# *Static Analysis of Programs: A Heap-Centric View*

Uday Khedker

(www.cse.iitb.ac.in/˜uday)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

5 April 2008

*Part 1*

# *Introduction*

# Copyright

A large part of these slides are based on the following material:

1. Uday Khedker, Amitabha Sanyal, and Amey Karkare. *Heap Reference Analysis Using Access Graphs*. ACM Transactions on Programming Languages and Systems, 30(1), Nov 2007.
2. Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press USA. (Under Publication)
3. Uday Khedker. *Data Flow Analysis*. In *The Compiler Design Handbook*. Editors: Priti Shankar and Y. N. Srikant. CRC Press USA. 2002.

# Outline

- Motivation: The need of heap data flow analysis
- Formulating data flow analysis
- Mathematical foundations of data flow analysis
  - The set of data flow values
  - The set of flow functions
  - Solutions of data flow analyses
  - Algorithms for performing data flow analysis
  - Complexity of data flow analysis
- Data flow analysis for heap references
- Conclusions

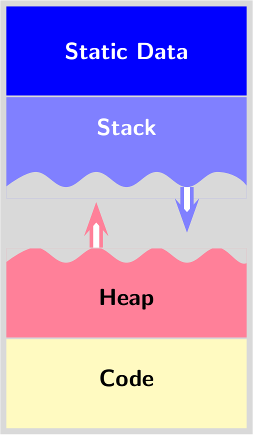
*Part 2*

# *Motivation*

# Motivation

- Heap memory allocation
- Limitations of garbage collection
- Optimization to improve garbage collection

# 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 $\equiv$ Automatic Deallocation

- Retain active data structure.
  Deallocate inactive data structure.

- What is an Active Data Structure?

# Garbage Collection $\equiv$ Automatic Deallocation

- Retain active data structure.
  Deallocate inactive data structure.

- What is an Active Data Structure?

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

# Garbage Collection ≡ Automatic Deallocation

- Retain active data structure.
  Deallocate inactive data structure.

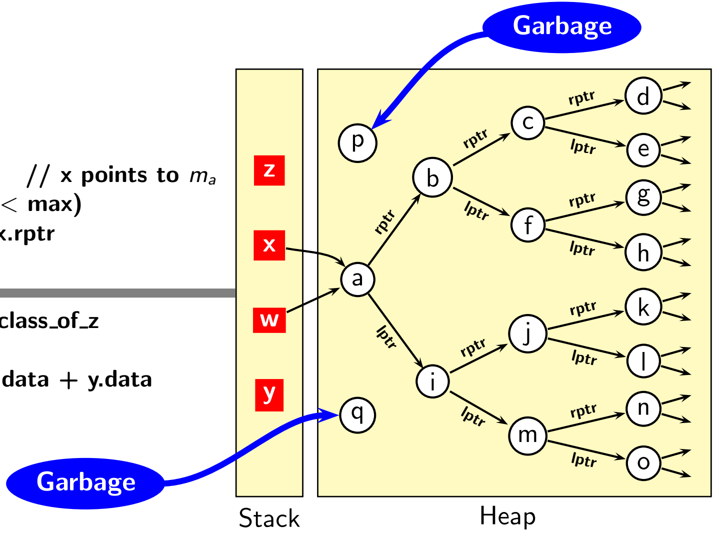
- What is an Active Data Structure?

  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?

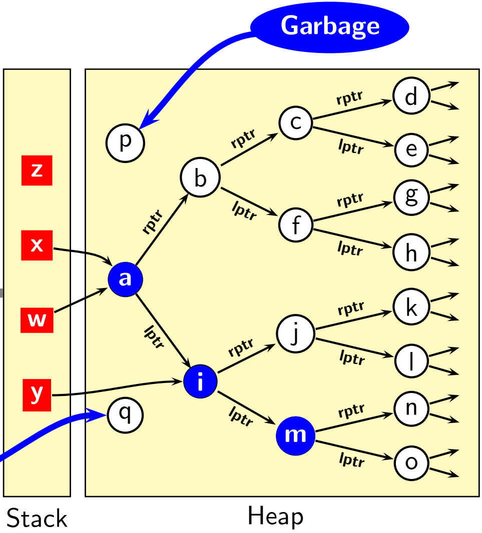
```
w = x          // x points to m_a
if (x.data < max)
        x = x.rptr
y = x.lptr

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



# What is Garbage?

False

```
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
```

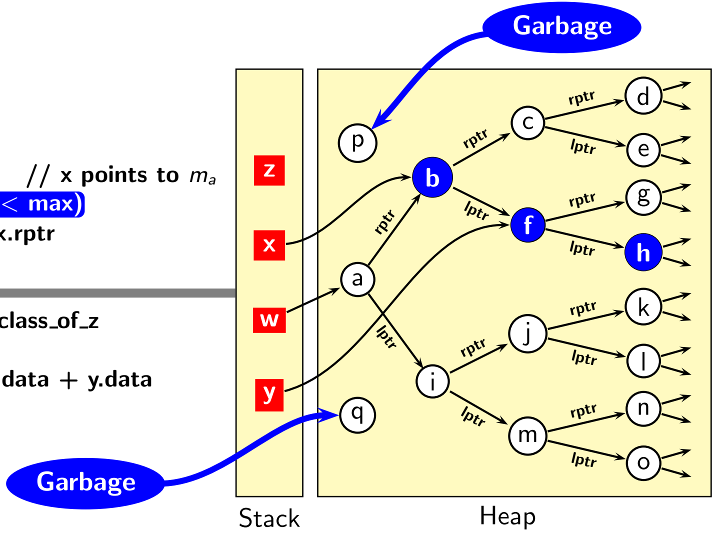



Garbage

# What is Garbage?

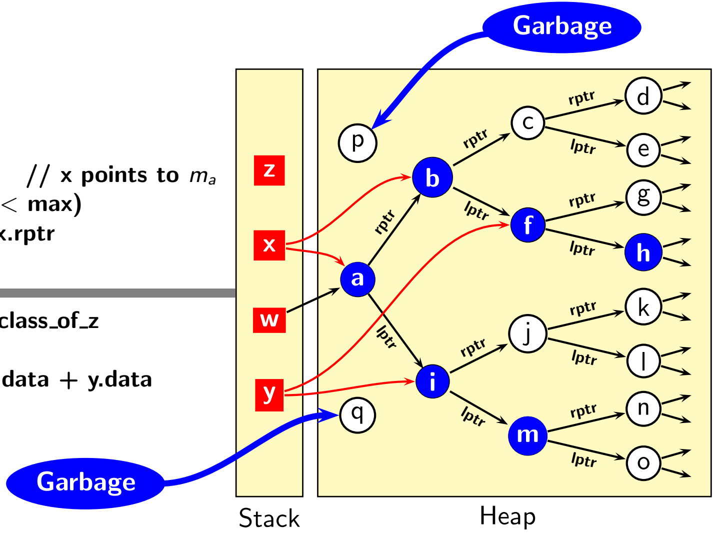
True

```
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
```



# 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.

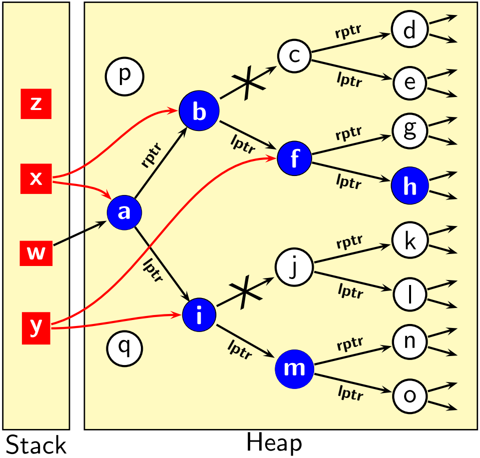
  Even with the state of art of garbage collection, 24% to 76% unused memory remains unclaimed

- Yet we have no way of distinguishing.

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

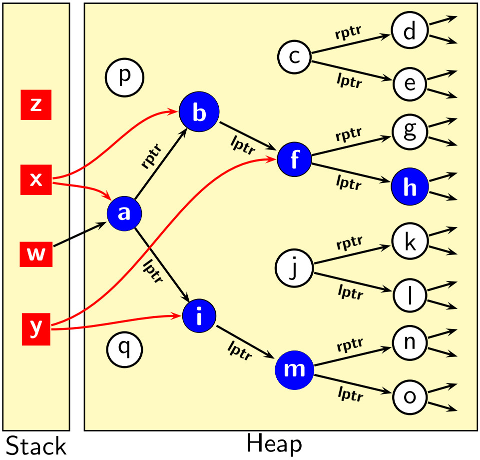
# Cedar Mesa Folk Wisdom

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



# Cedar Mesa Folk Wisdom

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



# Cedar Mesa Folk Wisdom

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

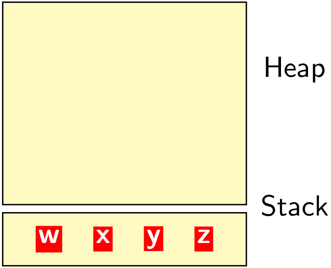
# Distinguishing Between Reachable and Live

The state of art

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

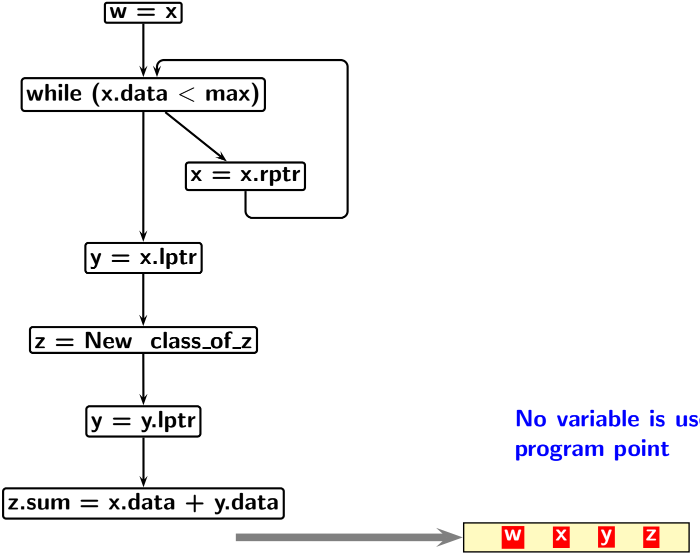
# Liveness of Stack Data

