$x.\text{lptr} = y$	x, O_1, y	x, y

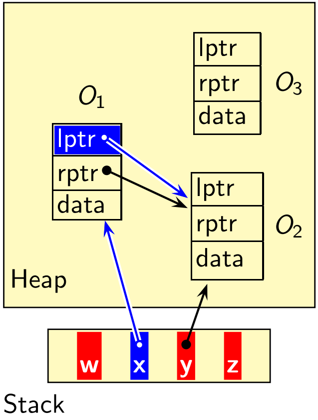


What Makes a Link Live?

Assuming that a statement must be executed, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for comparing the address of the corresponding target object:

Example	Objects read	Live access paths
if (x.lptr == null)	x, O_1	x, x→lptr

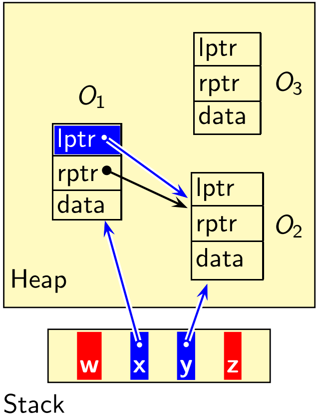


What Makes a Link Live?

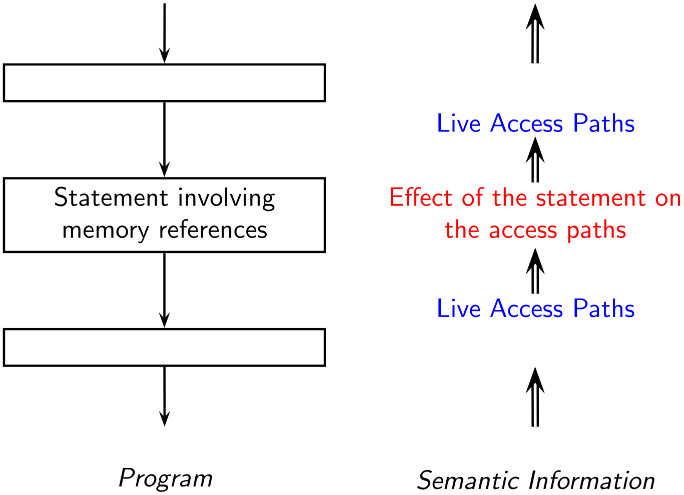
Assuming that a statement must be executed, if nullifying a link read in the statement can change the semantics of the program, then the link is live.

Reading a link for comparing the address of the corresponding target object:

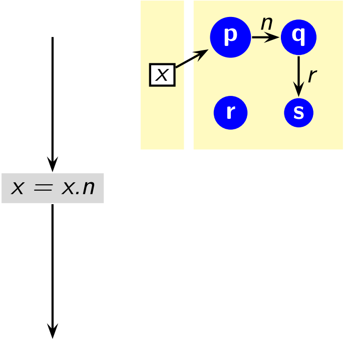
Example	Objects read	Live access paths
if (x.lptr == null)	x, O_1	x, $x \rightarrow$lptr
if (y == x.lptr)	x, O_1, y	x, $x \rightarrow$lptr, y



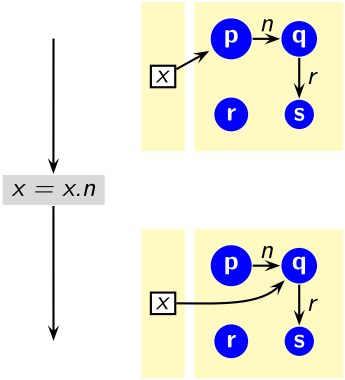
Liveness Analysis



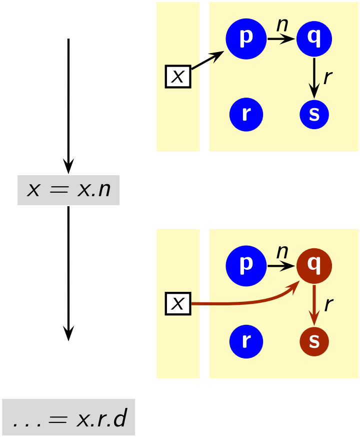
Key Idea #2 : Transfer of Access Paths



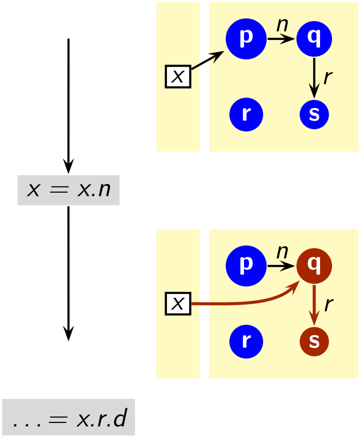
Key Idea #2 : Transfer of Access Paths



Key Idea #2 : Transfer of Access Paths

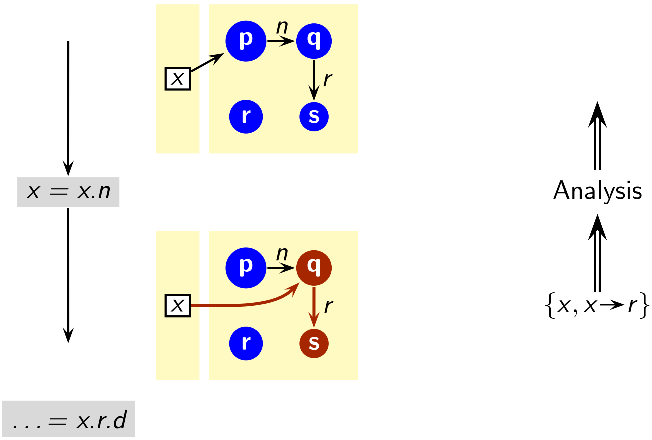


Key Idea #2 : Transfer of Access Paths

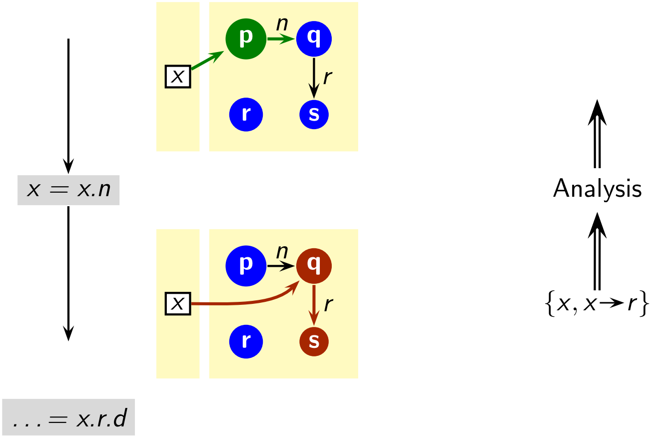


$\{x, x \rightarrow r\}$

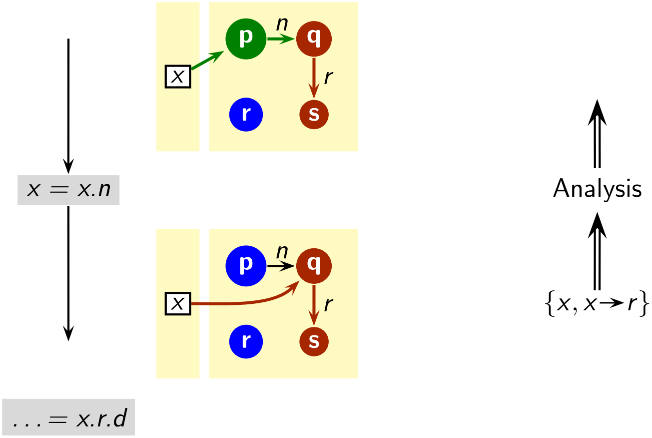
Key Idea #2 : Transfer of Access Paths



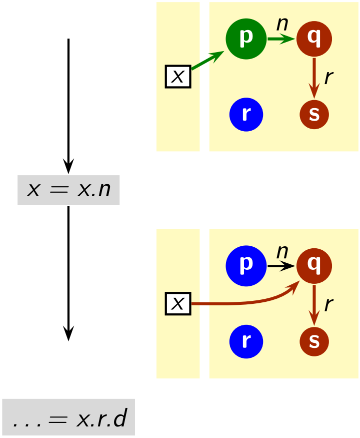
Key Idea #2 : Transfer of Access Paths



Key Idea #2 : Transfer of Access Paths



Key Idea #2 : Transfer of Access Paths



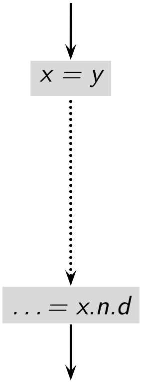
Generated	$\{x, x\rightarrow n, x\rightarrow n\rightarrow r\}$
Killed	$\{x, x\rightarrow r\}$

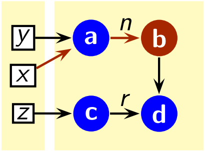
Analysis

$\{x, x\rightarrow r\}$

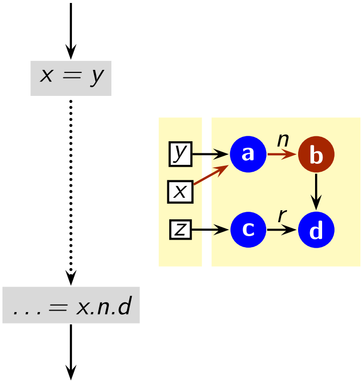
x after the assignment is same as $x\rightarrow n$ before the assignment

Key Idea #3 : Liveness Closure Under Link Aliasing





Key Idea #3 : Liveness Closure Under Link Aliasing

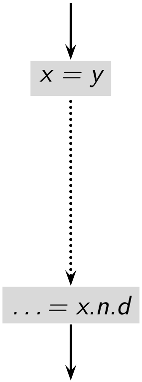


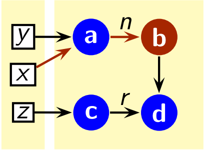
x and y are node aliases

$x.n$ and $y.n$ are link aliases

$x \rightarrow n$ is live $\Rightarrow$ $y \rightarrow n$ is live

Key Idea #3 : Liveness Closure Under Link Aliasing





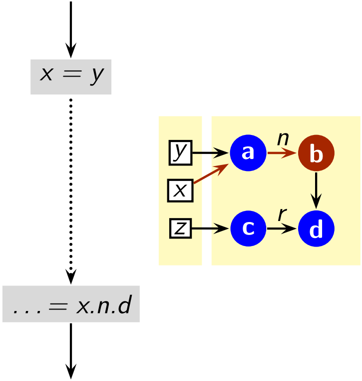
x and y are node aliases

$x.n$ and $y.n$ are link aliases

$x \rightarrow n$ is live $\Rightarrow$ $y \rightarrow n$ is live

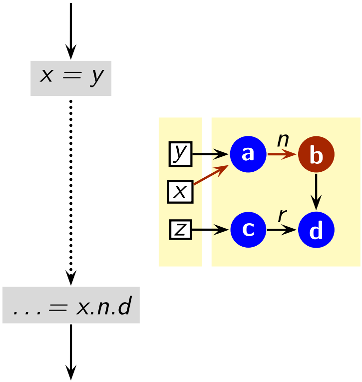
Nullifying $y \rightarrow n$ will have the side effect of nullifying $x \rightarrow n$

Explicit and Implicit Liveness



$x \rightarrow n$ is live $\Rightarrow$ $y \rightarrow n$ is live

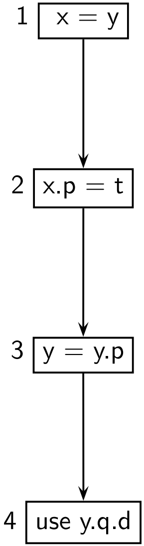
Explicit and Implicit Liveness



$x \rightarrow n$ is live $\Rightarrow$ $y \rightarrow n$ is live

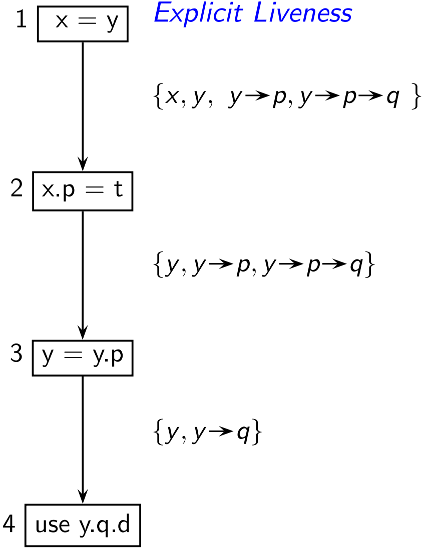
$y \rightarrow n$ is implicitly live
$x \rightarrow n$ is explicitly live

Key Idea #4: Aliasing is Required with Explicit Liveness

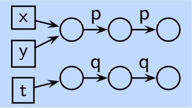


Key Idea #4: Aliasing is Required with Explicit Liveness

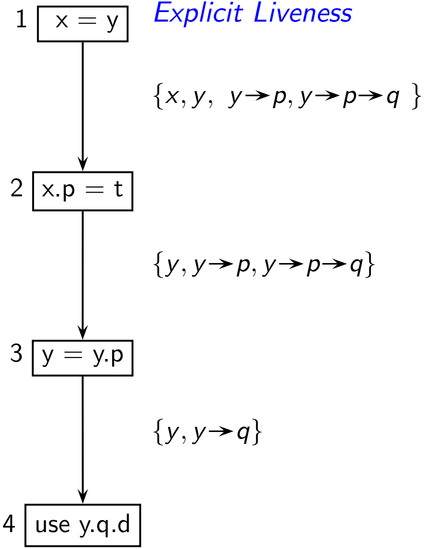
Explicit Liveness



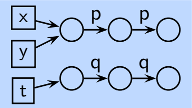
Key Idea #4: Aliasing is Required with Explicit Liveness

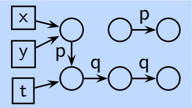


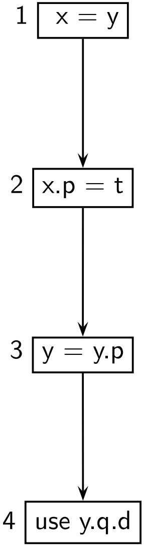
Explicit Liveness



Key Idea #4: Aliasing is Required with Explicit Liveness







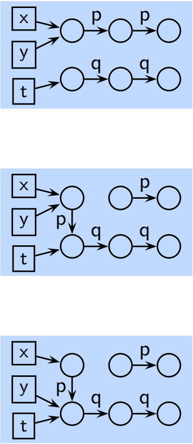
Explicit Liveness

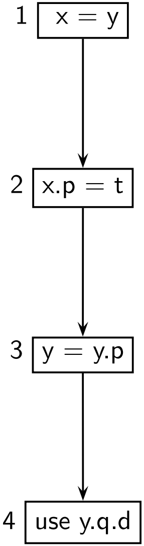
$\{x, y,\ y \rightarrow p, y \rightarrow p \rightarrow q\ \}$