```
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
```



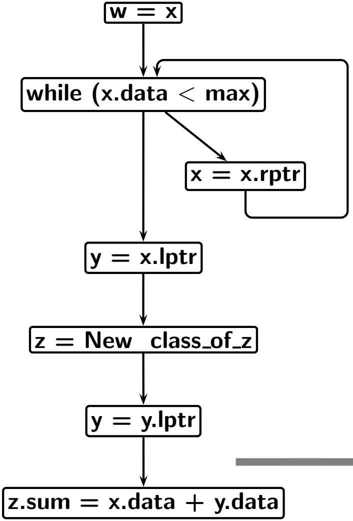

if changed to while

# Liveness of Stack Data

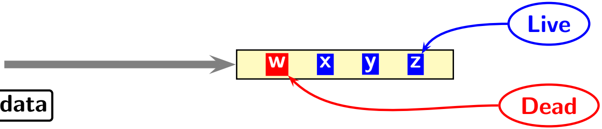



No variable is used beyond this program point

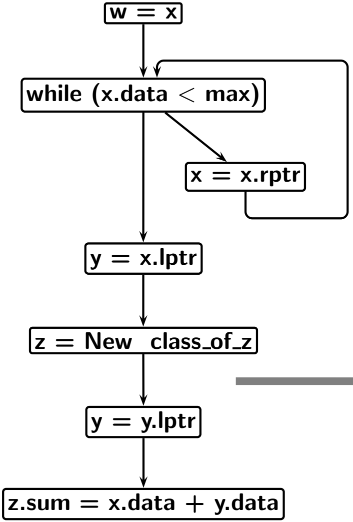
# Liveness of Stack Data



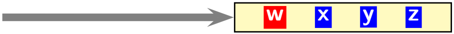

Current values of x, y, and z are used beyond this program point



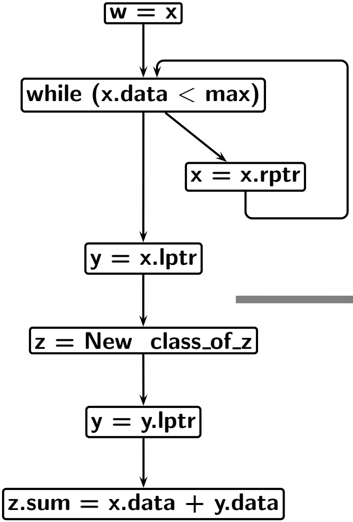
# Liveness of Stack Data

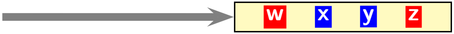



- Current values of x, y, and z are used beyond this program point
- The value of y is different before and after the assignment to y



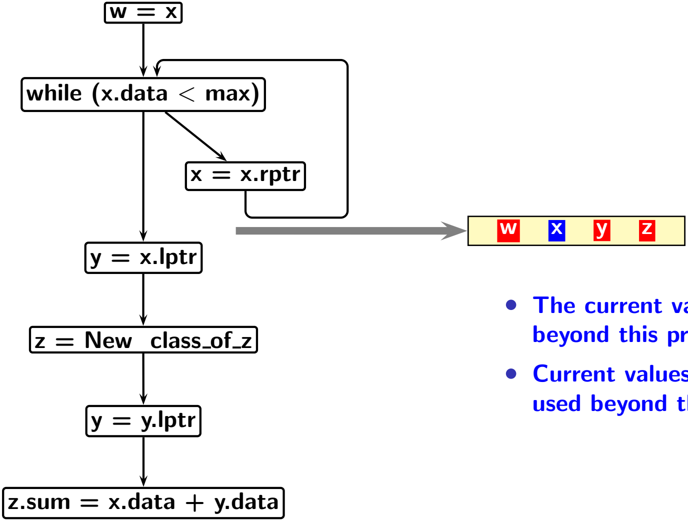
# Liveness of Stack Data



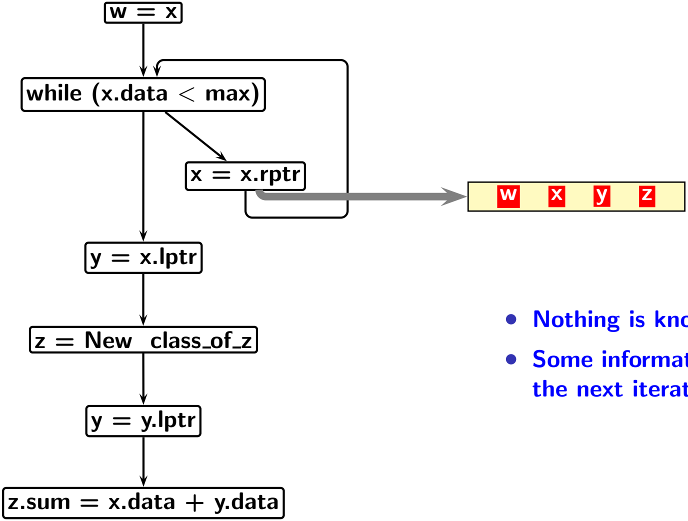

- The current values of x and y are used beyond this program point
- The current value of z is not used beyond this program point

# Liveness of Stack Data



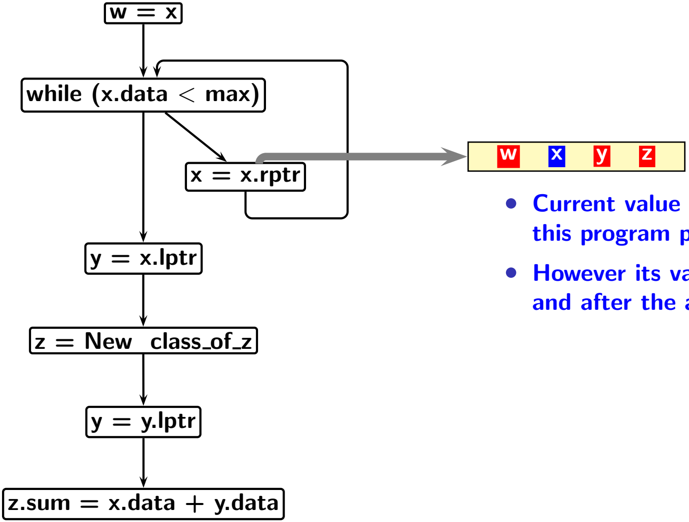

- The current values of x is used beyond this program point
- Current values of y and z are not used beyond this program point

# Liveness of Stack Data



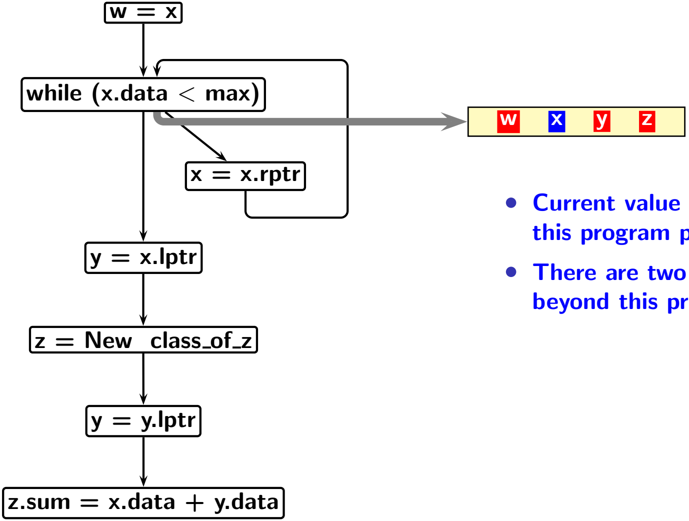

- Nothing is known as of now
- Some information will be available in the next iteration point

# Liveness of Stack Data



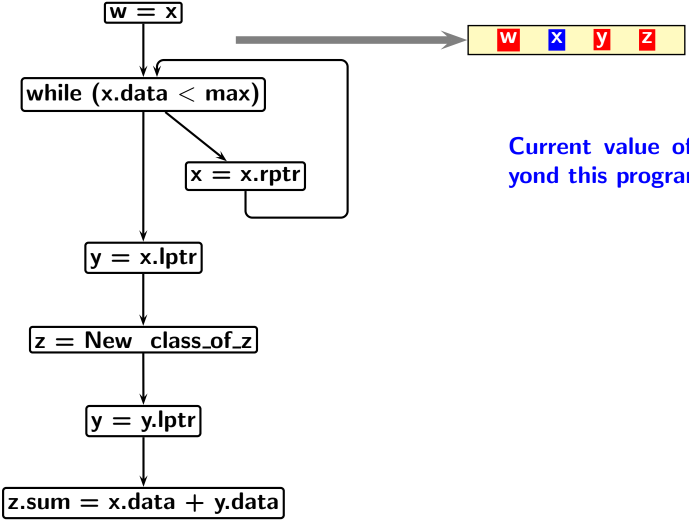

- Current value of x is used beyond this program point
- However its value is different before and after the assignment

# Liveness of Stack Data



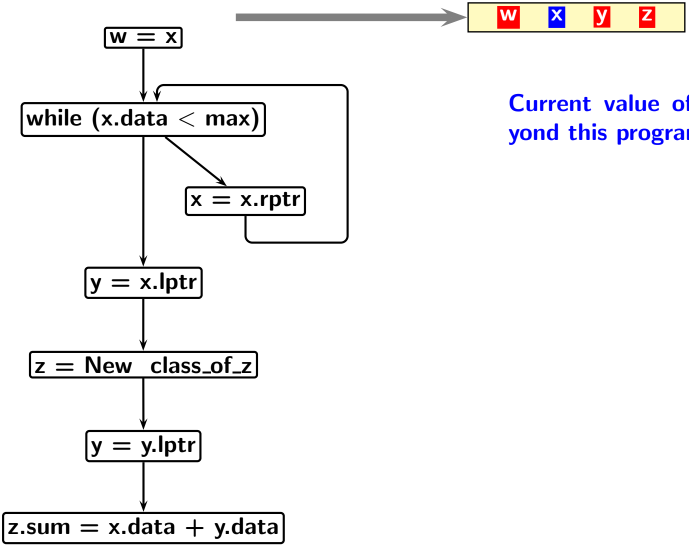

- Current value of x is used beyond this program point
- There are two control flow paths beyond this program point

# Liveness of Stack Data



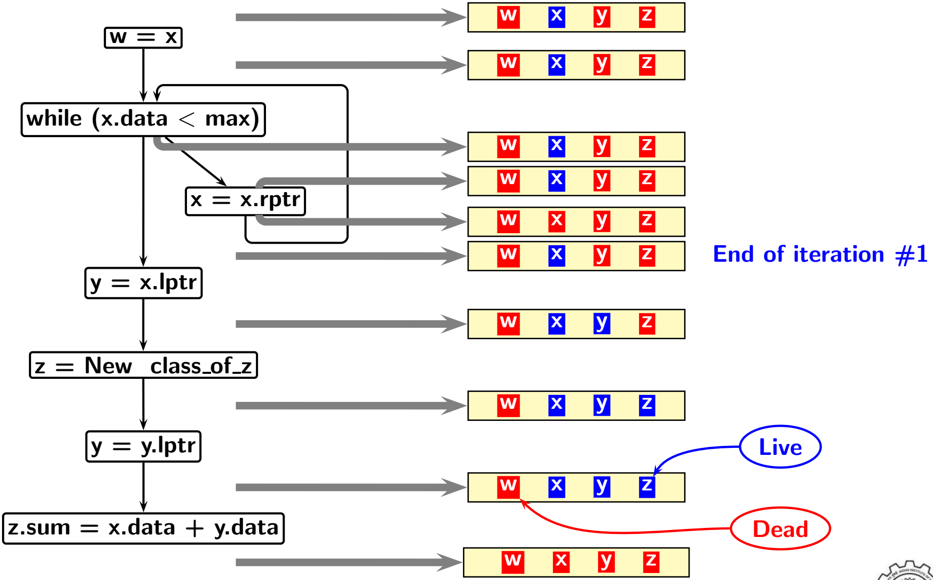

Current value of x is used beyond this program point

# Liveness of Stack Data

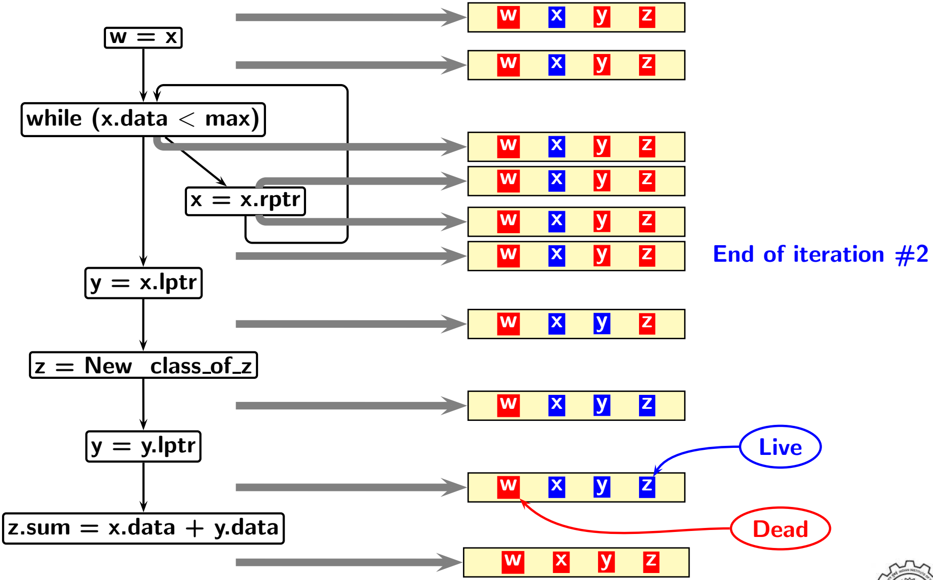



Current value of x is used beyond this program point

# Liveness of Stack Data



# Liveness of Stack Data

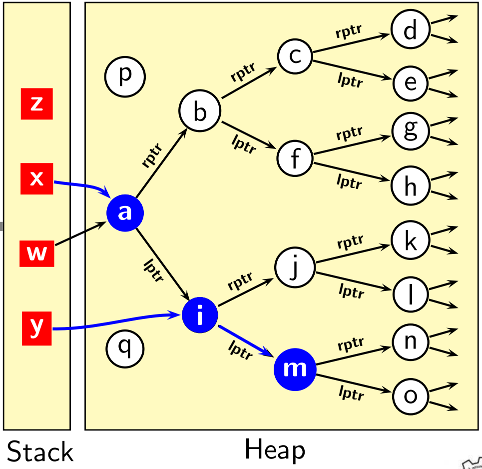


# Applying Cedar Mesa Folk Wisdom to Heap Data

## 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
```