$\{y, y \rightarrow p, y \rightarrow p \rightarrow q\}$

$\{y, y \rightarrow q\}$

Key Idea #4: Aliasing is Required with Explicit Liveness





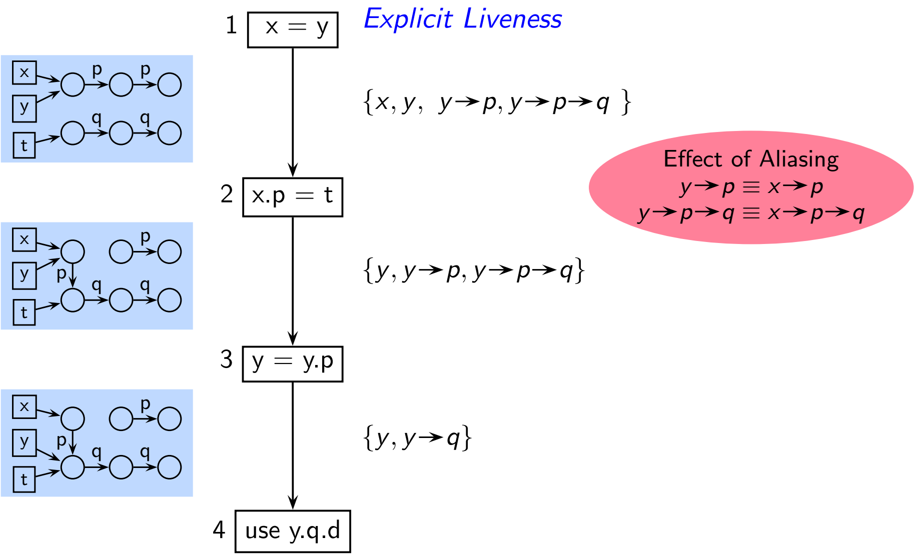
Explicit Liveness

$\{x, y,\ y \rightarrow p, y \rightarrow p \rightarrow q\ \}$

$\{y, y \rightarrow p, y \rightarrow p \rightarrow q\}$

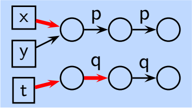
$\{y, y \rightarrow q\}$

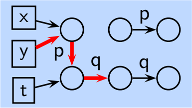
Key Idea #4: Aliasing is Required with Explicit Liveness

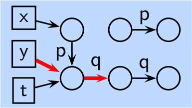


Key Idea #4: Aliasing is Required with Explicit Liveness

	Statement	*Explicit Liveness*	*Required Liveness*
1	x = y		
		$\{x, y,\ y\rightarrow p, y\rightarrow p\rightarrow q\ \}$	$\{x, y,\ t, t\rightarrow q\ \}$
2	x.p = t		
		$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$	$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$
3	y = y.p		
		$\{y, y\rightarrow q\}$	$\{y, y\rightarrow q\}$
4	use y.q.d		

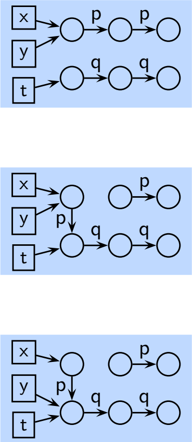






Effect of Aliasing
$y\rightarrow p \equiv x\rightarrow p$
$y\rightarrow p\rightarrow q \equiv x\rightarrow p\rightarrow q$

Key Idea #4: Aliasing is Required with Explicit Liveness



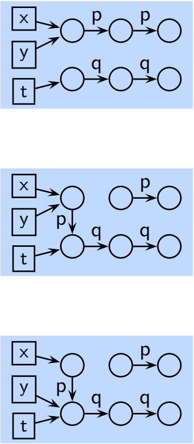
Statement	*Explicit Liveness*	*Required Liveness*
	$\{x, y, y\rightarrow p, y\rightarrow p\rightarrow q\}$	$\{x, y, t, t\rightarrow q\}$
1 x = y		
2 x.p = t	$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$	$\{y, y\rightarrow p, y\rightarrow p\rightarrow q\}$
3 y = y.p	$\{y, y\rightarrow q\}$	$\{y, y\rightarrow q\}$
4 use y.q.d		

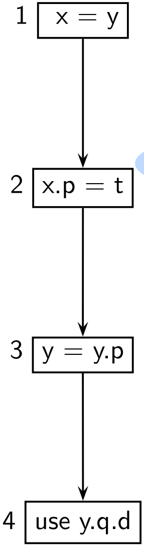
Spurious: $y\rightarrow p, y\rightarrow p\rightarrow q$

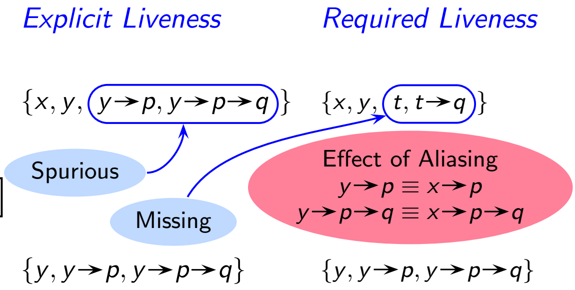
Missing: $t, t\rightarrow q$

Effect of Aliasing
$y\rightarrow p \equiv x\rightarrow p$
$y\rightarrow p\rightarrow q \equiv x\rightarrow p\rightarrow q$

Key Idea #4: Aliasing is Required with Explicit Liveness







The need of link alias closure of LHS

- Transferring liveness to RHS (soundness)
- Killing liveness (precision)

Link alias closure of RHS can be computed later for implicit liveness but transfer and killing cannot wait

Notation for Defining Flow Functions for Explicit Liveness

- Basic entities
  - Variables $u, v \in \mathbb{V}\text{ar}$
  - Pointer variables $w, x, y, z \in \mathbf{P} \subseteq \mathbb{V}\text{ar}$
  - Pointer fields $f, g, h \in pF$
  - Non-pointer fields $a, b, c, d \in npF$
- Additional notation
  - Sequence of pointer fields $\sigma \in pF^*$ (could be ϵ)
  - Access paths $\rho \in \mathbf{P} \times pF^*$
    Example: $\{x, x \rightarrow f, x \rightarrow f \rightarrow g\}$
  - Summarized access paths rooted at x or $x \rightarrow \sigma$ for a given x and σ
    - $x \rightarrow * = \{x \rightarrow \sigma \mid \sigma \in pF^*\}$
    - $x \rightarrow \sigma \rightarrow * = \{x \rightarrow \sigma \rightarrow \sigma' \mid \sigma' \in pF^*\}$

Data Flow Equations for Explicit Liveness Analysis

$$In_n = \left(Out_n - \text{Kill}_n(Out_n)\right) \cup \text{Gen}_n(Out_n)$$

$$Out_n = \begin{cases} BI & n \text{ is } End \\ \bigcup\limits_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

Flow Functions for Explicit Liveness Analysis

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \;\middle|\; z \rightarrow f \rightarrow \sigma \in X, z \in A(x)\right\}$	$\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

Flow Functions for Explicit Liveness Analysis

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \,\middle|\, z \rightarrow f \rightarrow \sigma \in X, z \in A(x)\right\}$	$\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

May link aliasing for soundness

Flow Functions for Explicit Liveness Analysis

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \mid z \rightarrow f \rightarrow \sigma \in X, z \in A(x)\right\}$	$\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

May link aliasing for soundness

Must link aliasing for precision

Flow Functions for Explicit Liveness Analysis

Let

- Why is $y \notin \text{Gen}_n(X)$ for $x.f = y$ when $x \notin X$?

  Strong liveness

- Why is $y \notin \text{Gen}_n(X)$ for $x = y.f$ when $x \rightarrow \sigma \notin X$?

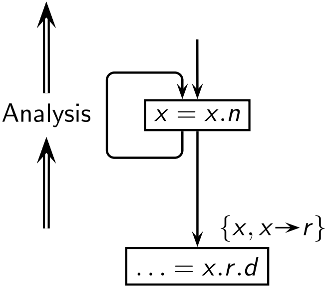
  Strong liveness

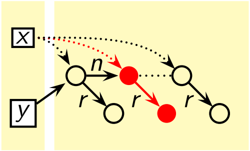
- Why is $x \notin \text{Gen}_n(X)$ for $x.f = y$?
  - If $x \rightarrow f \rightarrow \sigma \notin Out_n$, we can do dead code elimination
  - If $\exists\, x \rightarrow f \rightarrow \sigma \in Out_n$, then $\exists\, x \in Out_n$
    It will not be killed, so no need of $x \in \text{Gen}_n$

  Unlike LFCPA, $x.f$ cannot point to a variable whose pointees are to be found

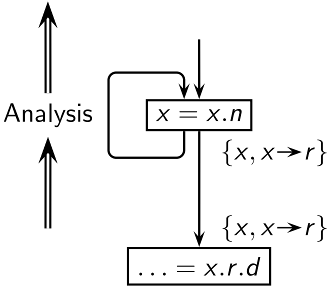
  We are using a storeless abstraction of memory, LFCPA uses a store-based abstraction

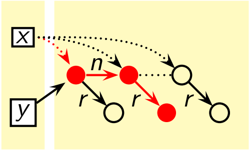
Computing Explicit Liveness Using Sets of Access Paths

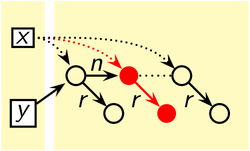




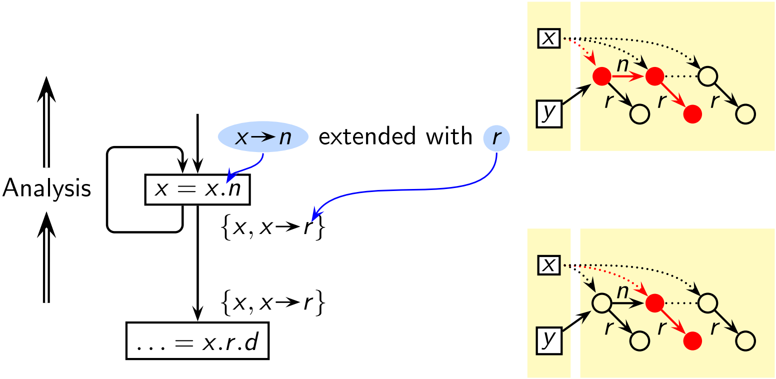
Computing Explicit Liveness Using Sets of Access Paths



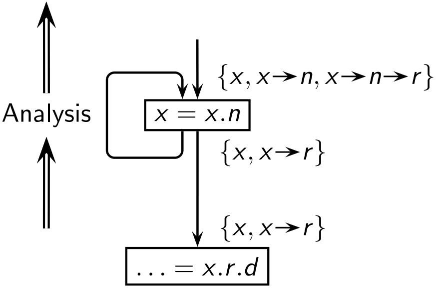


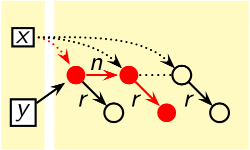


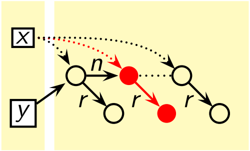
Computing Explicit Liveness Using Sets of Access Paths



Computing Explicit Liveness Using Sets of Access Paths

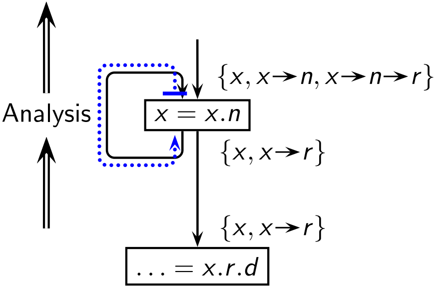






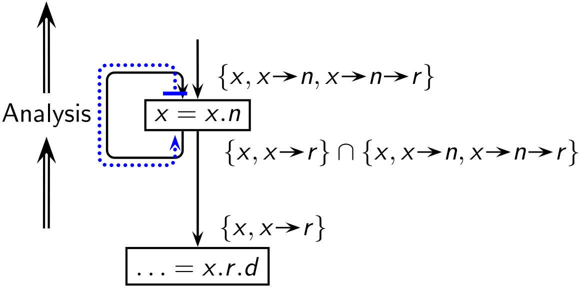
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



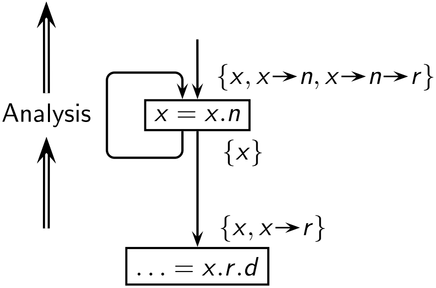
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



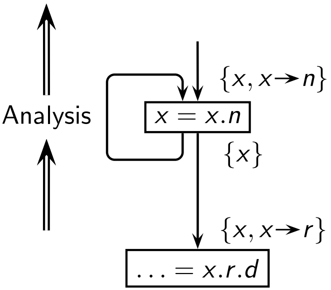
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



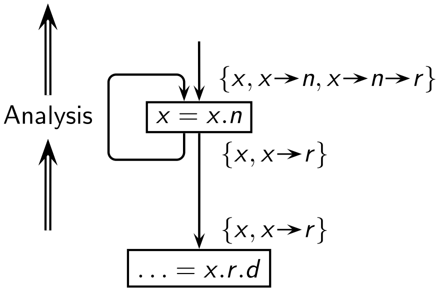
Computing Explicit Liveness Using Sets of Access Paths

Anticipability of Heap References: An *All Paths* problem



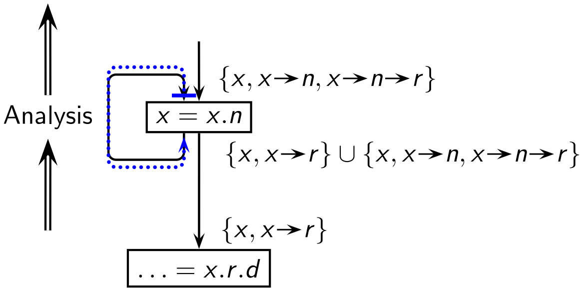
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



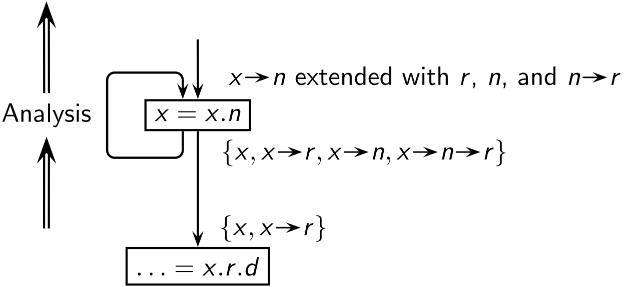
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



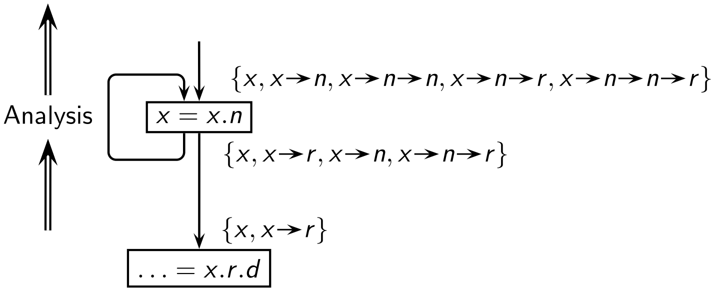
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



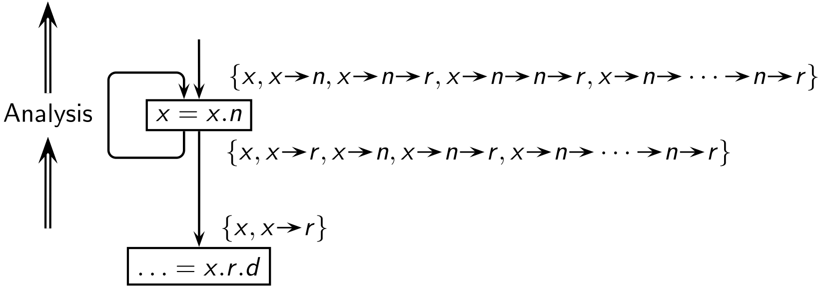
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



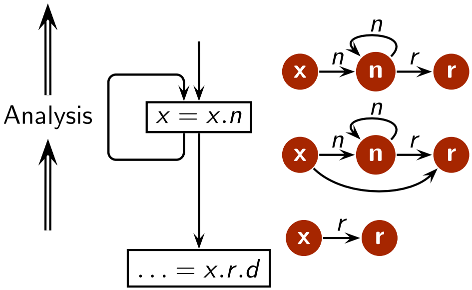
Computing Explicit Liveness Using Sets of Access Paths

Liveness of Heap References: An *Any Path* problem



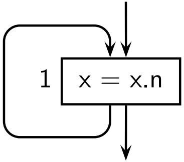
Infinite Number of Unbounded Access Paths

Key Idea #5: Using Graphs as Data Flow Values



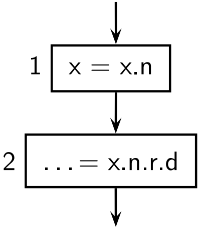
Finite Number of Bounded Structures

Key Idea #6 : Include Program Point in Graphs



$\{x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow n \rightarrow n, \ldots\}$

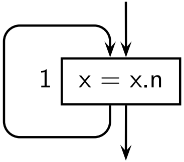
Different occurrences of n's in an access path are
Indistinguishable



$\{x, x \rightarrow n, x \rightarrow n \rightarrow n, x \rightarrow n \rightarrow n \rightarrow r\}$

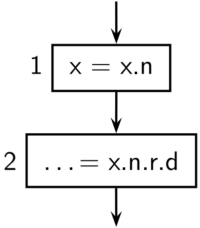
Different occurrences of n's in an access path are
Distinct

Key Idea #6 : Include Program Point in Graphs



$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow n, \ldots\}$

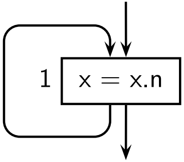
Different occurrences of n's in an access path are
Indistinguishable



$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow r\}$

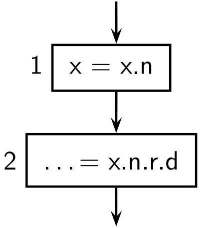
Different occurrences of n's in an access path are
Distinct
(pattern of subsequent dereferences could be distinct)

Key Idea #6 : Include Program Point in Graphs



$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow n, \ldots\}$

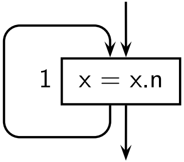
Different occurrences of n's in an access path are
Indistinguishable
(pattern of subsequent dereferences remains same)



$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow r\}$

Different occurrences of n's in an access path are
Distinct
(pattern of subsequent dereferences could be distinct)

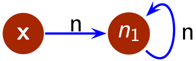
Key Idea #6 : Include Program Point in Graphs

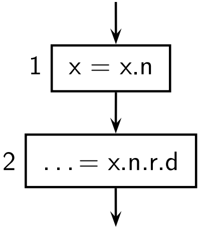


$\{x, x\to n, x\to n\to n, x\to n\to n\to n, \ldots\}$

Different occurrences of n's in an access path are
Indistinguishable
(pattern of subsequent dereferences remains same)

Access Graph :

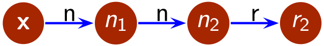




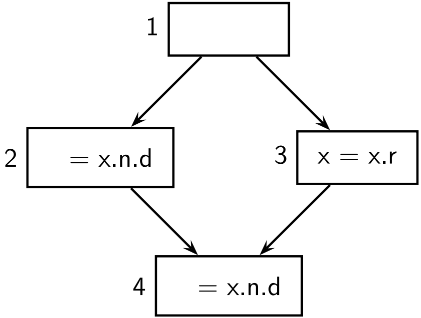
$\{x, x\to n, x\to n\to n, x\to n\to n\to r\}$

Different occurrences of n's in an access path are
Distinct
(pattern of subsequent dereferences could be distinct)

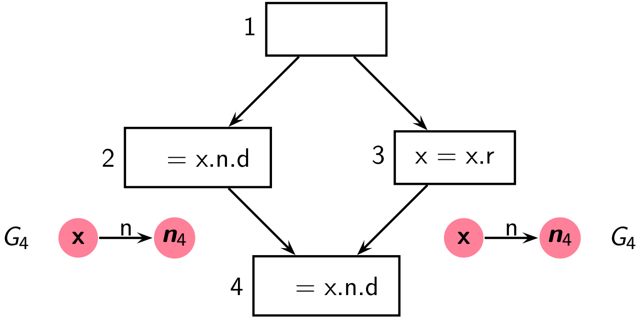
Access Graph :



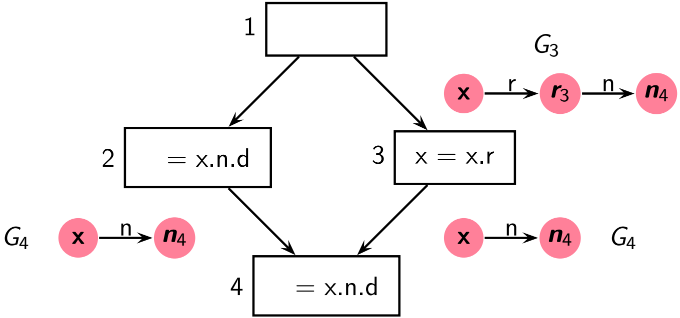
Inclusion of Program Point Facilitates Summarization



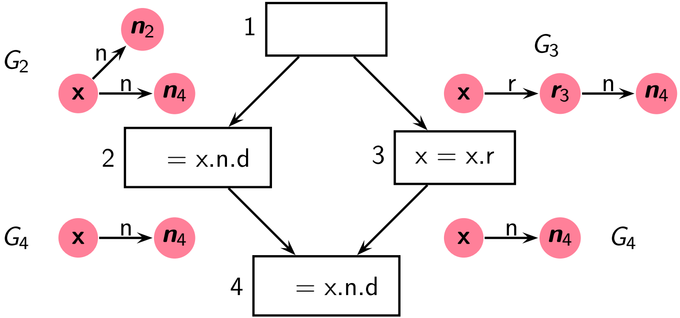
Inclusion of Program Point Facilitates Summarization



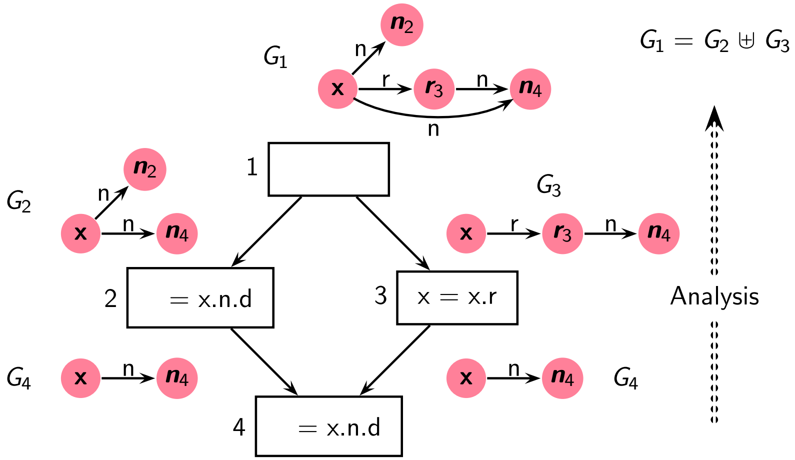
Inclusion of Program Point Facilitates Summarization



Inclusion of Program Point Facilitates Summarization

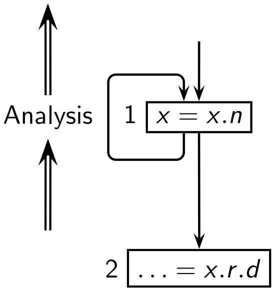


Inclusion of Program Point Facilitates Summarization



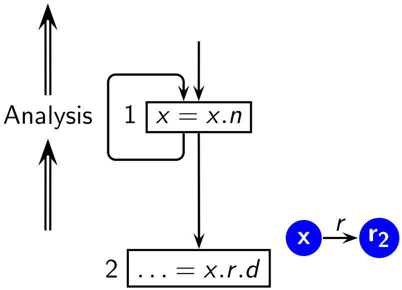
Inclusion of Program Point Facilitates Summarization

Iteration #1

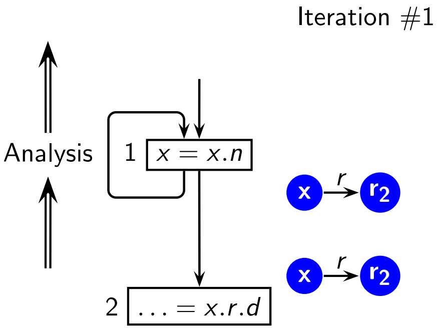


Inclusion of Program Point Facilitates Summarization

Iteration #1

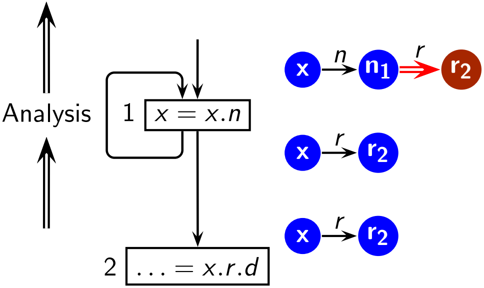


Inclusion of Program Point Facilitates Summarization

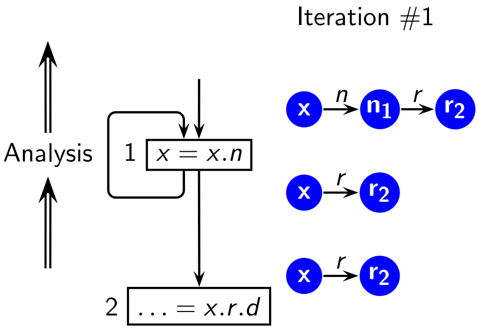


Inclusion of Program Point Facilitates Summarization

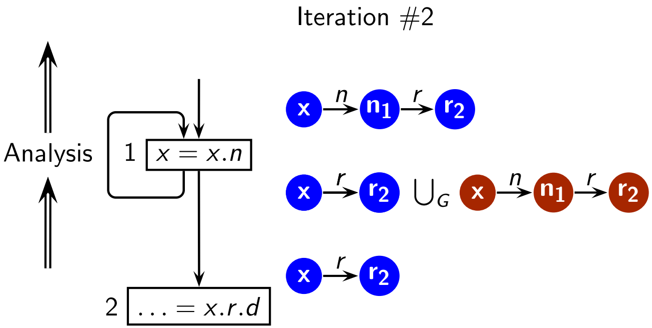
Iteration #1



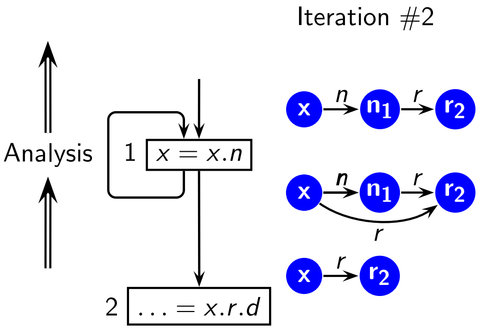
Inclusion of Program Point Facilitates Summarization



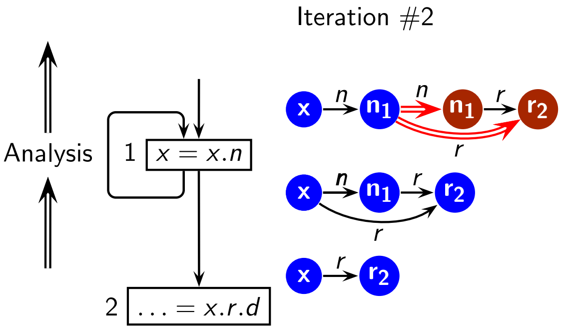
Inclusion of Program Point Facilitates Summarization



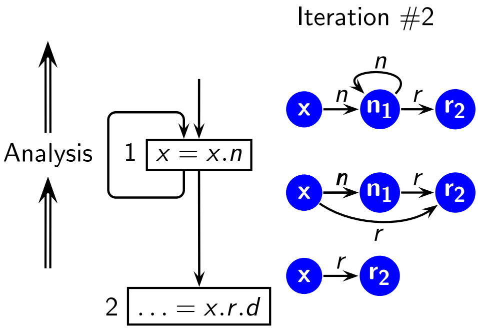
Inclusion of Program Point Facilitates Summarization



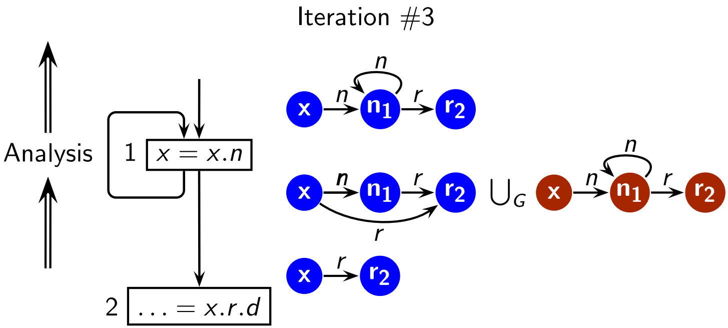
Inclusion of Program Point Facilitates Summarization



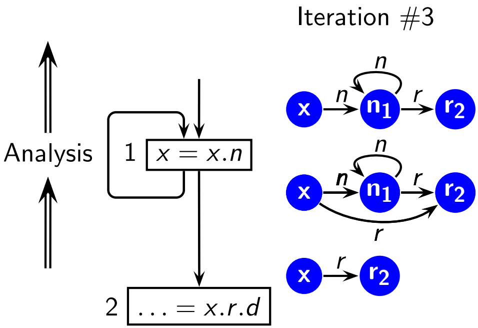
Inclusion of Program Point Facilitates Summarization



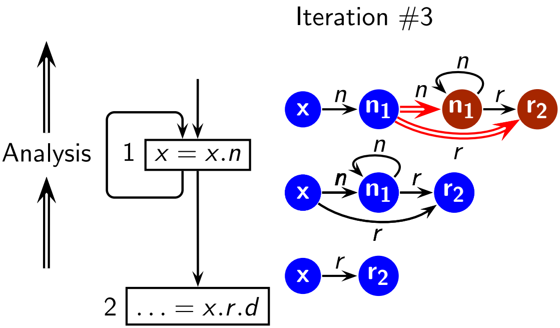
Inclusion of Program Point Facilitates Summarization



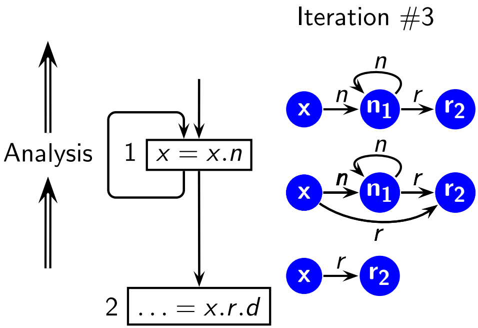
Inclusion of Program Point Facilitates Summarization