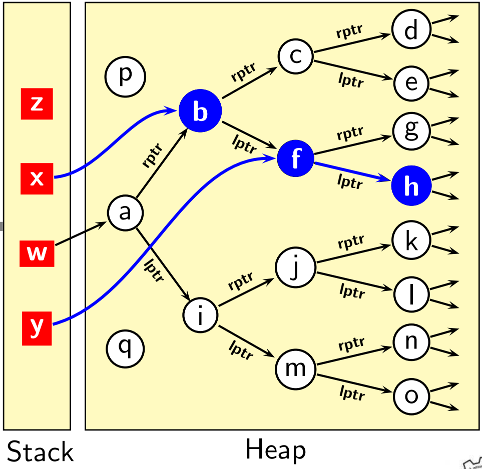


# Applying Cedar Mesa Folk Wisdom to Heap Data

## 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
```

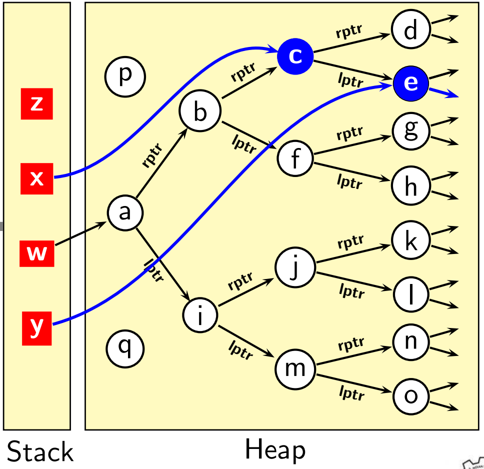


# Applying Cedar Mesa Folk Wisdom to Heap Data

## 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?

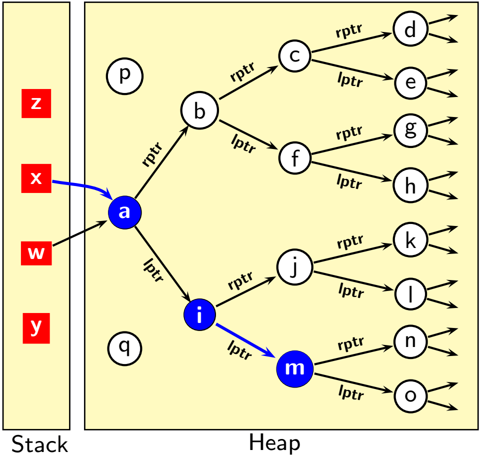
# Our Solution

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

# Our Solution

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

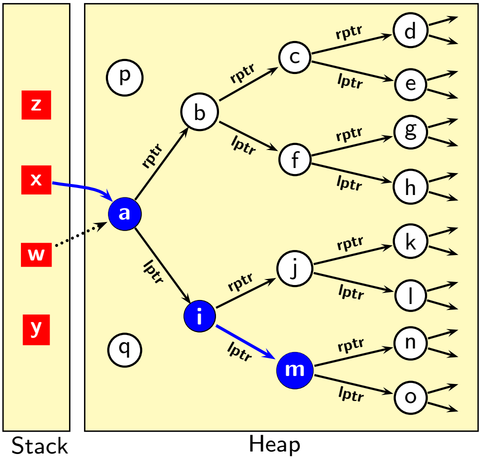
While loop is not executed even once



# Our Solution

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

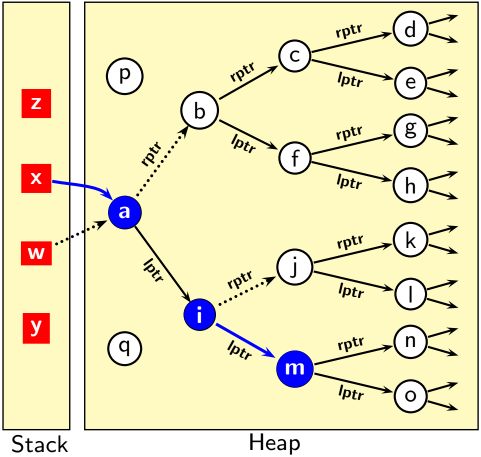
While loop is not executed even once



# Our Solution

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

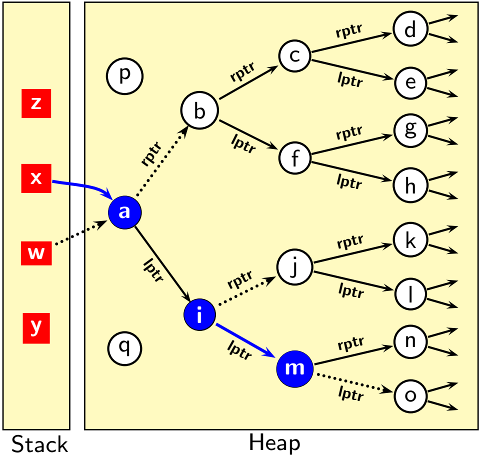
While loop is not executed even once



# Our Solution

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

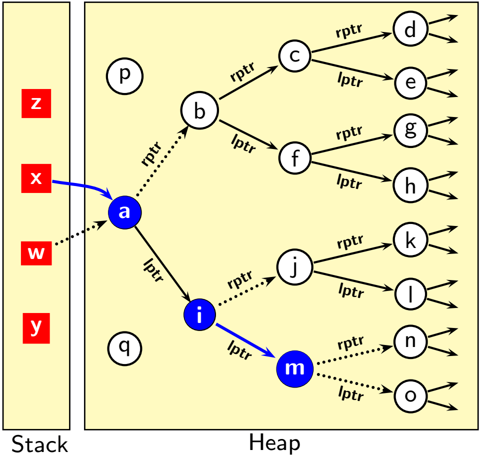
While loop is not executed even once



# Our Solution

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

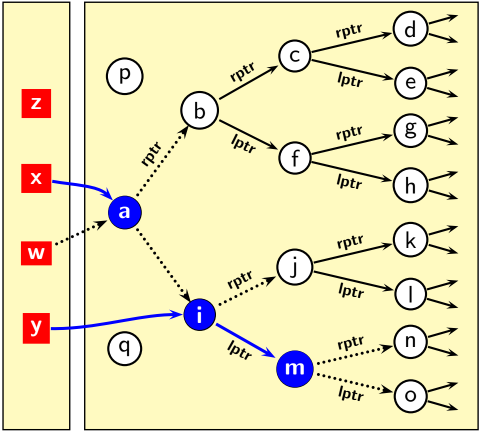
While loop is not executed even once



# Our Solution

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

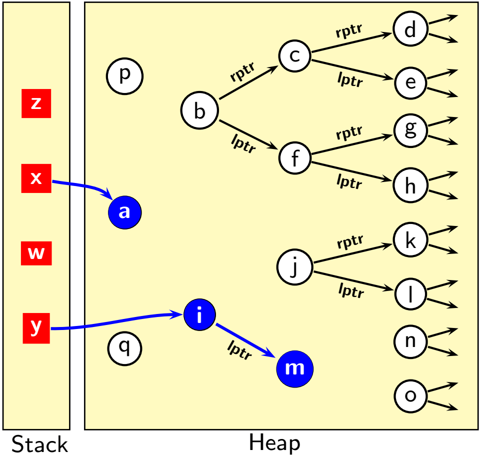
While loop is not executed even once



# Our Solution

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

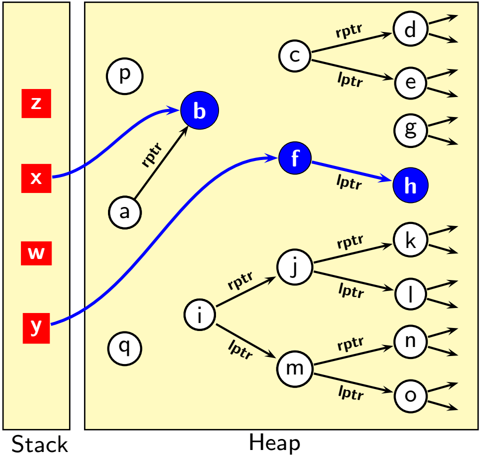
While loop is not executed even once



# Our Solution

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

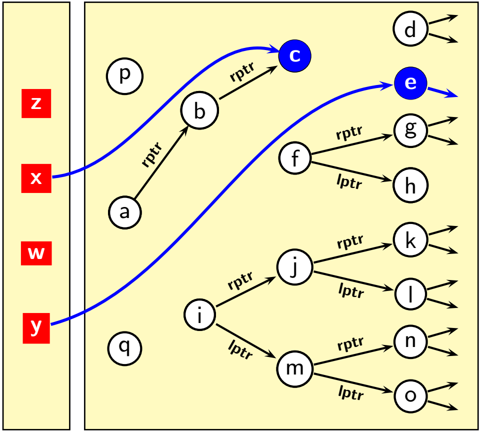
While loop is executed once



# Our Solution

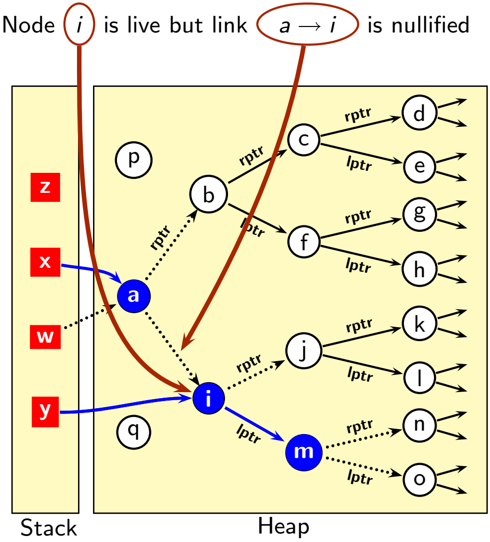
```
    y = z = null
1   w = x
    w = null
2   while (x.data < max)
    {       x.lptr = null
3           x = x.rptr       }
    x.rptr = x.lptr.rptr = null
    x.lptr.lptr.lptr = null
    x.lptr.lptr.rptr = null
4   y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null
5   z =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