Inclusion of Program Point Facilitates Summarization

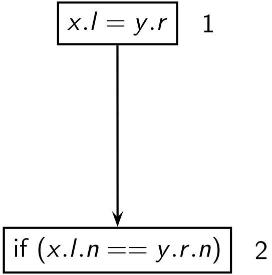


Inclusion of Program Point Facilitates Summarization



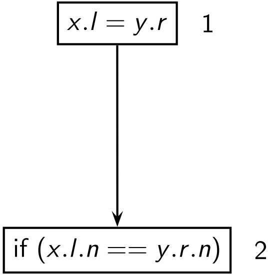
Access Graph and Memory Graph

Program Fragment

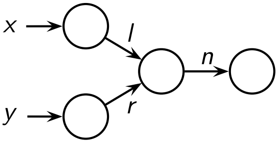


Access Graph and Memory Graph

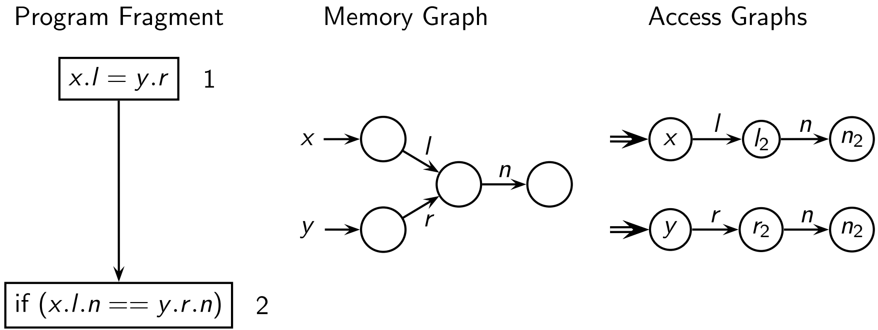
Program Fragment



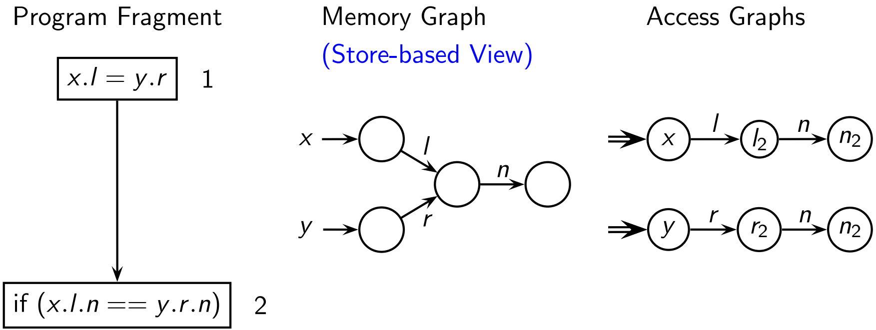
Memory Graph



Access Graph and Memory Graph

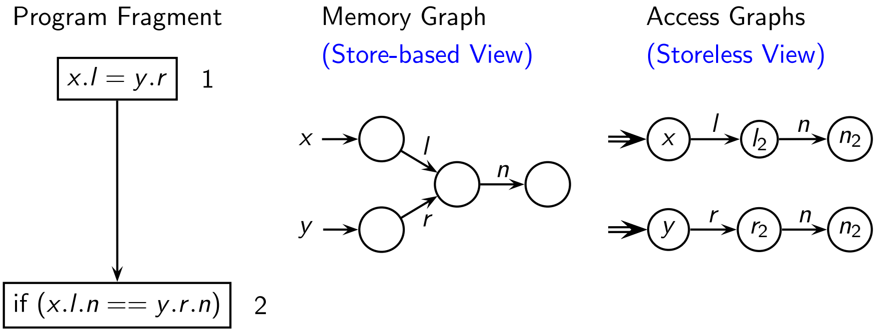


Access Graph and Memory Graph



- Memory Graph: Nodes represent locations and edges represent links (i.e. pointers).

Access Graph and Memory Graph



- Memory Graph: Nodes represent locations and edges represent links (i.e. pointers).
- Access Graphs: Nodes represent dereference of links at particular statements. Memory locations are implicit.

Lattice of Access Graphs

- Finite number of nodes in an access graph for a variable
- $\uplus$ induces a partial order on access graphs
  - $\Rightarrow$ a finite (and hence complete) lattice
  - $\Rightarrow$ All standard results of classical data flow analysis can be extended to this analysis.

    Termination and boundedness, convergence on MFP, complexity etc.

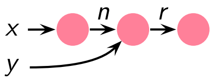
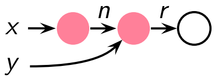
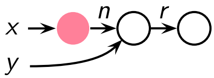
Access Graph Operations

- *Union.* $G \uplus G'$
- *Path Removal*

  $G \ominus R$ removes those access paths in G which have $\rho \in R$ as a prefix
- *Factorization* $(/)$
- *Extension*

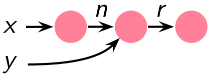
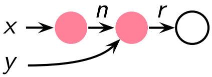
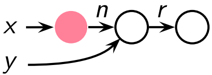
Defining Factorization

Given statement $x.n = y$, what should be the result of transfer?

Live AP	Memory Graph	Result of Transfer	Remainder
$x \rightarrow n \rightarrow r$		$y \rightarrow r$	r (LHS is contained in the live access path)
$x \rightarrow n$		y	ϵ (LHS is contained in the live access path)
x		no transfer	?? (LHS is not contained in the live access path)

Defining Factorization

Given statement $x.n = y$, what should be the result of transfer?

Live AP	Memory Graph	Result of Transfer	Remainder
$x \rightarrow n \rightarrow r$		$y \rightarrow r$	r (LHS is contained in the live access path)
$x \rightarrow n$		y	ϵ (LHS is contained in the live access path)
x		no transfer	?? (LHS is not contained in the live access path) Quotient is empty So no remainder

Semantics of Access Graph Operations

- $P(G)$ is the set of all paths in graph G
- $P(G, M)$ is the set of paths in G terminating on nodes in M
- S is the set of remainder graphs
- $P(S)$ is the set of all paths in all remainder graphs in S

Operation		Access Paths
Union	$G_3 = G_1 \uplus G_2$	$P(G_3) \supseteq P(G_1) \cup P(G_2)$
Path Removal	$G_2 = G_1 \ominus X$	$P(G_2) \supseteq P(G_1) - \{\rho\rightarrow\sigma \mid \rho \in X, \rho\rightarrow\sigma \in P(G_1)\}$
Factorization	$S = G_1/\rho$	$P(S) = \{\sigma \mid \rho\rightarrow\sigma \in P(G_1)\}$
Extension	$G_2 = (G_1, M)\#\emptyset$	$P(G_2) = \emptyset$
	$G_2 = (G_1, M)\# S$	$P(G_2) \supseteq P(G_1) \cup \{\rho\rightarrow\sigma \mid \rho \in P(G_1, M),\ \sigma \in P(S)\}$

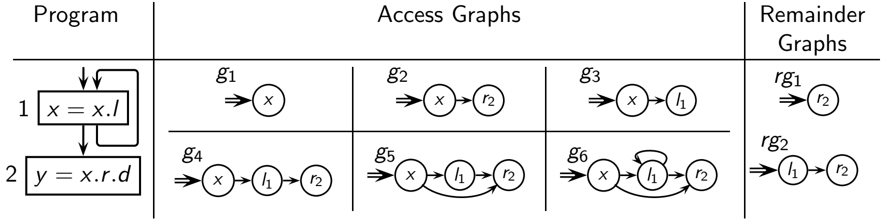
Semantics of Access Graph Operations

- $P(G)$ is the set of all paths in graph G
- $P(G, M)$ is the set of paths in G terminating on nodes in M
- S is the set of remainder graphs
- $P(S)$ is the set of all paths in all remainder graphs in S

Operation		Access Paths
Union	$G_3 = G_1 \uplus G_2$	$P(G_3) \supseteq P(G_1) \cup P(G_2)$
Path Removal	$G_2 = G_1 \ominus X$	$P(G_2) \supseteq P(G_1) - \{\rho \rightarrow \sigma \mid \rho \in X, \rho \rightarrow \sigma \in P(G_1)\}$
Factorization	$S = G_1/\rho$	$P(S) = \{\sigma \mid \rho \rightarrow \sigma \in P(G_1)\}$
Extension	$G_2 = (G_1, M)\# \emptyset$	$P(G_2) = \emptyset$
	$G_2 = (G_1, M)\# S$	$P(G_2) \supseteq P(G_1) \cup \{\rho \rightarrow \sigma \mid \rho \in P(G_1, M),\ \sigma \in P(S)\}$

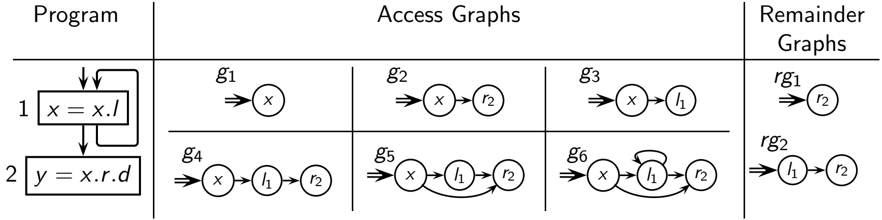
σ represents remainder

Access Graph Operations: Examples



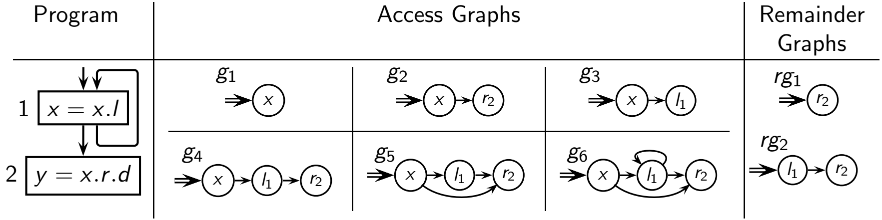
Union	Path Removal	Factorisation	Extension

Access Graph Operations: Examples



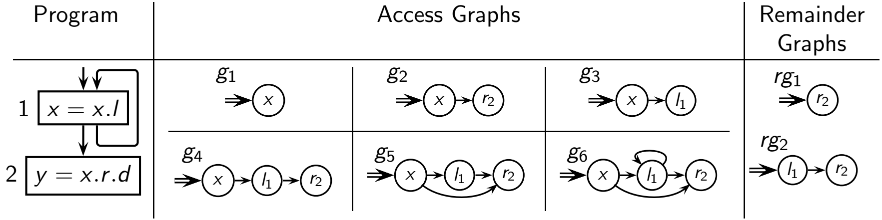
Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$ $g_2 \uplus g_4 = g_5$ $g_5 \uplus g_4 = g_5$ $g_5 \uplus g_6 = g_6$			

Access Graph Operations: Examples



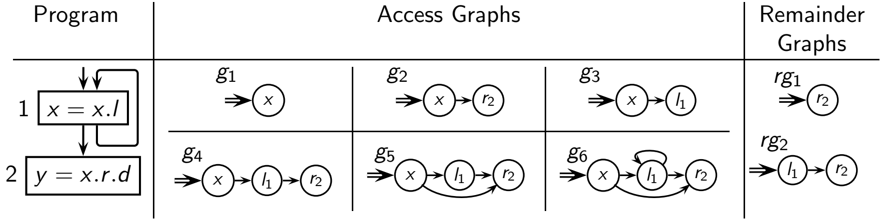
Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$ $g_2 \uplus g_4 = g_5$ $g_5 \uplus g_4 = g_5$ $g_5 \uplus g_6 = g_6$	$g_6 \ominus \{x \rightarrow l\} = g_2$ $g_5 \ominus \{x\} = \mathcal{E}_G$ $g_4 \ominus \{x \rightarrow r\} = g_4$ $g_4 \ominus \{x \rightarrow l\} = g_1$		

Access Graph Operations: Examples



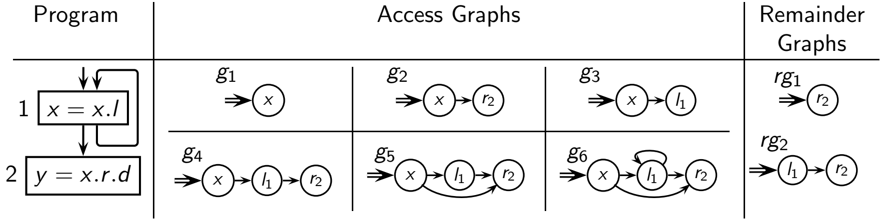
Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$	$g_6 \ominus \{x \rightarrow l\} = g_2$	$g_2 / x = \{rg_1\}$	
$g_2 \uplus g_4 = g_5$	$g_5 \ominus \{x\} = \mathcal{E}_G$	$g_5 / x = \{rg_1, rg_2\}$	
$g_5 \uplus g_4 = g_5$	$g_4 \ominus \{x \rightarrow r\} = g_4$	$g_5 / x \rightarrow r = \{\epsilon_{RG}\}$	
$g_5 \uplus g_6 = g_6$	$g_4 \ominus \{x \rightarrow l\} = g_1$	$g_4 / x \rightarrow r = \emptyset$	

Access Graph Operations: Examples



Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$	$g_6 \ominus \{x \rightarrow l\} = g_2$	$g_2 / x = \{rg_1\}$	$(g_3, \{l_1\}) \# \{rg_1\} = g_4$
$g_2 \uplus g_4 = g_5$	$g_5 \ominus \{x\} = \mathcal{E}_G$	$g_5 / x = \{rg_1, rg_2\}$	$(g_3, \{x, l_1\}) \# \{rg_1, rg_2\} = g_6$
$g_5 \uplus g_4 = g_5$	$g_4 \ominus \{x \rightarrow r\} = g_4$	$g_5 / x \rightarrow r = \{\epsilon_{RG}\}$	$(g_2, \{r_2\}) \# \{\epsilon_{RG}\} = g_2$
$g_5 \uplus g_6 = g_6$	$g_4 \ominus \{x \rightarrow l\} = g_1$	$g_4 / x \rightarrow r = \emptyset$	$(g_2, \{r_2\}) \# \emptyset = \mathcal{E}_G$

Access Graph Operations: Examples



Union	Path Removal	Factorisation	Extension
$g_3 \uplus g_4 = g_4$	$g_6 \ominus \{x\rightarrow l\} = g_2$	$g_2/x = \{rg_1\}$	$(g_3, \{l_1\}) \# \{rg_1\} = g_4$
$g_2 \uplus g_4 = g_5$	$g_5 \ominus \{x\} = \mathcal{E}_G$	$g_5/x = \{rg_1, rg_2\}$	$(g_3, \{x, l_1\}) \# \{rg_1, rg_2\} = g_6$
$g_5 \uplus g_4 = g_5$	$g_4 \ominus \{x\rightarrow r\} = g_4$	$g_5/x\rightarrow r = \{\epsilon_{RG}\}$	$(g_2, \{r_2\}) \# \{\epsilon_{RG}\} = g_2$
$g_5 \uplus g_6 = g_6$	$g_4 \ominus \{x\rightarrow l\} = g_1$	$g_4/x\rightarrow r = \emptyset$	$(g_2, \{r_2\}) \# \emptyset = \mathcal{E}_G$

Remainder is empty

Quotient is empty

Data Flow Equations for Explicit Liveness Analysis: Access Graphs Version

$$In_n = \left(Out_n \ominus \text{Kill}_n(Out_n)\right) \uplus \text{Gen}_n(Out_n)$$

$$Out_n = \begin{cases} BI & n \text{ is } End \\ \biguplus_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

- In_n, Out_n, and Gen_n are access graphs
- Kill_n is a set of access paths

Flow Functions for Explicit Liveness Analysis: Access Paths Version

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{ y \rightarrow \sigma \;\middle\vert\; z \rightarrow f \rightarrow \sigma \in X, z \in A(x) \right\}$	$\bigcup\limits_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

Flow Functions for Explicit Liveness Analysis: Access Paths Version

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \mid z \rightarrow f \rightarrow \sigma \in X, z \in A(x)\right\}$	$\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

May link aliasing for soundness

Flow Functions for Explicit Liveness Analysis: Access Paths Version

Let A denote May Aliases at the exit of node n

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$\{y \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x = y.f$	$\{y \rightarrow f \rightarrow \sigma \mid x \rightarrow \sigma \in X\}$	$x \rightarrow *$
$x.f = y$	$\left\{y \rightarrow \sigma \mid z \rightarrow f \rightarrow \sigma \in X, z \in A(x)\right\}$	$\bigcup_{z \in Must(A)(x)} z \rightarrow f \rightarrow *$
$x = new$	$\emptyset$	$x \rightarrow *$
$x = null$	$\emptyset$	$x \rightarrow *$
other	$\emptyset$	$\emptyset$

May link aliasing for soundness

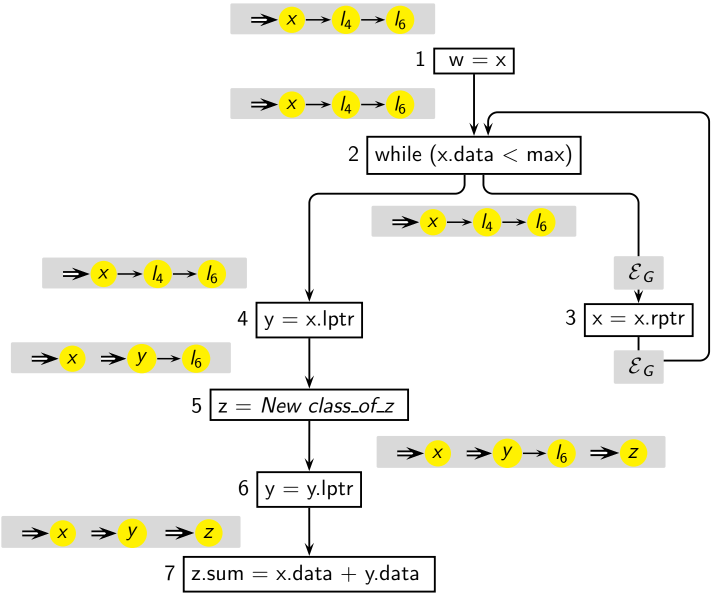
Must link aliasing for precision

Flow Functions for Explicit Liveness Analysis: Access Graphs Version

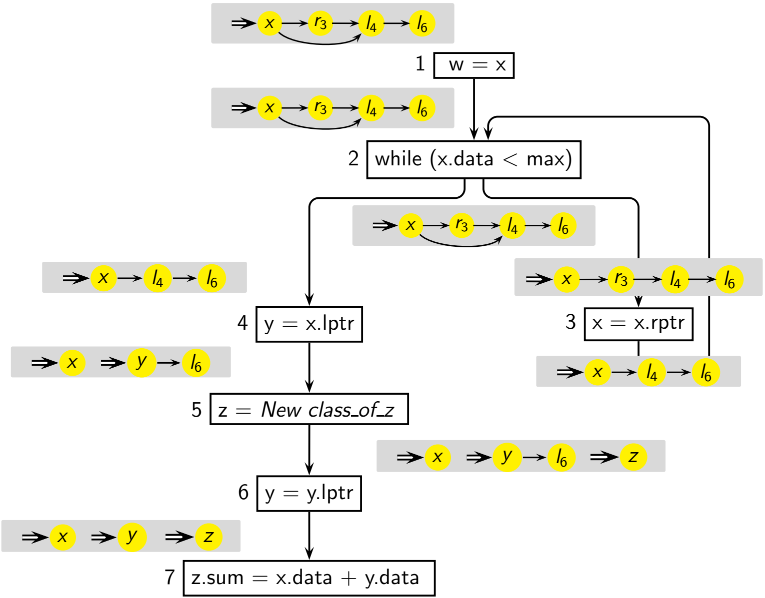
- A denotes May Aliases at the exit of node n
- $mkGraph(\rho)$ creates an access graph for access path ρ

Statement n	$\text{Gen}_n(X)$	$\text{Kill}_n(X)$
$x = y$	$mkGraph(y)\#(X/x)$	$\{x\}$
$x = y.f$	$mkGraph(y \rightarrow f)\#(X/x)$	$\{x\}$
$x.f = y$	$mkGraph(y)\#\left(\bigcup_{z \in A(x)} (X/(z \rightarrow f))\right)$	$\{z \rightarrow f \mid z \in Must(A)(x)\}$
$x = new$	$\emptyset$	$\{x\}$
$x = null$	$\emptyset$	$\{x\}$
other	$\emptyset$	$\emptyset$

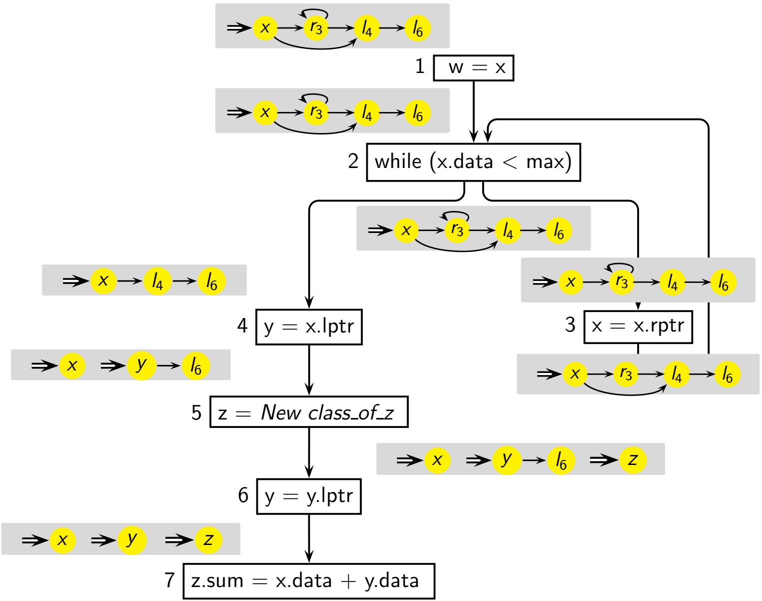
Liveness Analysis of Example Program: Ist Iteration



Liveness Analysis of Example Program: 2nd Iteration



Liveness Analysis of Example Program: 3rd Iteration



IIT Bombay
cs618: Program Analysis

Topic:

General Frameworks

Section:

Precise Modelling of General Flows

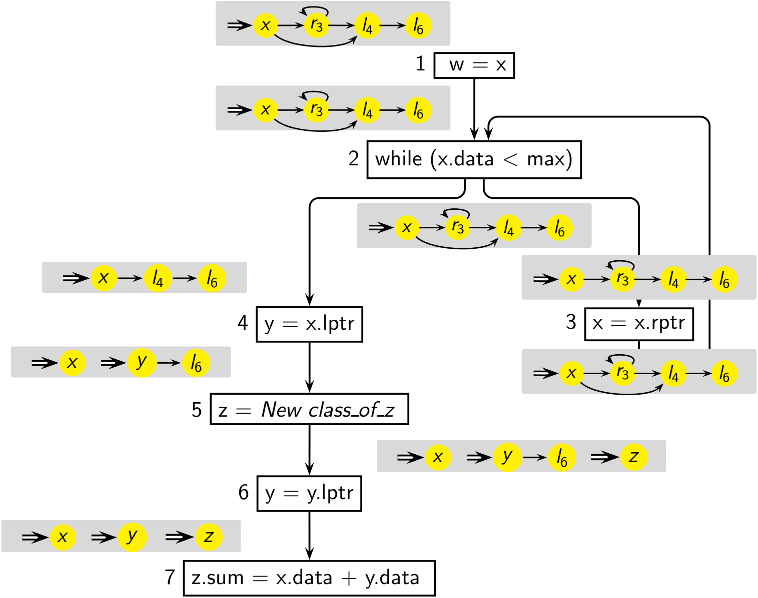
Constant Propagation

Strongly Live Variables Analysis

Pointer Analysis

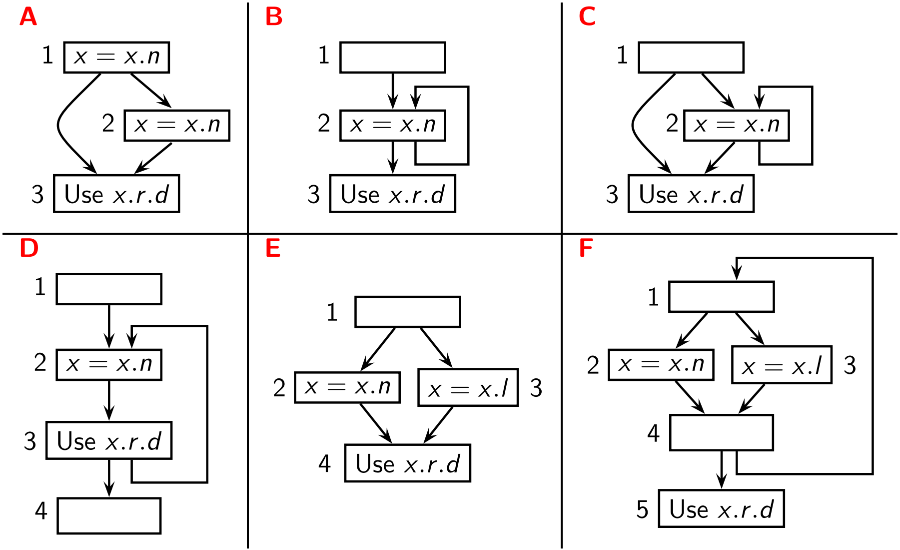
Heap Reference Analysis

Liveness Analysis of Example Program: 4th Iteration



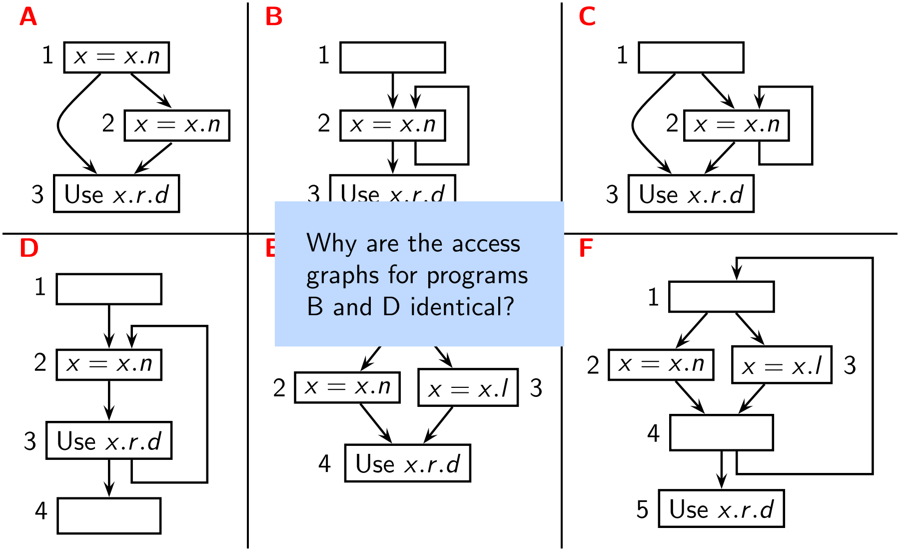
Tutorial Problem for Explicit Liveness (1)

Construct access graphs at the entry of block 1 for the following programs



Tutorial Problem for Explicit Liveness (1)

Construct access graphs at the entry of block 1 for the following programs



Why are the access graphs for programs B and D identical?

IIT Bombay
cs618: Program Analysis

Topic:

General Frameworks

Section:

Precise Modelling of General Flows

Constant Propagation

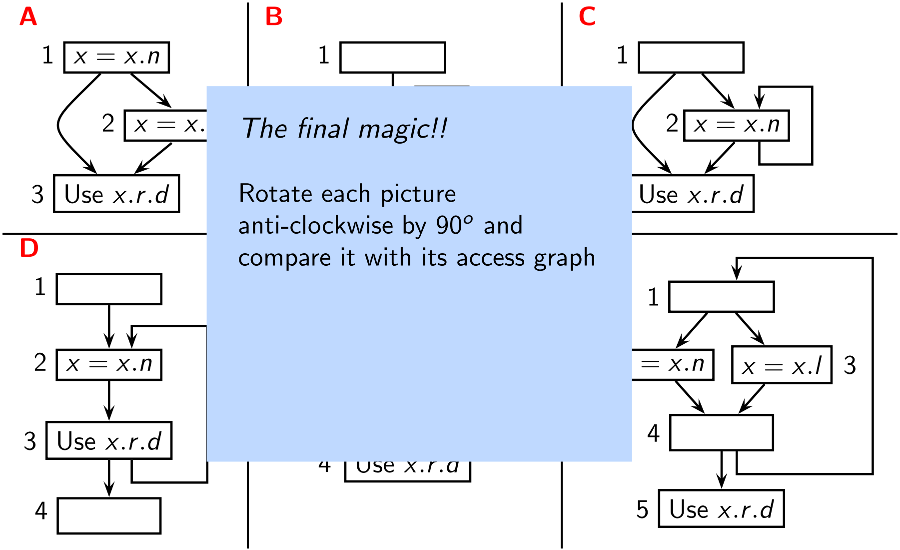
Strongly Live Variables Analysis

Pointer Analysis

Heap Reference Analysis

Tutorial Problem for Explicit Liveness (1)

Construct access graphs at the entry of block 1 for the following programs

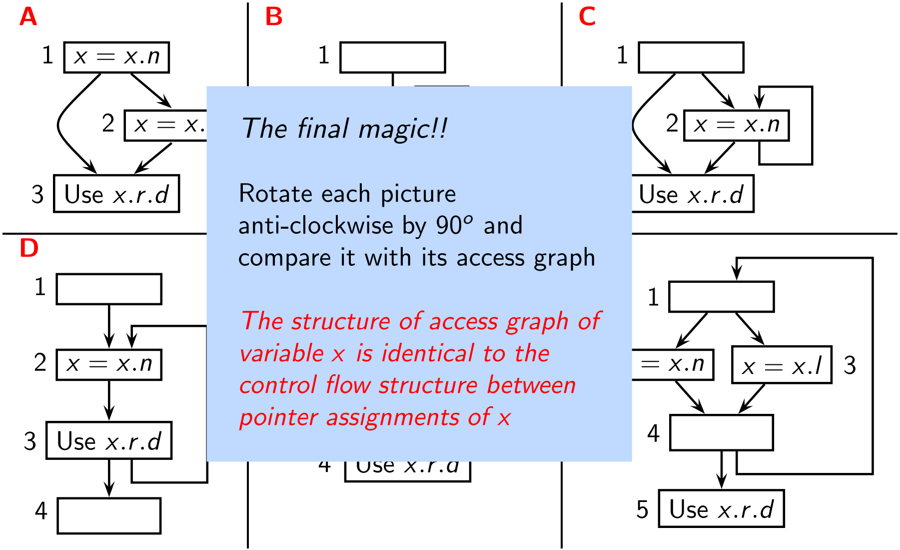


The final magic!!

Rotate each picture anti-clockwise by 90^o and compare it with its access graph

Tutorial Problem for Explicit Liveness (1)

Construct access graphs at the entry of block 1 for the following programs



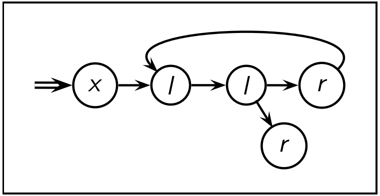
The final magic!!