While loop is executed twice



# Some Observations

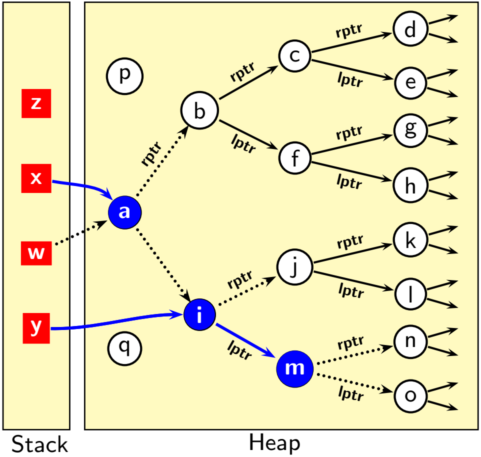
```
  y = z = null
1 w = x
  w = null
2 while (x.data < max)
  {      x.lptr = null
3        x = x.rptr      }
  x.rptr = x.lptr.rptr = null
  x.lptr.lptr.lptr = null
  x.lptr.lptr.rptr = null
4 y = x.lptr
  x.lptr = y.rptr = null
  y.lptr.lptr = y.lptr.rptr = null
5 z = New class_of_z
  z.lptr = z.rptr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



# Some Observations

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

New access expressions are created.
Can they cause exceptions?

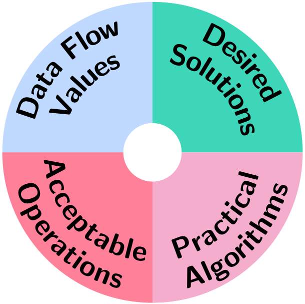


# Goals of This Tutorial

- Main Goal: Explain how to get the modified program
- Side goals:

# Goals of This Tutorial

- Main Goal: Explain how to get the modified program
- Side goals:




- Explain the intuitions
- Relate them to mathematical rigour
- Highlight the frontiers

*Part 3*

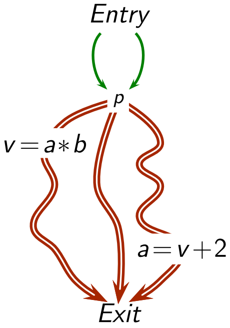
# *Formulating Data Flow Analysis*

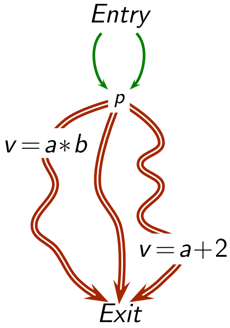
# Formulating Data Flow Analysis

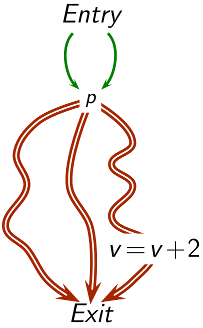
- Live variables analysis and available expressions analysis
- Local and global data flow properties
- Common form of data flow equations

# Live Variables Analysis

A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$.



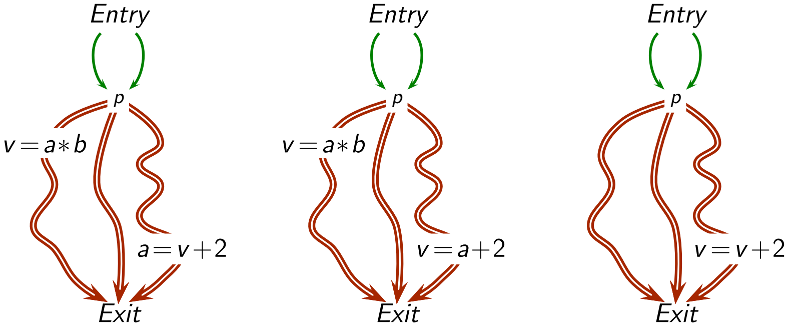



# Live Variables Analysis

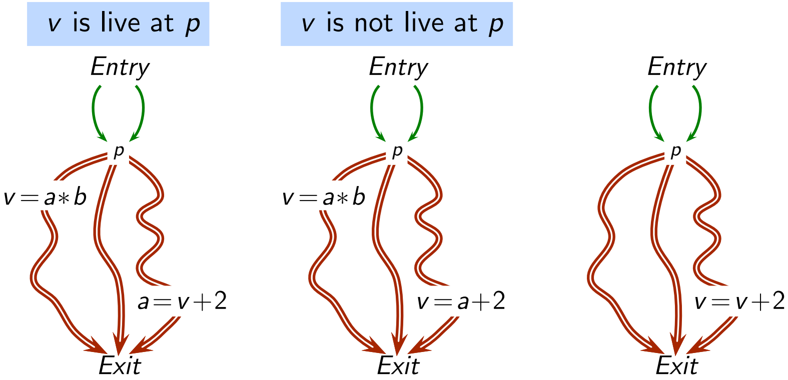
A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$.

$v$ is live at $p$



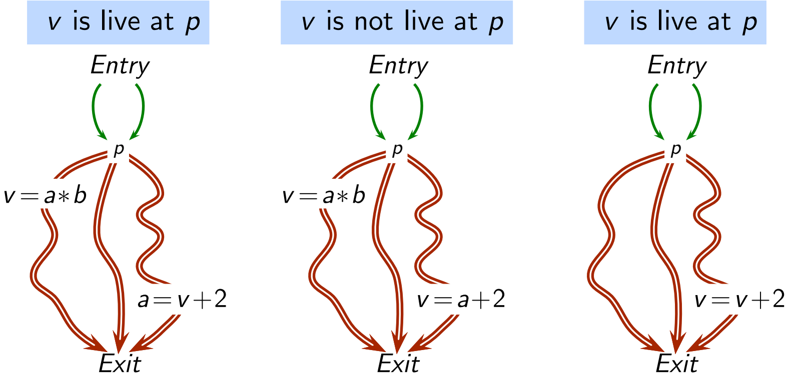
# Live Variables Analysis

A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$.



# Live Variables Analysis

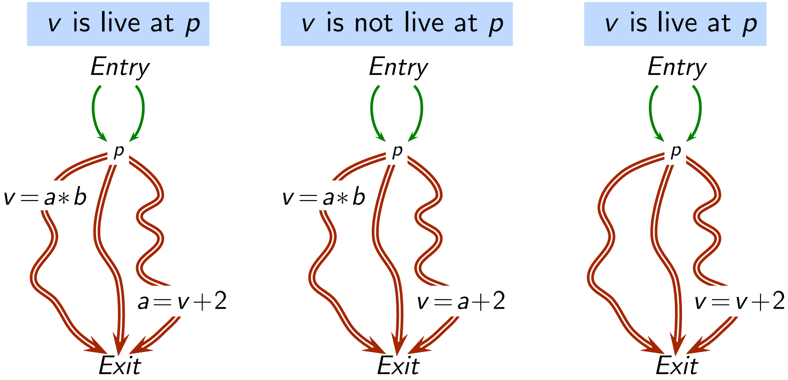
A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$.



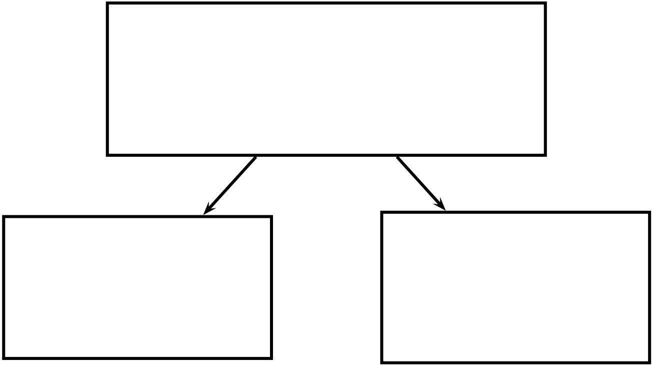
# Live Variables Analysis

A variable $v$ is live at a program point $p$, if some path from $p$ to program exit contains an r-value occurrence of $v$ which is not preceded by an l-value occurrence of $v$.

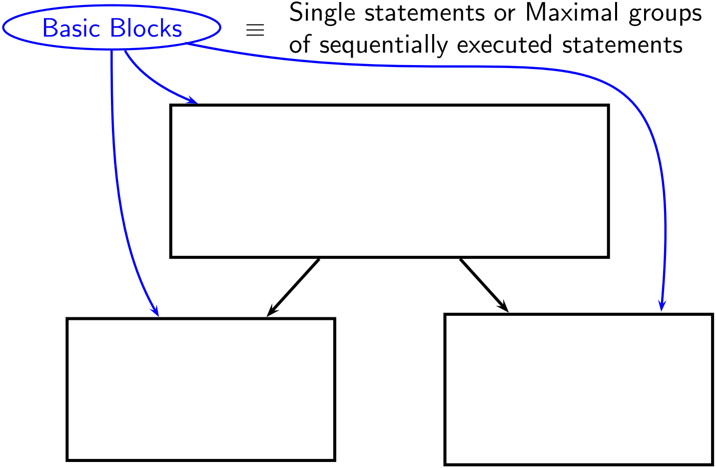
Path based specification



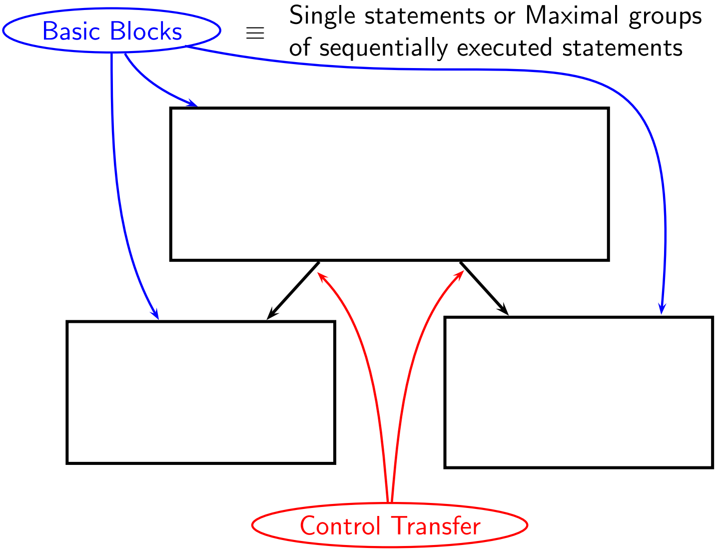
# Defining Data Flow Analysis for Live Variables Analysis



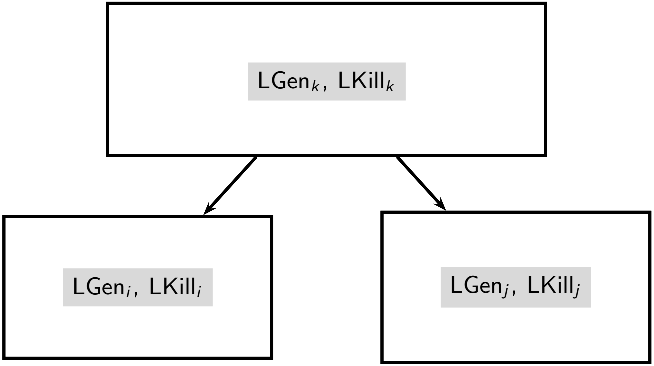
# Defining Data Flow Analysis for Live Variables Analysis



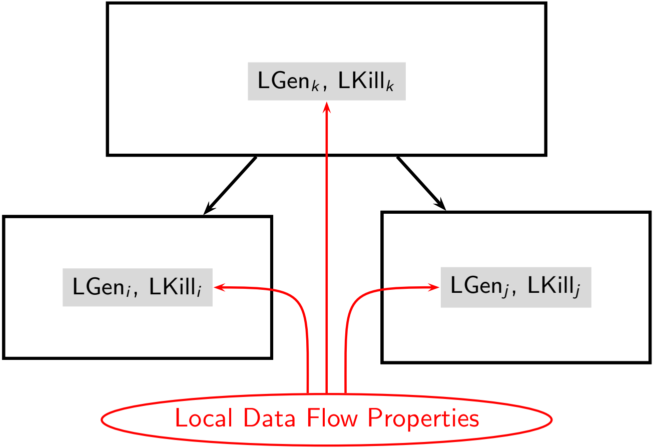
# Defining Data Flow Analysis for Live Variables Analysis



# Defining Data Flow Analysis for Live Variables Analysis



# Defining Data Flow Analysis for Live Variables Analysis



# Local Data Flow Properties for Live Variables Analysis

$$\begin{aligned}
\mathsf{LGen}_b &= \{\ v \mid \text{variable } v \text{ is used in basic block } b \text{ and} \\
&\qquad\quad \text{is not preceded by a definition of } v\ \} \\
\mathsf{LKill}_b &= \{\ v \mid \text{basic block } b \text{ contains a definition of } v\ \}
\end{aligned}$$

# Local Data Flow Properties for Live Variables Analysis

r-value occurrence

Value is only read, e.g. x,y,z in

x.sum = y.data + z.data

$$\text{LGen}_b = \{\, v \mid \text{variable } v \text{ is used in basic block } b \text{ and is not preceded by a definition of } v \,\}$$

$$\text{LKill}_b = \{\, v \mid \text{basic block } b \text{ contains a definition of } v \,\}$$

# Local Data Flow Properties for Live Variables Analysis

r-value occurrence

Value is only read, e.g. x,y,z in

x.sum = y.data + z.data

l-value occurrence

Value is modified e.g. y in

y = x.lptr

$$\begin{aligned} \text{LGen}_b &= \{\, v \mid \text{variable } v \text{ is used in basic block } b \text{ and} \\ &\qquad \text{is not preceded by a definition of } v \,\} \\ \text{LKill}_b &= \{\, v \mid \text{basic block } b \text{ contains a definition of } v \,\} \end{aligned}$$

# Local Data Flow Properties for Live Variables Analysis

l-value occurrence

Value is modified e.g. y in

y = x.lptr

r-value occurrence

Value is only read, e.g. x,y,z in

x.sum = y.data + z.data

$$\begin{aligned}
\text{LGen}_b &= \{\, v \mid \text{variable } v \text{ is used in basic block } b \text{ and} \\
&\qquad \text{is not preceded by a definition of } v \,\} \\
\text{LKill}_b &= \{\, v \mid \text{basic block } b \text{ contains a definition of } v \,\}
\end{aligned}$$

within $b$

# Local Data Flow Properties for Live Variables Analysis

r-value occurrence

Value is only read, e.g. x,y,z in

x.sum = y.data + z.data

l-value occurrence

Value is modified e.g. y in

y = x.lptr

$$\text{LGen}_b = \{\, v \mid \text{variable } v \text{ is } \boxed{\text{used}} \text{ in basic block } b \text{ and is not } \boxed{\text{preceded}} \text{ by a } \boxed{\text{definition}} \text{ of } v \,\}$$

$$\text{LKill}_b = \{\, v \mid \text{basic block } b \boxed{\text{contains}} \text{ a definition of } v \,\}$$

within $b$

anywhere in $b$

# Defining Data Flow Analysis for Live Variables Analysis

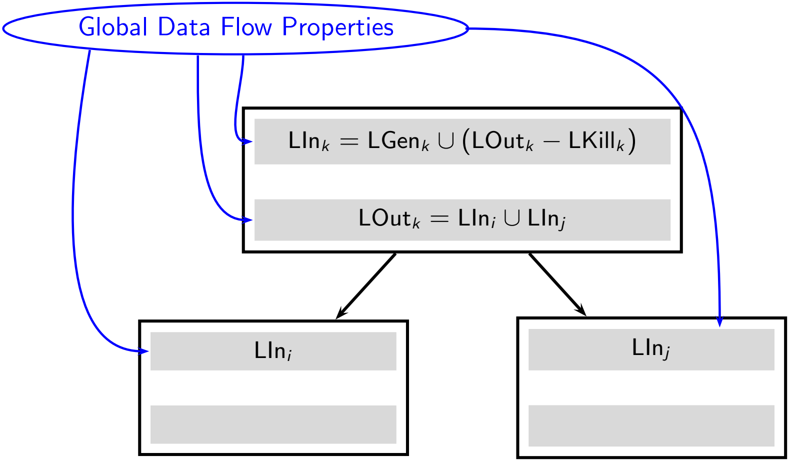
$$LIn_k = LGen_k \cup (LOut_k - LKill_k)$$

$$LOut_k = LIn_i \cup LIn_j$$

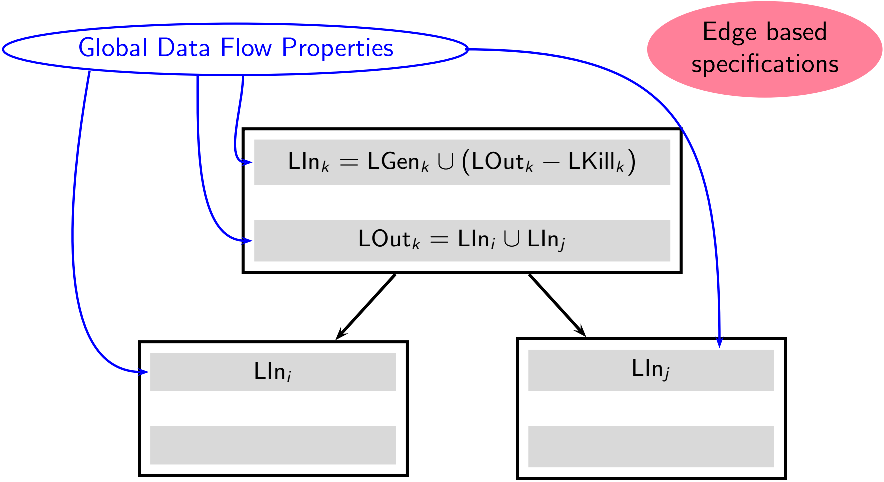
$LIn_i$

$LIn_j$

# Defining Data Flow Analysis for Live Variables Analysis



# Defining Data Flow Analysis for Live Variables Analysis



# Data Flow Equations For Live Variables Analysis

$$\mathsf{LIn}_b = \mathsf{LGen}_b \cup (\mathsf{LOut}_b - \mathsf{LKill}_b)$$

$$\mathsf{LOut}_b = \begin{cases} \emptyset & b \text{ is the exit node} \\ \displaystyle\bigcup_{s \in succ(b)} \mathsf{LIn}_s & \text{otherwise} \end{cases}$$

# Data Flow Equations For Live Variables Analysis

$$\mathsf{LIn}_b = \mathsf{LGen}_b \cup (\mathsf{LOut}_b - \mathsf{LKill}_b)$$

$$\mathsf{LOut}_b = \begin{cases} \emptyset & b \text{ is the exit node} \\ \displaystyle\bigcup_{s \in succ(b)} \mathsf{LIn}_s & \text{otherwise} \end{cases}$$

Alternatively,

$$\mathsf{LIn}_b = f_b(\mathsf{LOut}_b), \qquad \text{where}$$

$$f_b(X) = \mathsf{LGen}_b \cup (X - \mathsf{LKill}_b)$$

# Data Flow Equations For Live Variables Analysis

$$\mathsf{LIn}_b = \mathsf{LGen}_b \cup (\mathsf{LOut}_b - \mathsf{LKill}_b)$$

$$\mathsf{LOut}_b = \begin{cases} \emptyset & b \text{ is the exit node} \\ \bigcup\limits_{s \in succ(b)} \mathsf{LIn}_s & \text{otherwise} \end{cases}$$

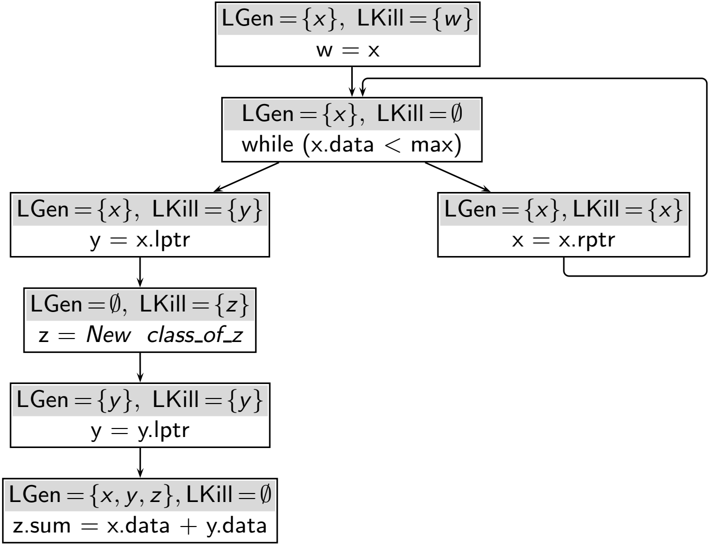
Alternatively,

$$\mathsf{LIn}_b = f_b(\mathsf{LOut}_b), \qquad \text{where}$$

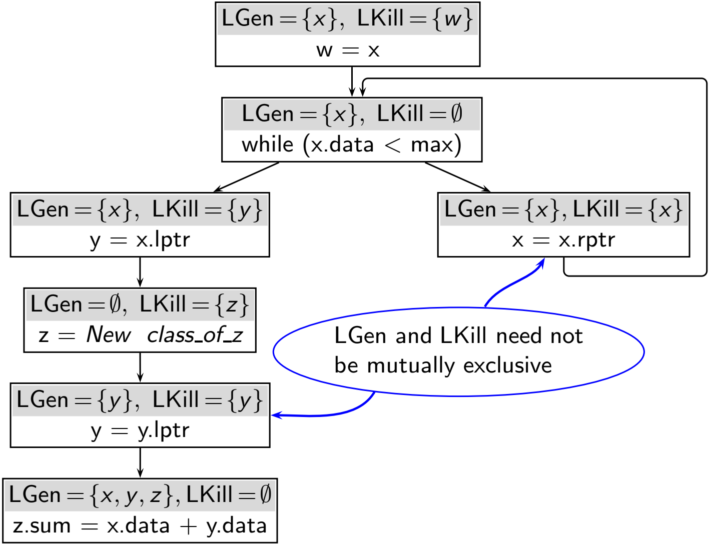
$$f_b(X) = \mathsf{LGen}_b \cup (X - \mathsf{LKill}_b)$$

$\mathsf{LIn}_b$ and $\mathsf{LOut}_b$ are sets of variables.

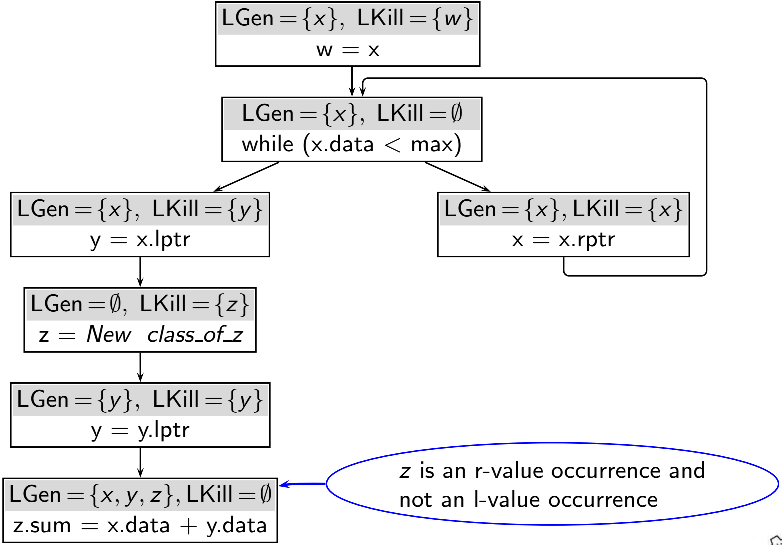
# Performing Live Variables Analysis



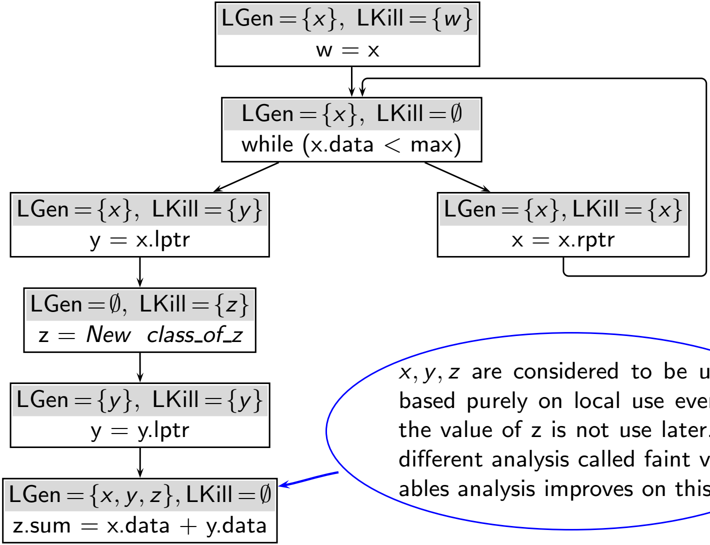
# Performing Live Variables Analysis



# Performing Live Variables Analysis

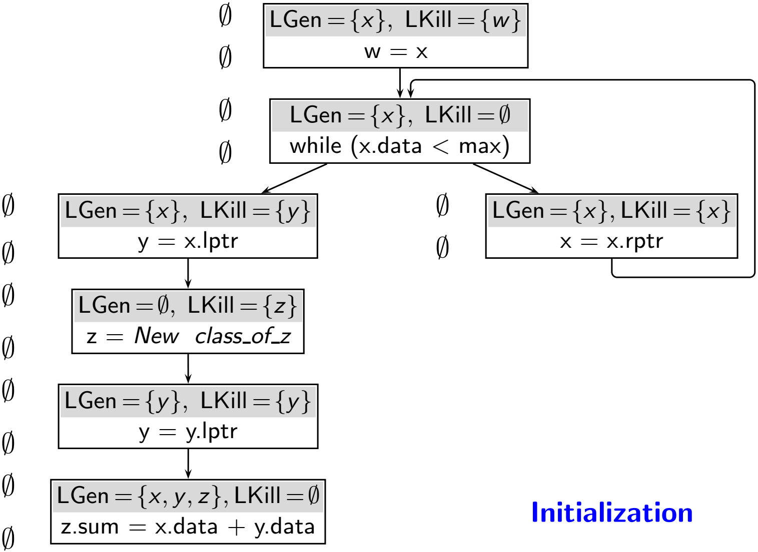


# Performing Live Variables Analysis



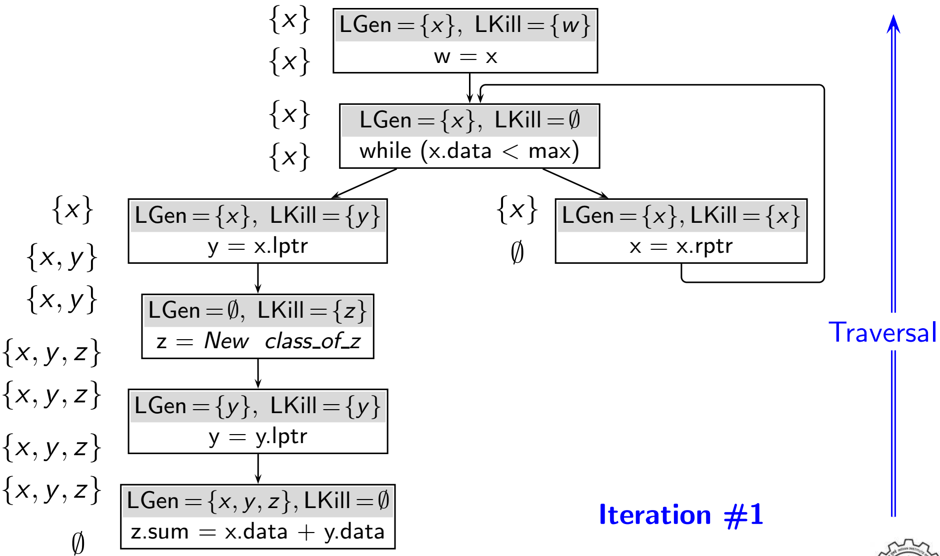

$x, y, z$ are considered to be used based purely on local use even if the value of z is not use later. A different analysis called faint variables analysis improves on this.

# Performing Live Variables Analysis

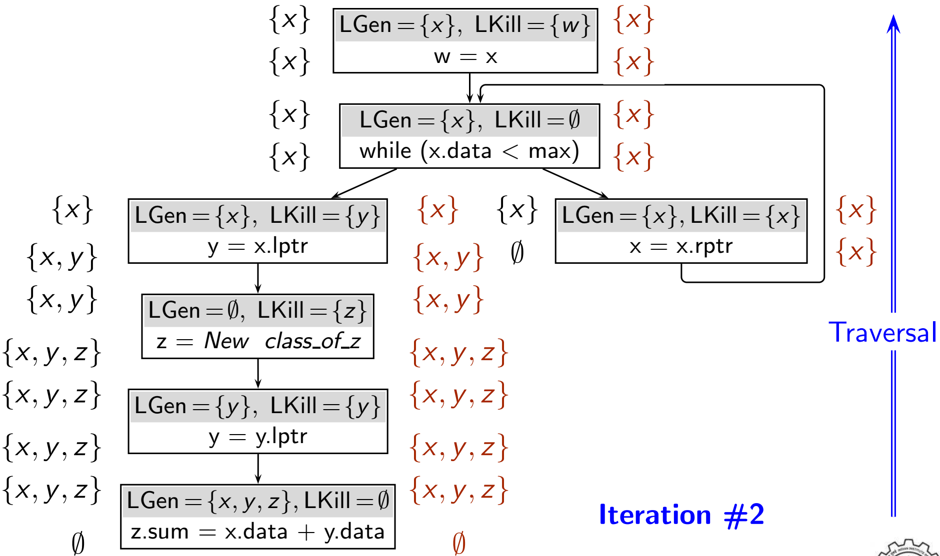



**Initialization**

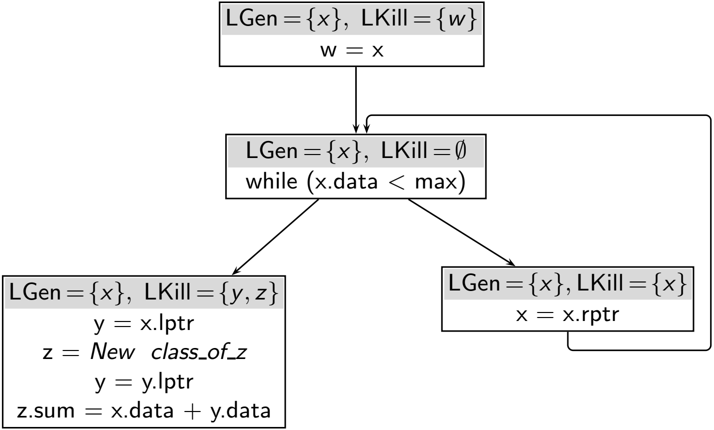
# Performing Live Variables Analysis



# Performing Live Variables Analysis

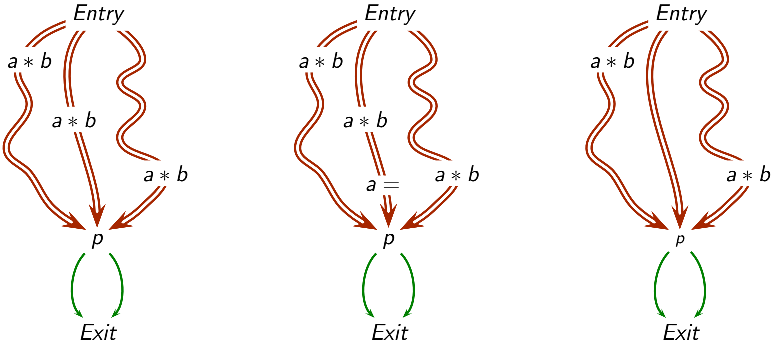


# Performing Live Variables Analysis



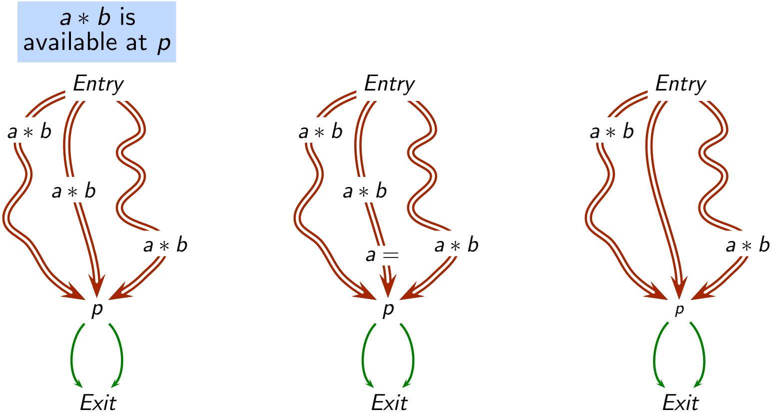
# Available Expressions Analysis

An expression $e$ is available at a program point $p$, if every path from program entry to $p$ contains an evaluation of $e$ which is not followed by a definition of any operand of $e$.



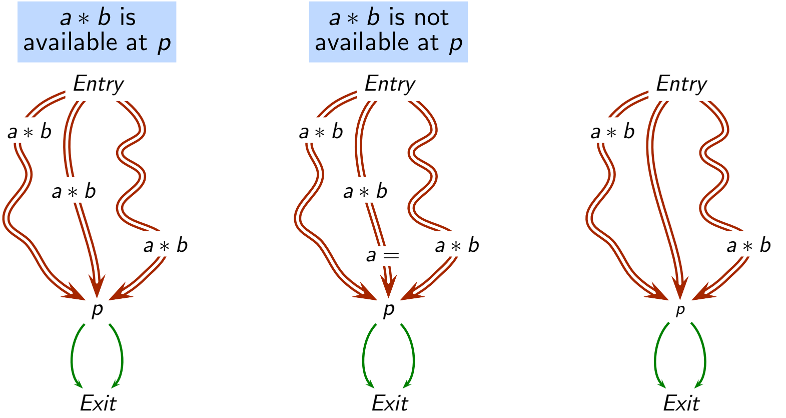
# Available Expressions Analysis

An expression $e$ is available at a program point $p$, if
every path from program entry to $p$ contains an evaluation of $e$
which is not followed by a definition of any operand of $e$.



# Available Expressions Analysis

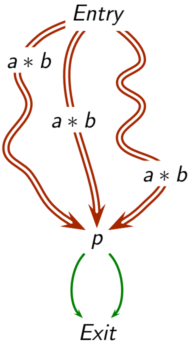
An expression $e$ is available at a program point $p$, if every path from program entry to $p$ contains an evaluation of $e$ which is not followed by a definition of any operand of $e$.



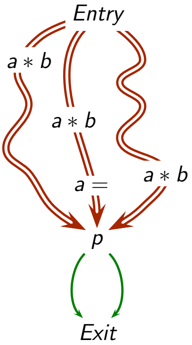
# Available Expressions Analysis

An expression $e$ is available at a program point $p$, if every path from program entry to $p$ contains an evaluation of $e$ which is not followed by a definition of any operand of $e$.

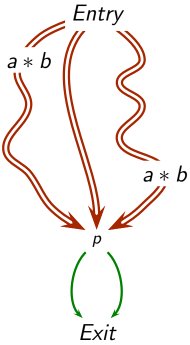
$a * b$ is available at $p$




$a * b$ is not available at $p$




$a * b$ is not available at $p$



# Local Data Flow Properties for Available Expressions Analysis

$$\begin{aligned}
\mathsf{AGen}_b &= \{\, e \mid \text{expression } e \text{ is evaluated in basic block } b \text{ and} \\
&\qquad \text{this evaluation is not followed by a definition of} \\
&\qquad \text{any operand of } e \} \\
\mathsf{AKill}_b &= \{\, e \mid \text{basic block } b \text{ contains a definition of an operand of } e \}
\end{aligned}$$

# Data Flow Equations For Available Expressions Analysis

$$\mathsf{AIn}_b = \begin{cases} \emptyset & b \text{ is the entry node} \\ \displaystyle\bigcap_{p \in pred(b)} \mathsf{AOut}_p & \text{otherwise} \end{cases}$$

$$\mathsf{AOut}_b = \mathsf{AGen}_b \cup (\mathsf{AIn}_b - \mathsf{AKill}_b)$$

# Data Flow Equations For Available Expressions Analysis

$$
\mathsf{AIn}_b = \begin{cases} \emptyset & b \text{ is the entry node} \\ \bigcap\limits_{p \in pred(b)} \mathsf{AOut}_p & \text{otherwise} \end{cases}
$$

$$
\mathsf{AOut}_b = \mathsf{AGen}_b \cup (\mathsf{AIn}_b - \mathsf{AKill}_b)
$$

Alternatively,

$$
\mathsf{AOut}_b = f_b(\mathsf{AIn}_b), \qquad \text{where}
$$

$$
f_b(X) = \mathsf{AGen}_b \cup (X - \mathsf{AKill}_b)
$$