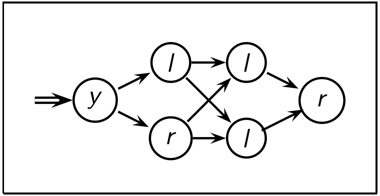
Rotate each picture anti-clockwise by 90^o and compare it with its access graph

The structure of access graph of variable x is identical to the control flow structure between pointer assignments of x

Tutorial Problem for Explicit Liveness (2)

- Unfortunately the student who constructed these access graphs forgot to attach statement numbers as subscripts to node labels and has misplaced the programs which gave rise to these graphs
- Please help her by constructing CFGs for which these access graphs represent explicit liveness at some program point in the CFGs

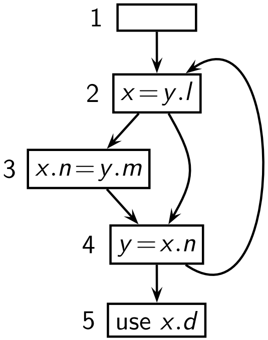




Tutorial Problem for Explicit Liveness (3)

- Compute explicit liveness for the program.
- Are the following access paths live at node 1? Show the corresponding execution sequence of statements

P1 : $y \rightarrow m \rightarrow l$
P2 : $y \rightarrow l \rightarrow n \rightarrow m$
P3 : $y \rightarrow l \rightarrow n \rightarrow l$
P4 : $y \rightarrow n \rightarrow l \rightarrow n$



Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

> Let ρ be accessible at p (i.e. available or anticipable)
> **for** each reference field f of the object pointed to by ρ
>   **if** $\rho \rightarrow f$ is not live at p **then**
>   Insert $\rho \rightarrow f = \text{null}$ at p subject to profitability

- For simple access paths, ρ is empty and f is the root variable name.

Which Access Paths Can be Nullified?

Can be safely dereferenced

- Consider extensions of accessible paths for nullification.

> Let ρ be accessible at p (i.e. available or anticipable)
> **for** each reference field f of the object pointed to by ρ
>   **if** $\rho \rightarrow f$ is not live at p **then**
>   Insert $\rho \rightarrow f =$ null at p subject to profitability

- For simple access paths, ρ is empty and f is the root variable name.

Which Access Paths Can be Nullified?

Can be safely dereferenced

Consider link aliases at p

- Consider extensions of accessible paths for nullification.

> Let ρ be accessible at p (i.e. available or anticipable)
> **for** each reference field f of the object pointed to by ρ
> **if** $\rho \rightarrow f$ is not live at p **then**
> Insert $\rho \rightarrow f = \text{null}$ at p subject to profitability

- For simple access paths, ρ is empty and f is the root variable name.

Which Access Paths Can be Nullified?

Can be safely dereferenced

Consider link aliases at p

- Consider extensions of accessible paths for nullification.

Let ρ be accessible at p (i.e. available or anticipable)
for each reference field f of the object pointed to by ρ
  if $\rho \rightarrow f$ is not live at p **then**
    Insert $\rho \rightarrow f = \text{null}$ at p subject to profitability

- For simple access paths, ρ is empty and f is the root variable name.

Cannot be hoisted and is not redefined at p

Availability and Anticipability Analyses

- ρ is available at program point p if the target of each prefix of ρ is guaranteed to be created along every control flow path reaching p.
- ρ is anticipable at program point p if the target of each prefix of ρ is guaranteed to be dereferenced along every control flow path starting at p.

Availability and Anticipability Analyses

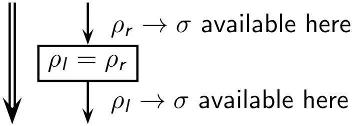
- ρ is available at program point p if the target of each prefix of ρ is guaranteed to be created along every control flow path reaching p.
- ρ is anticipable at program point p if the target of each prefix of ρ is guaranteed to be dereferenced along every control flow path starting at p.
- Finiteness.
  - An anticipable (available) access path must be anticipable (available) along every paths. Thus unbounded paths arising out of loops cannot be anticipable (available).
  - Due to "every control flow path nature", computation of anticipable and available access paths uses $\cap$ as the confluence. Thus the sets are bounded.

$\Rightarrow$ No need of access graphs.

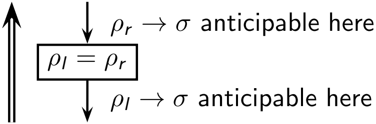
Transfer in Availability and Anticipability Analysis

The essential idea of the transfer of access paths remains same

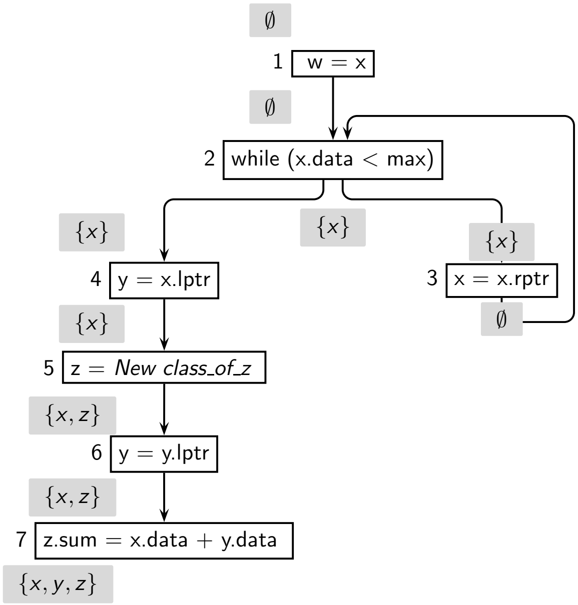
- Transfer in Availability Analysis is from the RHS to the LHS



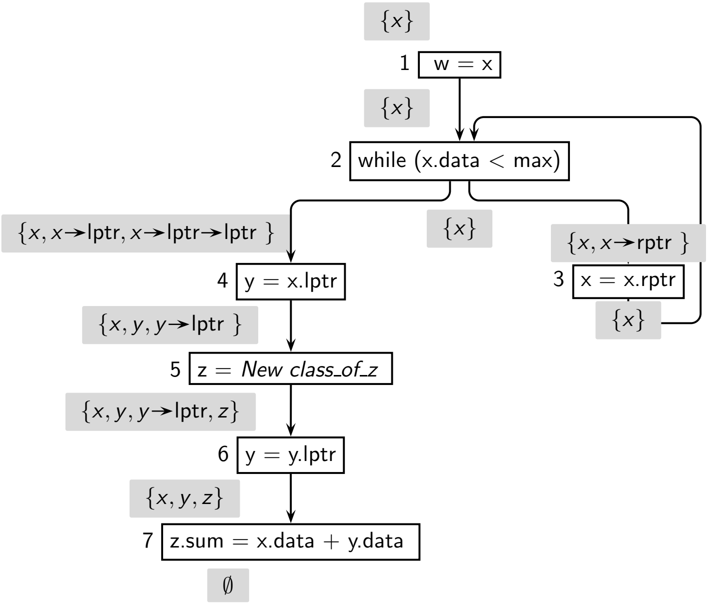
- Transfer in Anticipability Analysis is from the LHS to the RHS



Availability Analysis of Example Program



Anticipability Analysis of Example Program



IIT Bombay
cs618: Program Analysis

Topic:

General Frameworks

Section:

Precise Modelling of General Flows

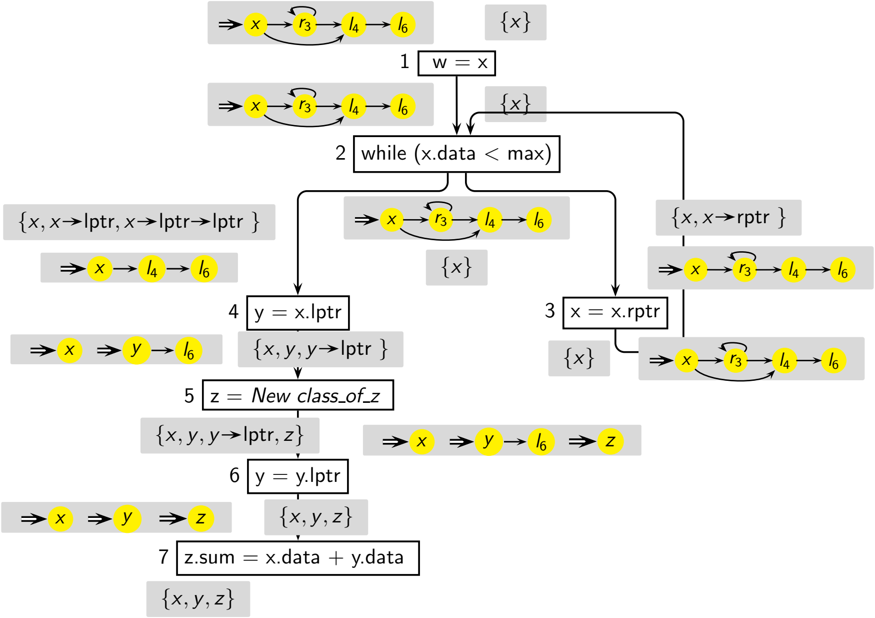
Constant Propagation

Strongly Live Variables Analysis

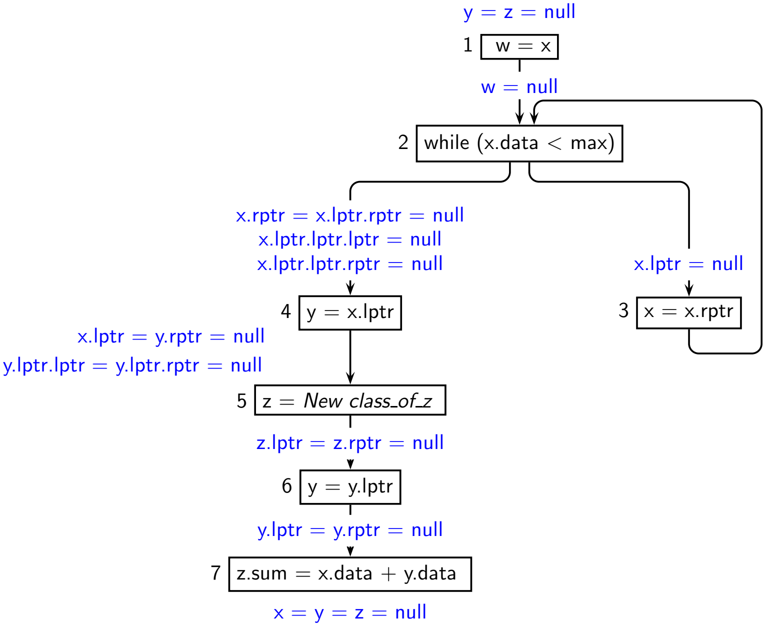
Pointer Analysis

Heap Reference Analysis

Live and Accessible Paths



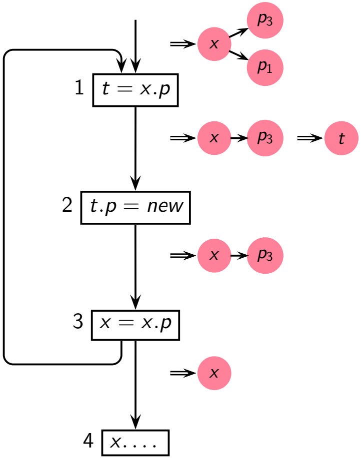
Creating null Assignments from Live and Accessible Paths



The Resulting Program

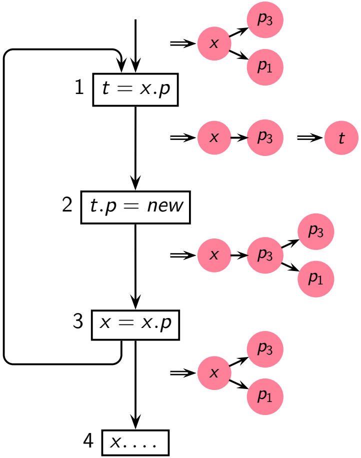
		y = z = null
1	w = x	
		w = null
2	while (x.data < max)	
	{	x.lptr = null
3	x = x.rptr }	
		x.rptr = x.lptr.rptr = null
		x.lptr.lptr.lptr = null
		x.lptr.lptr.rptr = null
4	y = x.lptr	
		x.lptr = y.rptr = null
		y.lptr.lptr = y.lptr.rptr = null
5	z = *New class_of_z*	
		z.lptr = z.rptr = null
6	y =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

Overapproximation Caused by Our Summarization



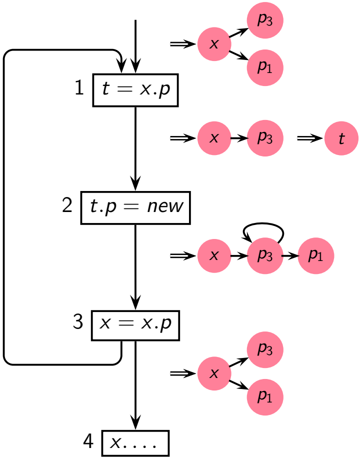
- The program allocates $x \rightarrow p$ in one iteration and uses it in the next

Overapproximation Caused by Our Summarization



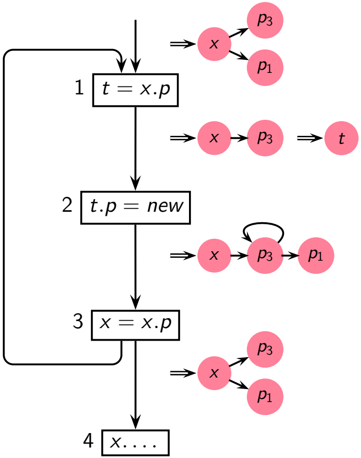
- The program allocates $x \rightarrow p$ in one iteration and uses it in the next

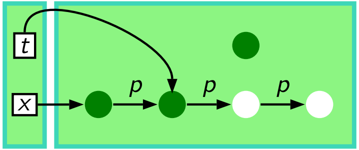
Overapproximation Caused by Our Summarization



- The program allocates $x \rightarrow p$ in one iteration and uses it in the next
- *Only $x \rightarrow p \rightarrow p$ is live at* Out_2

Overapproximation Caused by Our Summarization

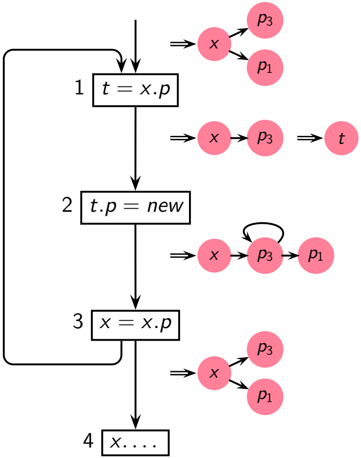


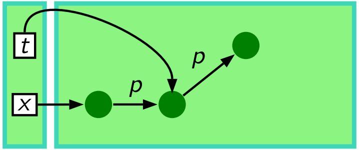


Out_1

- The program allocates $x \rightarrow p$ in one iteration and uses it in the next
- *Only* $x \rightarrow p \rightarrow p$ *is live at* Out_2

Overapproximation Caused by Our Summarization





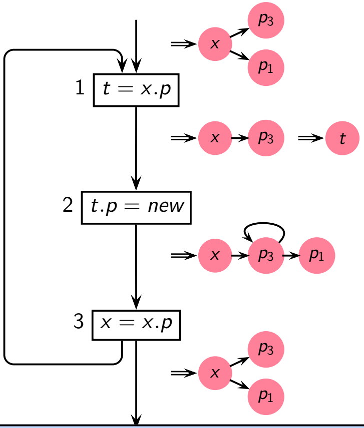
Out_2

- The program allocates $x \rightarrow p$ in one iteration and uses it in the next
- *Only $x \rightarrow p \rightarrow p$ is live at* Out_2
- $x \rightarrow p \rightarrow p$ is live at Out_2

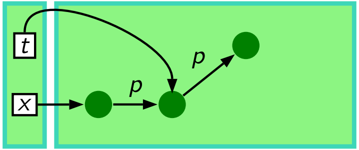
  $x \rightarrow p \rightarrow p \rightarrow p$ is dead at Out_2
- First p used in statement 3

  Second p used in statement 4
- Third p is reallocated

Overapproximation Caused by Our Summarization



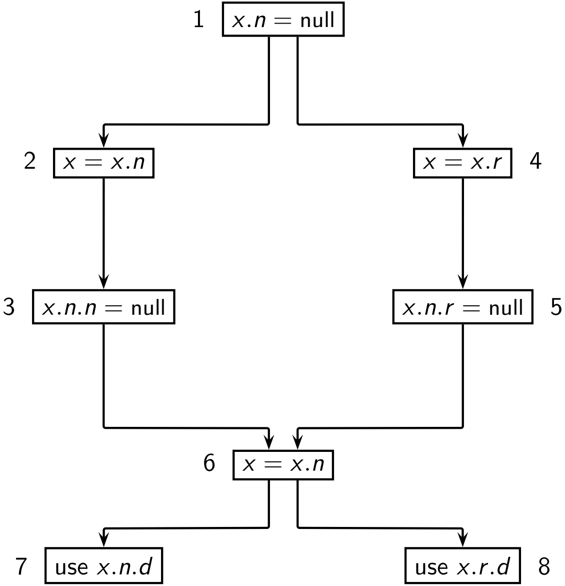
Second occurrence of a dereference does not necessarily mean an unbounded number of repetitions!