# Data Flow Equations For Available Expressions Analysis

$$\mathsf{AIn}_b = \begin{cases} \emptyset & b \text{ is the entry node} \\ \displaystyle\bigcap_{p \in pred(b)} \mathsf{AOut}_p & \text{otherwise} \end{cases}$$

$$\mathsf{AOut}_b = \mathsf{AGen}_b \cup (\mathsf{AIn}_b - \mathsf{AKill}_b)$$

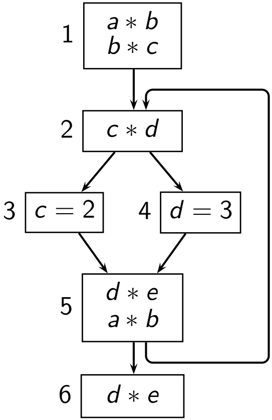
Alternatively,

$$\mathsf{AOut}_b = f_b(\mathsf{AIn}_b), \quad \text{where}$$

$$f_b(X) = \mathsf{AGen}_b \cup (X - \mathsf{AKill}_b)$$

$\mathsf{AIn}_b$ and $\mathsf{AOut}_b$ are sets of expressions.

# An Example of Available Expressions Analysis

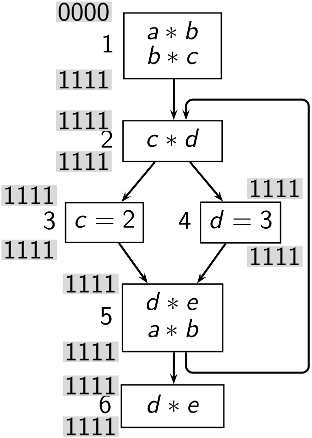



Let $e_1 \equiv a * b$, $e_2 \equiv b * c$, $e_3 \equiv c * d$, $e_4 \equiv d * e$

| Node | Computed | | Killed | | Available | | Redund. | |
|---|---|---|---|---|---|---|---|---|
| 1 | $\{e_1, e_2\}$ | 1100 | $\emptyset$ | 0000 | $\emptyset$ | 0000 | $\emptyset$ | 0000 |
| 2 | $\{e_3\}$ | 0010 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\emptyset$ | 0000 |
| 3 | $\emptyset$ | 0000 | $\{e_2, e_3\}$ | 0110 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 4 | $\emptyset$ | 0000 | $\{e_3, e_4\}$ | 0011 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 5 | $\{e_1, e_4\}$ | 1001 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\{e_1\}$ | 1000 |
| 6 | $\{e_4\}$ | 0001 | $\emptyset$ | 0000 | $\{e_1, e_4\}$ | 1001 | $\{e_4\}$ | 0001 |

# An Example of Available Expressions Analysis

## Initialisation

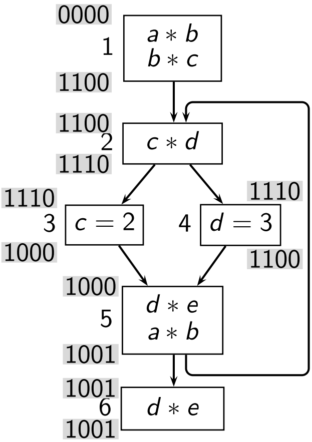



Let $e_1 \equiv a * b$, $e_2 \equiv b * c$, $e_3 \equiv c * d$, $e_4 \equiv d * e$

| Node | Computed | | Killed | | Available | | Redund. | |
|---|---|---|---|---|---|---|---|---|
| 1 | $\{e_1, e_2\}$ | 1100 | $\emptyset$ | 0000 | $\emptyset$ | 0000 | $\emptyset$ | 0000 |
| 2 | $\{e_3\}$ | 0010 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\emptyset$ | 0000 |
| 3 | $\emptyset$ | 0000 | $\{e_2, e_3\}$ | 0110 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 4 | $\emptyset$ | 0000 | $\{e_3, e_4\}$ | 0011 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 5 | $\{e_1, e_4\}$ | 1001 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\{e_1\}$ | 1000 |
| 6 | $\{e_4\}$ | 0001 | $\emptyset$ | 0000 | $\{e_1, e_4\}$ | 1001 | $\{e_4\}$ | 0001 |

# An Example of Available Expressions Analysis

## Iteration #1

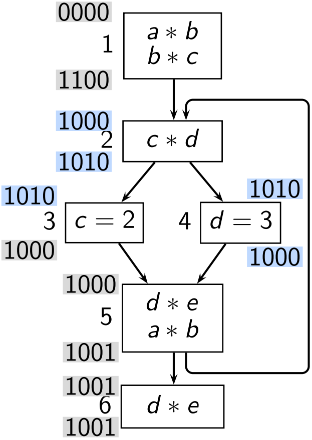



Let $e_1 \equiv a * b$, $e_2 \equiv b * c$, $e_3 \equiv c * d$, $e_4 \equiv d * e$

| Node | Computed | | Killed | | Available | | Redund. | |
|---|---|---|---|---|---|---|---|---|
| 1 | $\{e_1, e_2\}$ | 1100 | $\emptyset$ | 0000 | $\emptyset$ | 0000 | $\emptyset$ | 0000 |
| 2 | $\{e_3\}$ | 0010 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\emptyset$ | 0000 |
| 3 | $\emptyset$ | 0000 | $\{e_2, e_3\}$ | 0110 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 4 | $\emptyset$ | 0000 | $\{e_3, e_4\}$ | 0011 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 5 | $\{e_1, e_4\}$ | 1001 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\{e_1\}$ | 1000 |
| 6 | $\{e_4\}$ | 0001 | $\emptyset$ | 0000 | $\{e_1, e_4\}$ | 1001 | $\{e_4\}$ | 0001 |

# An Example of Available Expressions Analysis

## Iteration #2

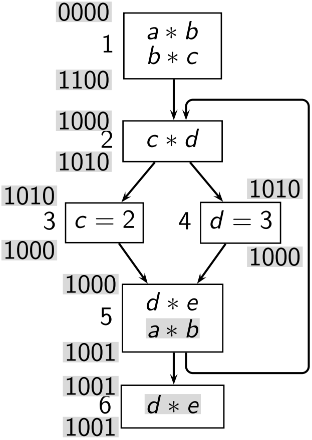



Let $e_1 \equiv a * b$, $e_2 \equiv b * c$, $e_3 \equiv c * d$, $e_4 \equiv d * e$

| Node | Computed | | Killed | | Available | | Redund. | |
|---|---|---|---|---|---|---|---|---|
| 1 | $\{e_1, e_2\}$ | 1100 | $\emptyset$ | 0000 | $\emptyset$ | 0000 | $\emptyset$ | 0000 |
| 2 | $\{e_3\}$ | 0010 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\emptyset$ | 0000 |
| 3 | $\emptyset$ | 0000 | $\{e_2, e_3\}$ | 0110 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 4 | $\emptyset$ | 0000 | $\{e_3, e_4\}$ | 0011 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 5 | $\{e_1, e_4\}$ | 1001 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\{e_1\}$ | 1000 |
| 6 | $\{e_4\}$ | 0001 | $\emptyset$ | 0000 | $\{e_1, e_4\}$ | 1001 | $\{e_4\}$ | 0001 |

# An Example of Available Expressions Analysis

## Final Result




Let $e_1 \equiv a * b$, $e_2 \equiv b * c$, $e_3 \equiv c * d$, $e_4 \equiv d * e$

| Node | Computed | | Killed | | Available | | Redund. | |
|---|---|---|---|---|---|---|---|---|
| 1 | $\{e_1, e_2\}$ | 1100 | $\emptyset$ | 0000 | $\emptyset$ | 0000 | $\emptyset$ | 0000 |
| 2 | $\{e_3\}$ | 0010 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\emptyset$ | 0000 |
| 3 | $\emptyset$ | 0000 | $\{e_2, e_3\}$ | 0110 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 4 | $\emptyset$ | 0000 | $\{e_3, e_4\}$ | 0011 | $\{e_1, e_3\}$ | 1010 | $\emptyset$ | 0000 |
| 5 | $\{e_1, e_4\}$ | 1001 | $\emptyset$ | 0000 | $\{e_1\}$ | 1000 | $\{e_1\}$ | 1000 |
| 6 | $\{e_4\}$ | 0001 | $\emptyset$ | 0000 | $\{e_1, e_4\}$ | 1001 | $\{e_4\}$ | 0001 |

# Using Data Flow Information

*Available Expressions Analysis*.

- Used for common subexpression elimination.

# Using Data Flow Information

*Available Expressions Analysis*.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block $b$ **and**

# Using Data Flow Information

*Available Expressions Analysis*.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block *b* **and**
  - a computation of the expression exists in *b* **such that**

# Using Data Flow Information

*Available Expressions Analysis*.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block *b* **and**
  - a computation of the expression exists in *b* **such that**
  - it is not preceded by a definition of any of its operands

# Using Data Flow Information

*Available Expressions Analysis*.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block $b$ **and**
  - a computation of the expression exists in $b$ **such that**
  - it is not preceded by a definition of any of its operands

  Then the expression is redundant.

# Using Data Flow Information

*Available Expressions Analysis*.

- Used for common subsexpression elimination.
  - If an expression is available at the entry of a block *b* **and**
  - a computation of the expression exists in *b* **such that**
  - it is not preceded by a definition of any of its operands

  Then the expression is redundant.
- Expression must be *upwards exposed* or *locally anticipable*.

# Using Data Flow Information

*Available Expressions Analysis*.

- Used for common subexpression elimination.
  - If an expression is available at the entry of a block $b$ **and**
  - a computation of the expression exists in $b$ **such that**
  - it is not preceded by a definition of any of its operands

  Then the expression is redundant.
- Expression must be *upwards exposed* or *locally anticipable*.
- Expressions in $\text{Gen}_b$ are *downwards exposed*.

# Using Data Flow Information

*Live Variables Analysis.*

- Used for register allocation.
  If variable $x$ is live in a basic block $b$, it is a potential candidate for register allocation.

# Using Data Flow Information

*Live Variables Analysis.*

- Used for register allocation.
  If variable $x$ is live in a basic block $b$, it is a potential candidate for register allocation.
- Used for dead code elimination.
  If variable $x$ is not live after an assignment $x = \ldots$, then the assginment is redundant and can be deleted as dead code.

# Reaching Definitions Analysis

- A definition $d_x : x = y$ reaches a program point $u$ if it appears (without a refefinition of $x$) on **some** path **from program entry to** $u$
- Application : Copy Propagation
  A use of a variable $x$ at a program point $u$ can be replaced by $y$ if $d_x : x = y$ is the only definition which reaches $p$ and $y$ is not modified between the point of $d_x$ and $p$.

# Defining Data Flow Analysis for Reaching Definitions Analysis

Let $d_v$ be a definition of variable $v$

$$\begin{aligned}\text{Gen}_b &= \{\, d_v \mid \text{variable } v \text{ is defined in basic block } b \text{ and} \\ &\qquad \text{this definition is not followed (within } b\text{) by} \\ &\qquad \text{a definition of } v\} \\[1em] \text{Kill}_b &= \{\, d_v \mid \text{basic block } b \text{ contains a definition of } v\}\end{aligned}$$

# Data Flow Equations for Reaching Definitions Analysis

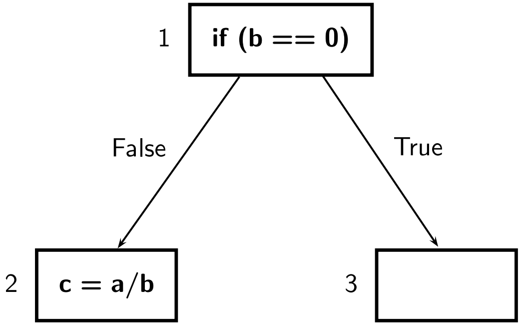
$$
\begin{aligned}
\mathsf{IN}_b &= \begin{cases} \{\} & b \text{ is the entry node} \\ \displaystyle\bigcup_{p \in pred(b)} \mathsf{OUT}_p & \text{otherwise} \end{cases} \\
\mathsf{OUT}_b &= \mathsf{Gen}_b \cup (\mathsf{IN}_b - \mathsf{Kill}_b)
\end{aligned}
$$

$\boxed{\mathsf{IN}_b \text{ and } \mathsf{OUT}_b \text{ are sets of definitions}}$

# Very Busy Expressions Analysis

- An expression $e$ is very busy at a program point $u$, if **every** path **from** $u$ **to the program exit** contains an evaluation of $e$ which is not preceded by a redefinition of any operand of $e$.
- Application : Safety of Code Motion

# Safety of Code Motion




- Expression $a/b$ is not very busy at the exit of 1.
- Moving $a/b$ to the exit of 1 is unsafe.

# Defining Data Flow Analysis for Very Busy Expressions Analysis

$$\text{Gen}_b = \{ e \mid \text{expression } e \text{ is evaluated in basic block } b \text{ and this evaluation is not preceded (within } b\text{) by a definition of any operand of } e\}$$

$$\text{Kill}_b = \{ e \mid \text{basic block } b \text{ contains a definition of an operand of } e\}$$

# Data Flow Equations for Very Busy Expressions Analysis

$$
\begin{aligned}
\mathrm{IN}_b &= \mathrm{Gen}_b \cup (\mathrm{OUT}_b - \mathrm{Kill}_b) \\
\mathrm{OUT}_b &= \begin{cases} \{\} & b \text{ is the exit node} \\ \displaystyle\bigcap_{a \in succ(b)} \mathrm{IN}_s & \text{otherwise} \end{cases}
\end{aligned}
$$

$\boxed{\mathrm{IN}_b \text{ and } \mathrm{OUT}_b \text{ are sets of expressions}}$