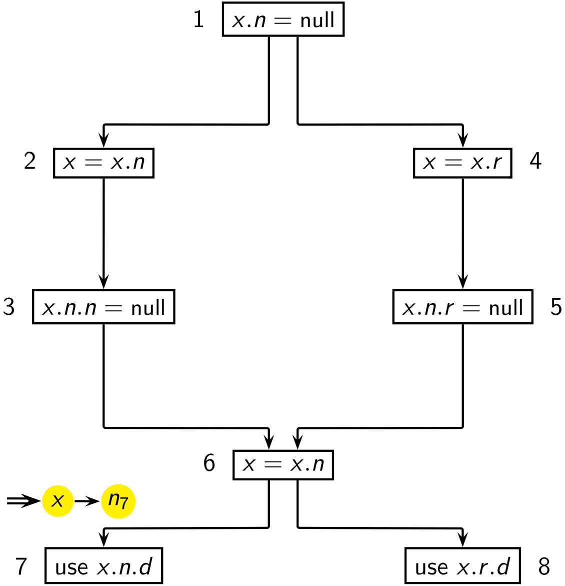
Out_2

- The program allocates $x \rightarrow p$ in one iteration and uses it in the next
- *Only $x \rightarrow p \rightarrow p$ is live at* Out_2
- $x \rightarrow p \rightarrow p$ is live at Out_2
  $x \rightarrow p \rightarrow p \rightarrow p$ is dead at Out_2
- First p used in statement 3
  Second p used in statement 4
- Third p is reallocated

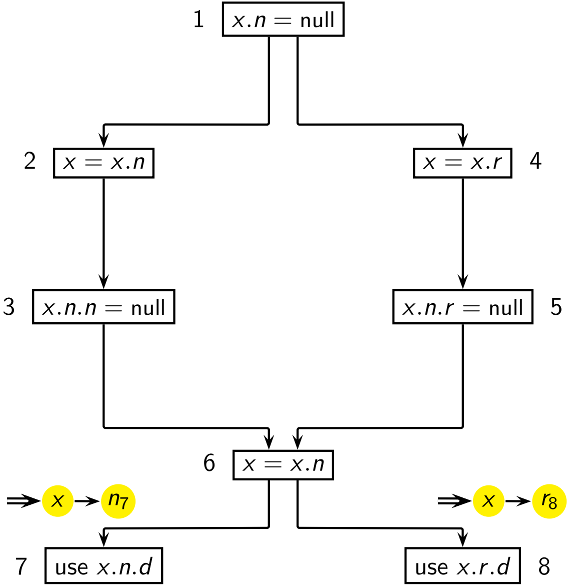
Non-Distributivity of Explicit Liveness Analysis



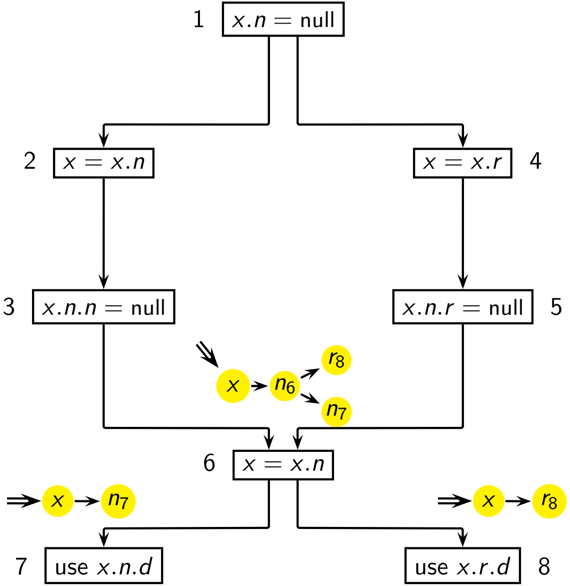
Non-Distributivity of Explicit Liveness Analysis



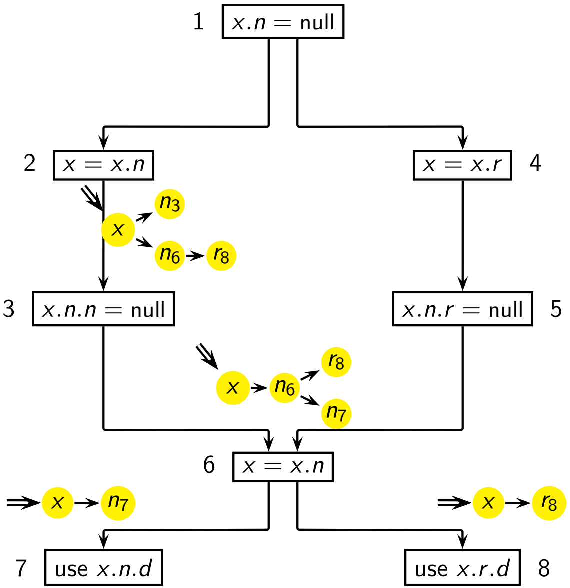
Non-Distributivity of Explicit Liveness Analysis



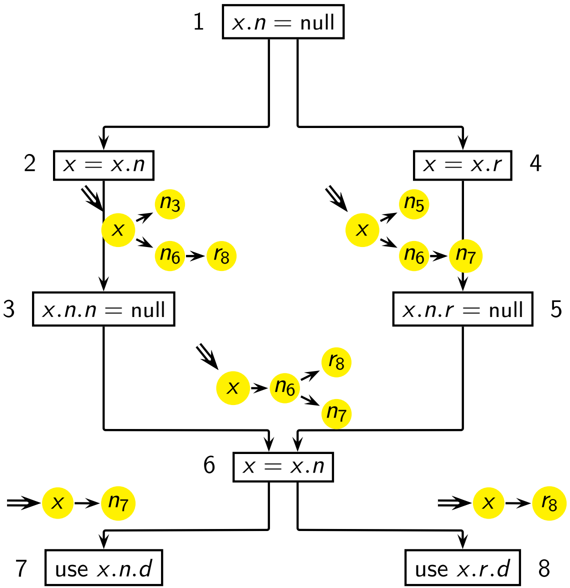
Non-Distributivity of Explicit Liveness Analysis



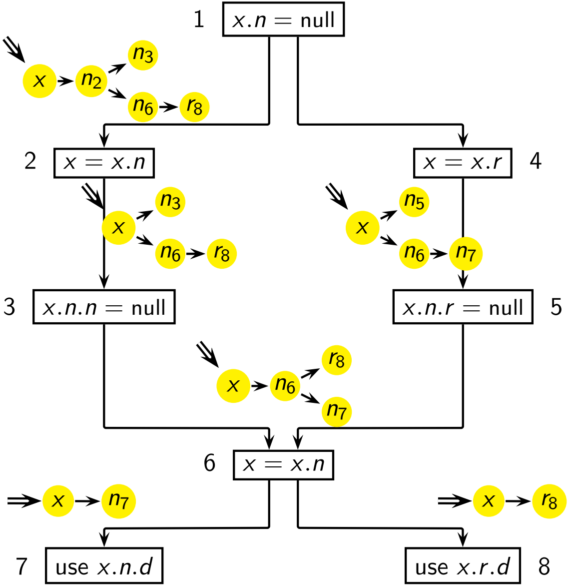
Non-Distributivity of Explicit Liveness Analysis



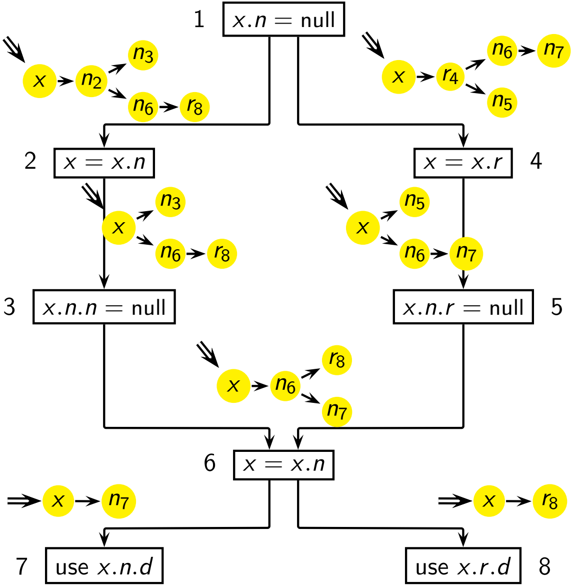
Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis

IIT Bombay
cs618: Program Analysis

Topic:

General Frameworks

Section:

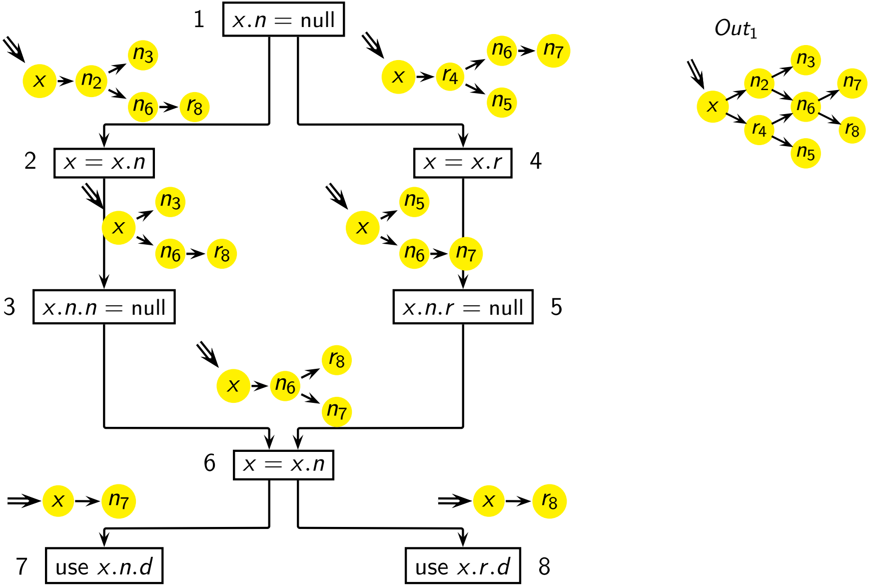
Precise Modelling of General Flows

Constant Propagation

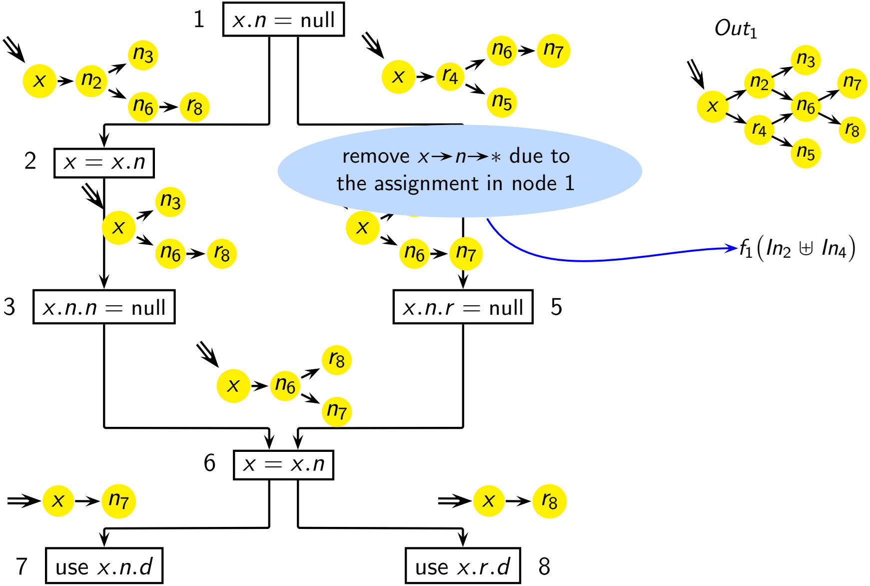
Strongly Live Variables Analysis

Pointer Analysis

Heap Reference Analysis



Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis

IIT Bombay
cs618: Program Analysis

Topic:

General Frameworks

Section:

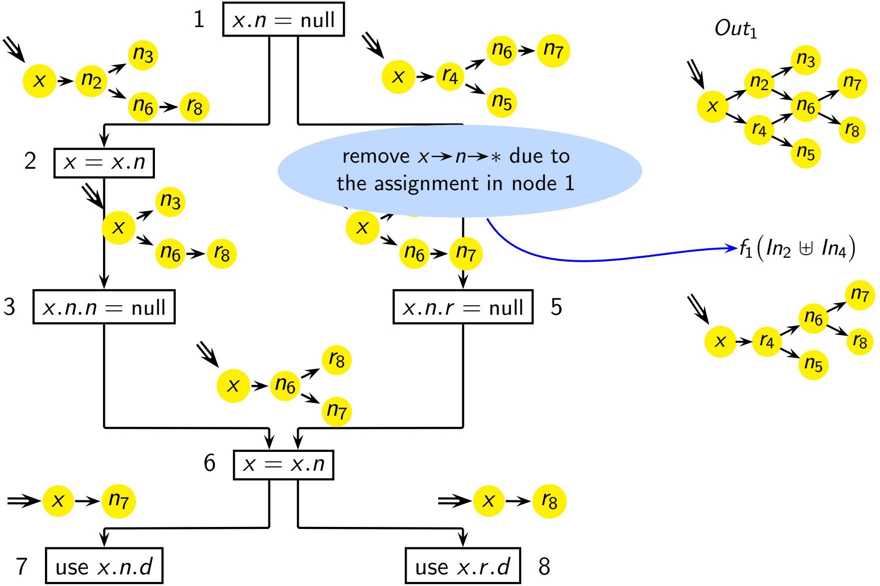
Precise Modelling of General Flows

Constant Propagation

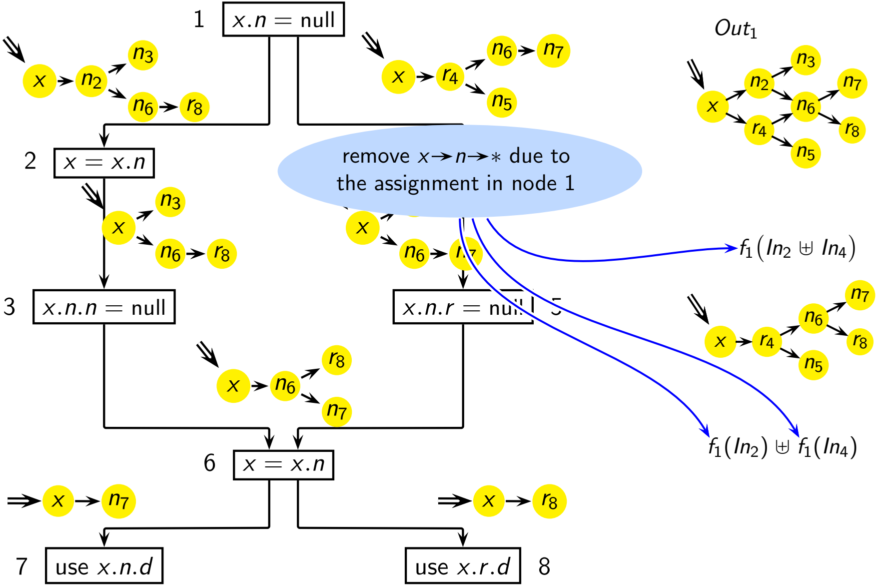
Strongly Live Variables Analysis

Pointer Analysis

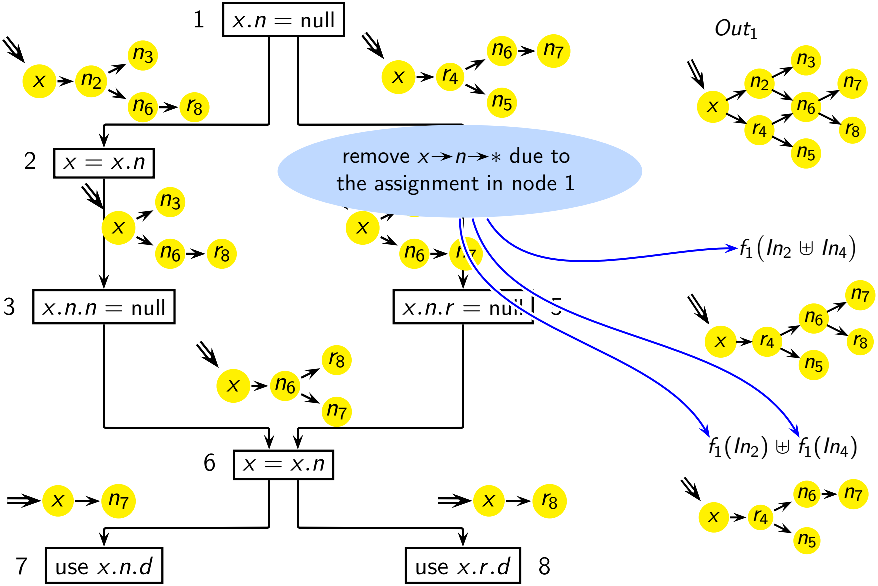
Heap Reference Analysis



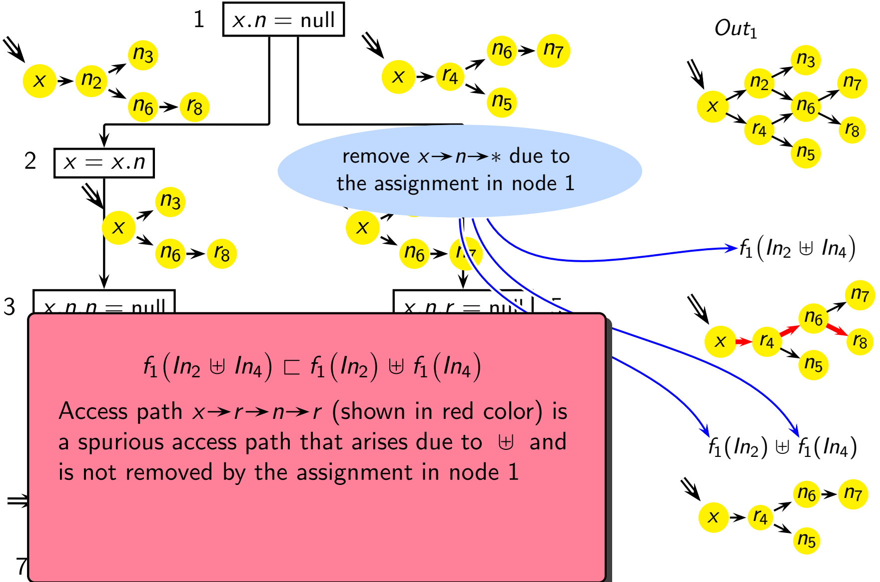
Non-Distributivity of Explicit Liveness Analysis



Non-Distributivity of Explicit Liveness Analysis



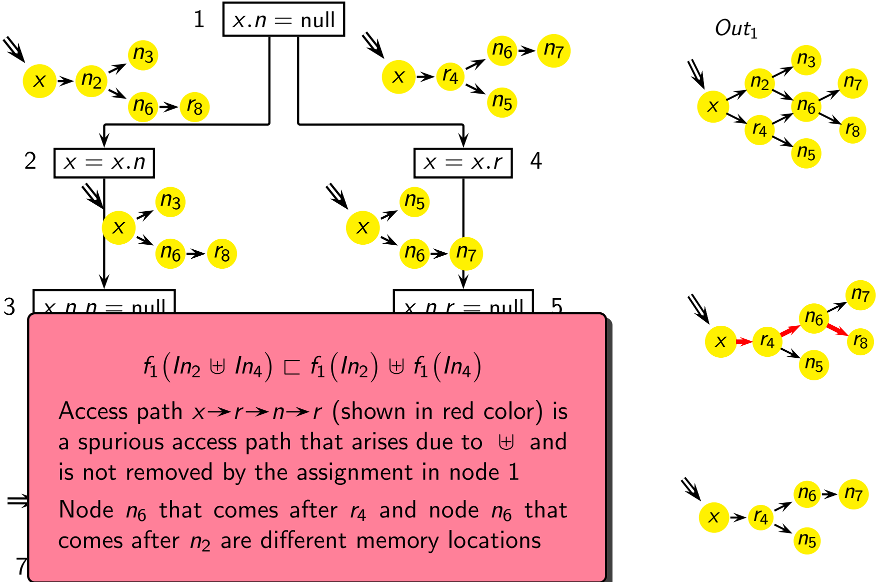
Non-Distributivity of Explicit Liveness Analysis



$$f_1(In_2 \uplus In_4) \sqsubset f_1(In_2) \uplus f_1(In_4)$$

Access path $x \rightarrow r \rightarrow n \rightarrow r$ (shown in red color) is a spurious access path that arises due to $\uplus$ and is not removed by the assignment in node 1

Non-Distributivity of Explicit Liveness Analysis



$$f_1(In_2 \uplus In_4) \sqsubset f_1(In_2) \uplus f_1(In_4)$$

Access path $x \rightarrow r \rightarrow n \rightarrow r$ (shown in red color) is a spurious access path that arises due to $\uplus$ and is not removed by the assignment in node 1

Node n_6 that comes after r_4 and node n_6 that comes after n_2 are different memory locations

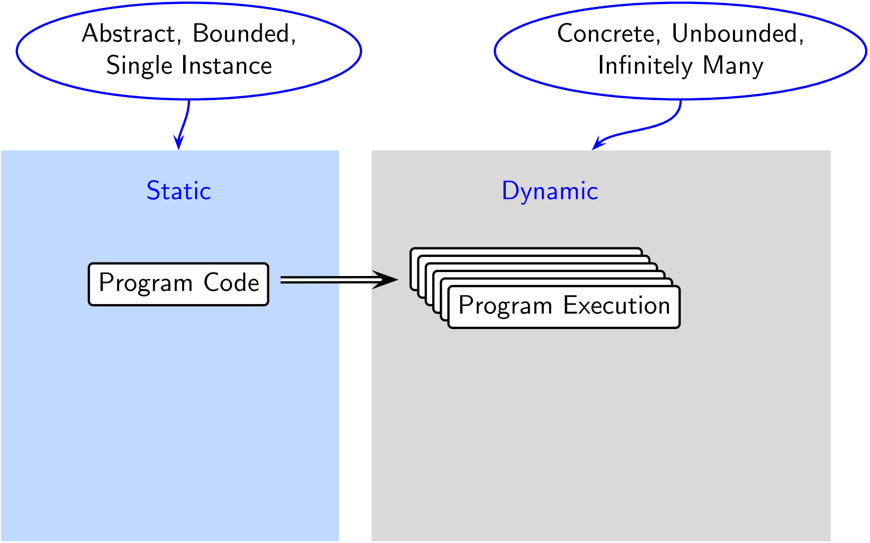
Issues Not Covered

- Precision of information
    - Cyclic Data Structures
    - Eliminating Redundant null Assignments
- Properties of Data Flow Analysis:
  Monotonicity, Boundedness, Complexity
- Interprocedural Analysis
- Extensions for C/C++
- Formulation for functional languages
- Issues that need to be researched: Good alias analysis of heap

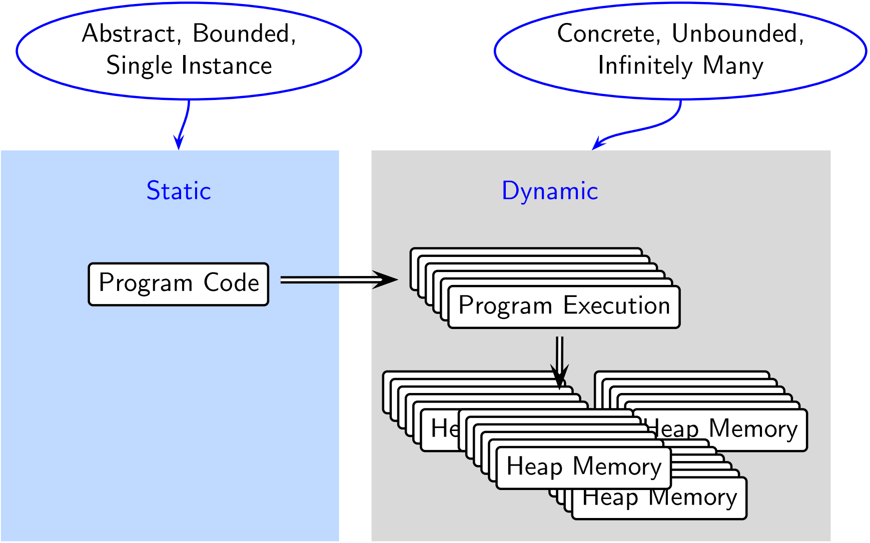
BTW, What is Static Analysis of Heap?

Static	Dynamic

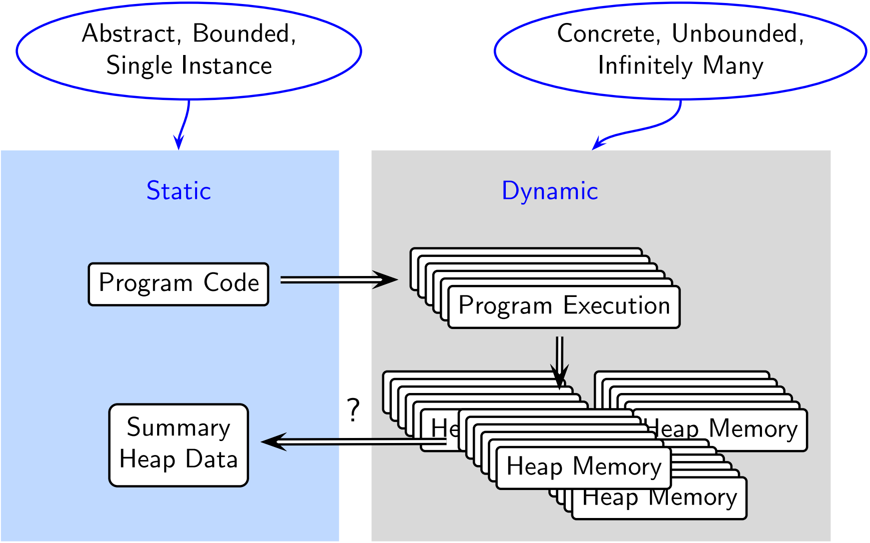
BTW, What is Static Analysis of Heap?



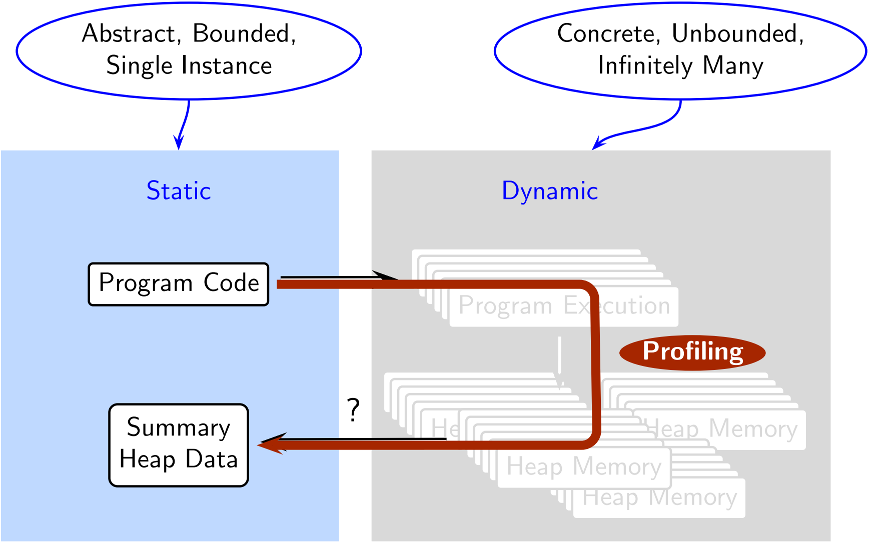
BTW, What is Static Analysis of Heap?



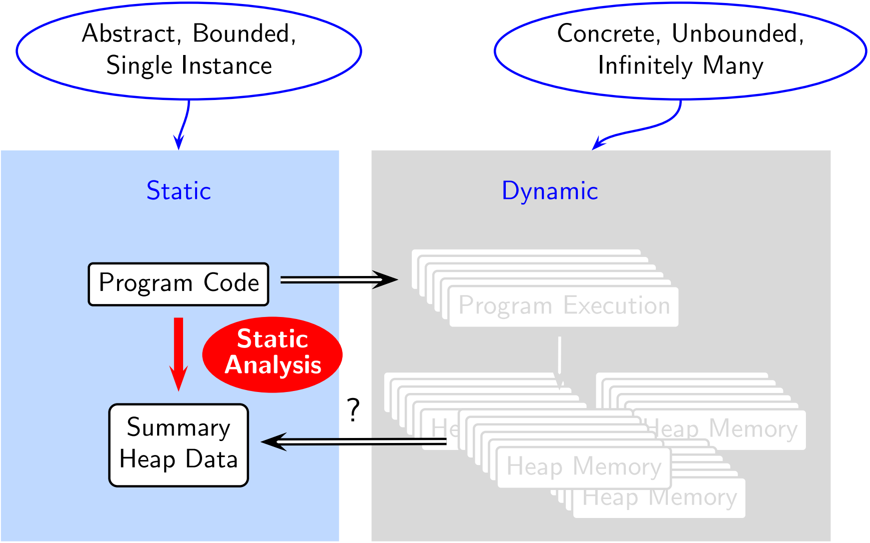
BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



BTW, What is Static Analysis of Heap?



Conclusions

- Unbounded information can be summarized using interesting insights
  - Contrary to popular perception, heap structure is not arbitrary

    Heap manipulations consist of repeating patterns which bear a close resemblance to program structure

    Analysis of heap data is possible despite the fact that the mappings between access expressions and l-values keep changing