# Common Form of Data Flow Equations

$$
\begin{aligned}
X_i &= f(Y_i) \\
Y_i &= \bigsqcap X_j
\end{aligned}
$$

# Common Form of Data Flow Equations

Data Flow Information

So far we have seen sets (or bit vectors).
Could be entities other than sets.

$$
\begin{aligned}
X_i &= f(Y_i) \\
Y_i &= \prod X_j
\end{aligned}
$$

# Common Form of Data Flow Equations

Data Flow Information

So far we have seen sets (or bit vectors).
Could be entities other than sets.

Flow Function

$$
\begin{aligned}
X_i &= f(Y_i) \\
Y_i &= \bigsqcap X_j
\end{aligned}
$$

# Common Form of Data Flow Equations

Data Flow Information

So far we have seen sets (or bit vectors).
Could be entities other than sets.

Flow Function

$$X_i = f(Y_i)$$
$$Y_i = \bigsqcap X_j$$

Confluence

So far we have seen $\cup$ and $\cap$.
Could be other operations.

# A Taxonomy of Bit Vector Data Flow Frameworks

| | Confluence | |
|---|---|---|
| | Union | Intersection |
| Forward | Reaching Definitions | Available Expressions |
| Backward | Live Variables | Anticipable Exressions |
| Bidirectional (limited) | | Partial Redundancy Elimination (Original M-R Formulation) |

# A Taxonomy of Bit Vector Data Flow Frameworks

Any Path

| | Confluence | |
|---|---|---|
| | Union | Intersection |
| Forward | Reaching Definitions | Available Expressions |
| Backward | Live Variables | Anticipable Exressions |
| Bidirectional (limited) | | Partial Redundancy Elimination (Original M-R Formulation) |

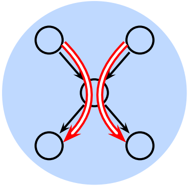
# A Taxonomy of Bit Vector Data Flow Frameworks

Any Path

All Paths

| | Confluence | |
|---|---|---|
| | Union | Intersection |
| Forward | Reaching Definitions | Available Expressions |
| Backward | Live Variables | Anticipable Exressions |
| Bidirectional (limited) | | Partial Redundancy Elimination (Original M-R Formulation) |

# A Taxonomy of Bit Vector Data Flow Frameworks

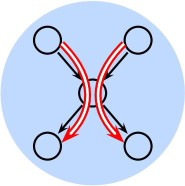


Any Path

All Paths

| | Confluence | |
|---|---|---|
| | Union | Intersection |
| Forward | Reaching Definitions | Available Expressions |
| Backward | Live Variables | Anticipable Exressions |
| Bidirectional (limited) | | Partial Redundancy Elimination (Original M-R Formulation) |

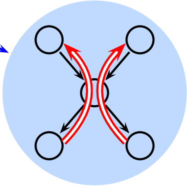
# A Taxonomy of Bit Vector Data Flow Frameworks



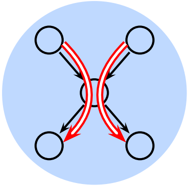
Any Path

All Paths

| | Confluence | |
|---|---|---|
| | Union | Intersection |
| Forward | Reaching Definitions | Available Expressions |
| Backward | Live Variables | Anticipable Exressions |
| Bidirectional (limited) | | Partial Redundancy Elimination (Original M-R Formulation) |



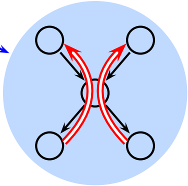
# A Taxonomy of Bit Vector Data Flow Frameworks

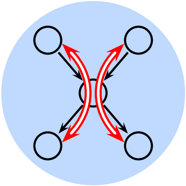


Any Path

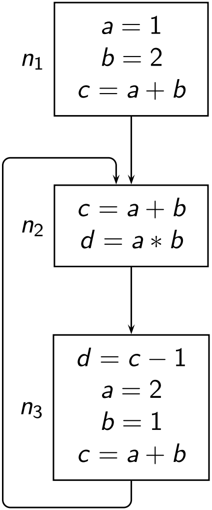
All Paths

| | Confluence | |
|---|---|---|
| | Union | Intersection |
| Forward | Reaching Definitions | Available Expressions |
| Backward | Live Variables | Anticipable Exressions |
| Bidirectional (limited) | | Partial Redundancy Elimination (Original M-R Formulation) |

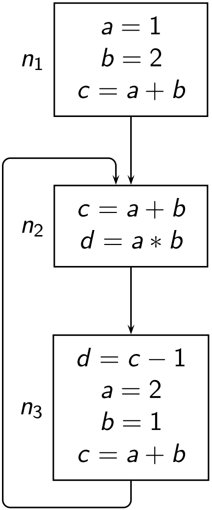


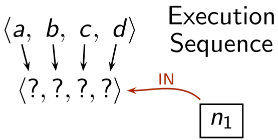


# An Introduction to Constant Propagation

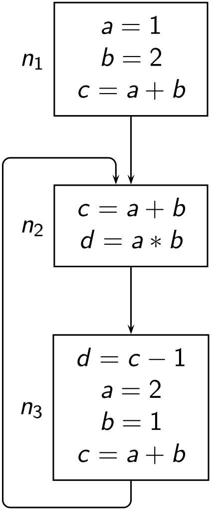


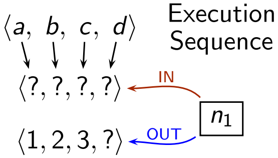
# An Introduction to Constant Propagation



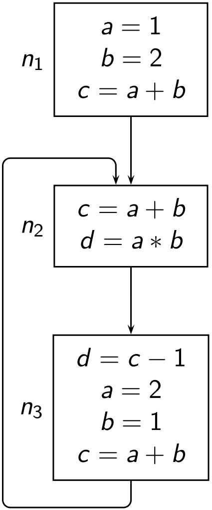



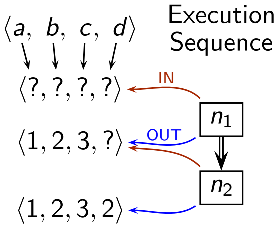
# An Introduction to Constant Propagation



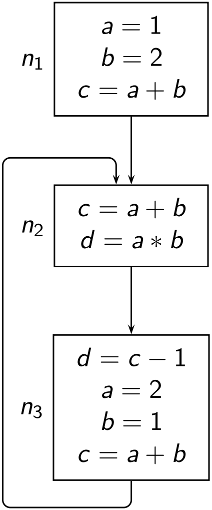



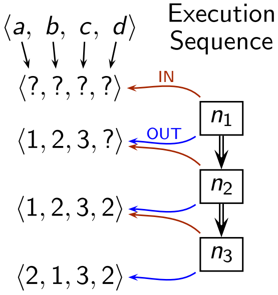
# An Introduction to Constant Propagation



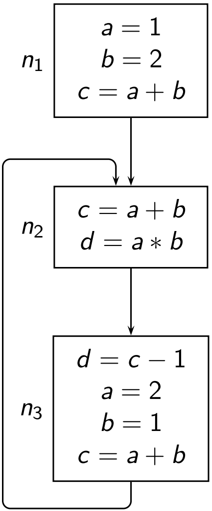



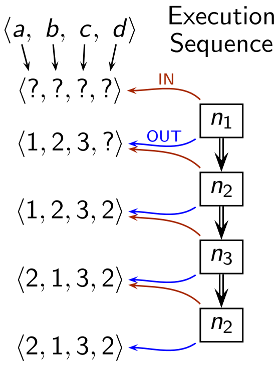
# An Introduction to Constant Propagation



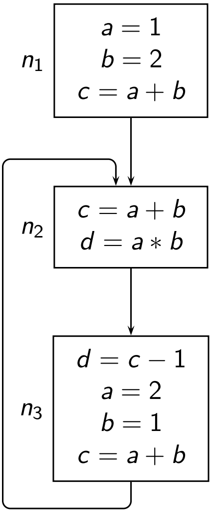



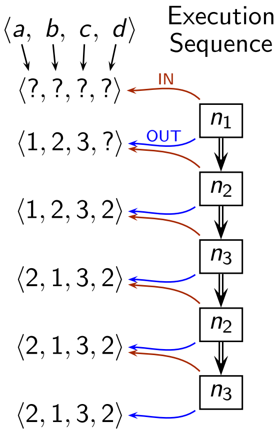
# An Introduction to Constant Propagation



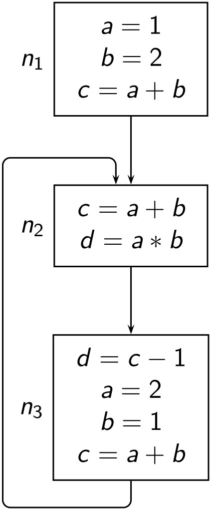



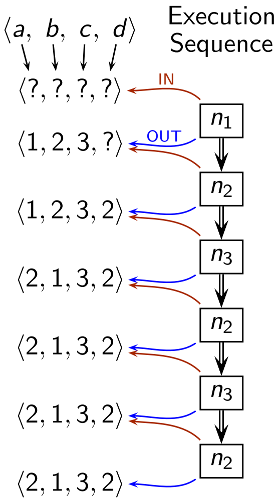
# An Introduction to Constant Propagation



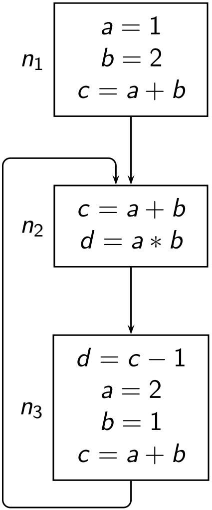



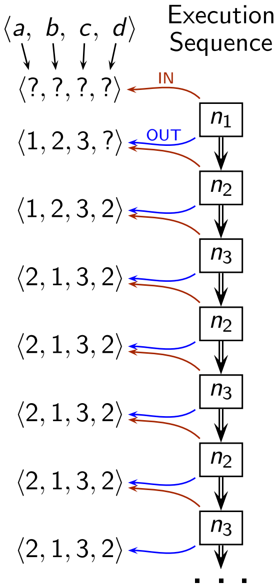
# An Introduction to Constant Propagation



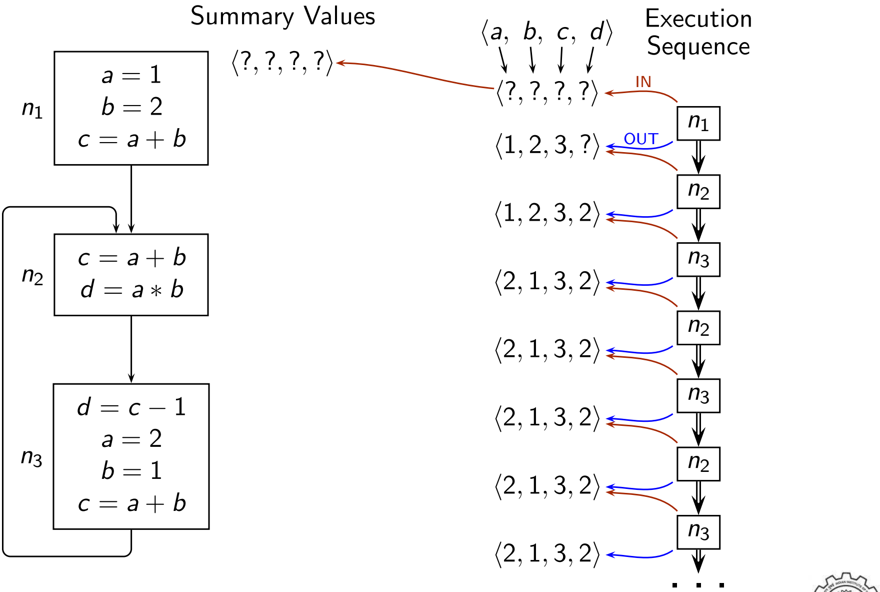



# An Introduction to Constant Propagation

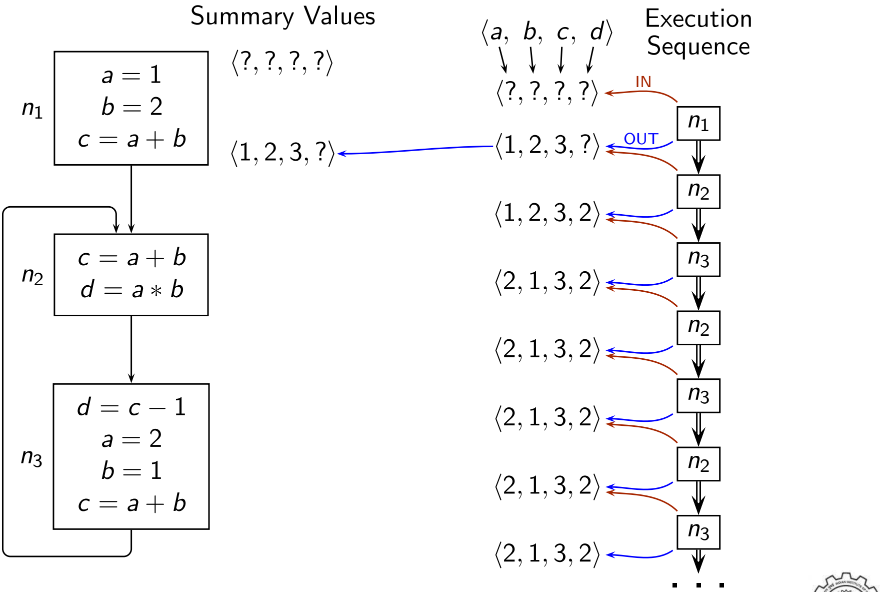


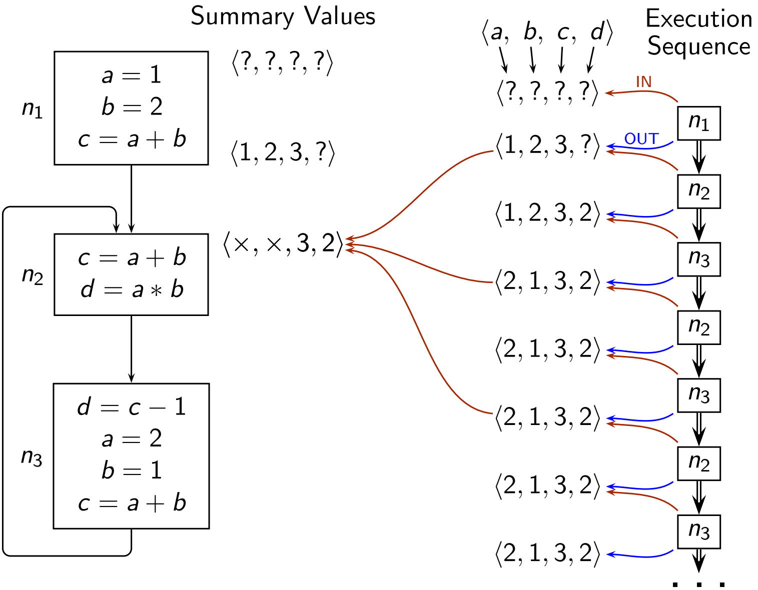
# An Introduction to Constant Propagation



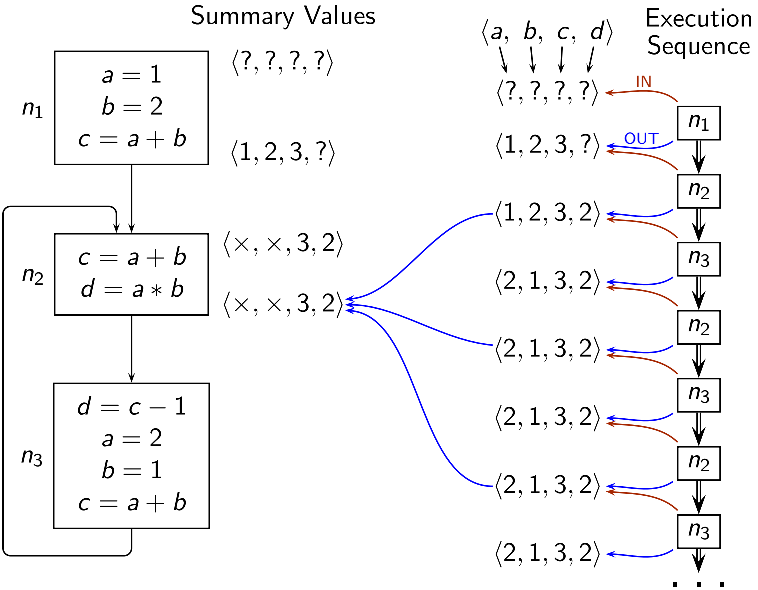
# An Introduction to Constant Propagation



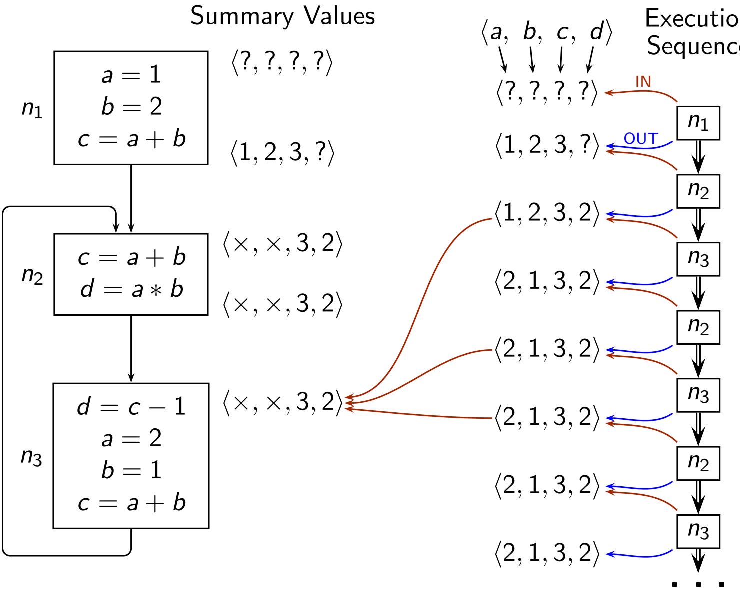
# An Introduction to Constant Propagation



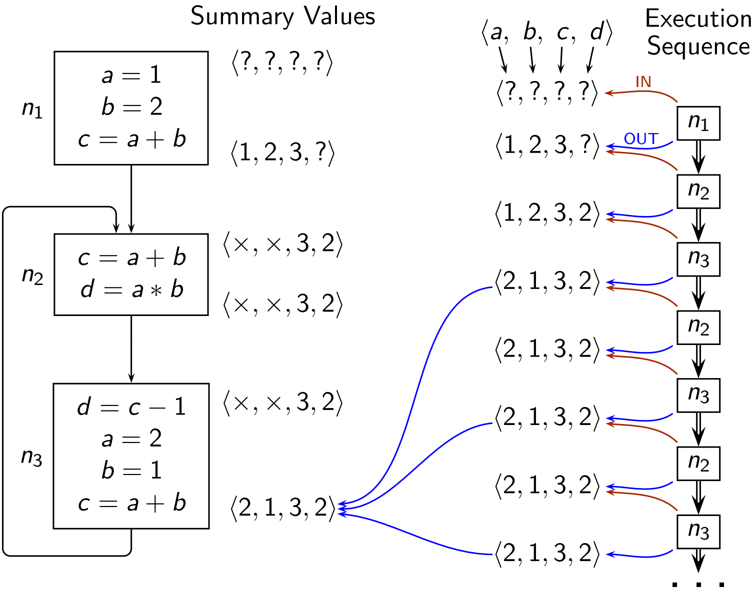
# An Introduction to Constant Propagation



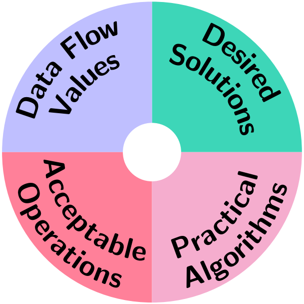
# An Introduction to Constant Propagation



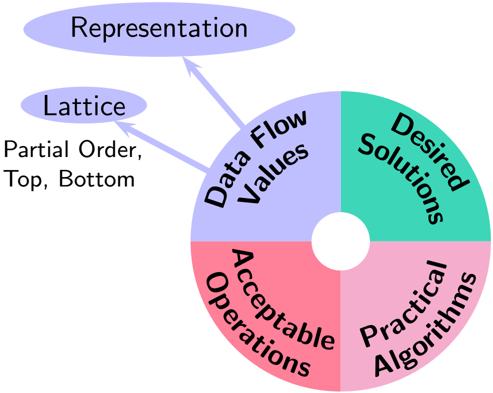
# An Introduction to Constant Propagation



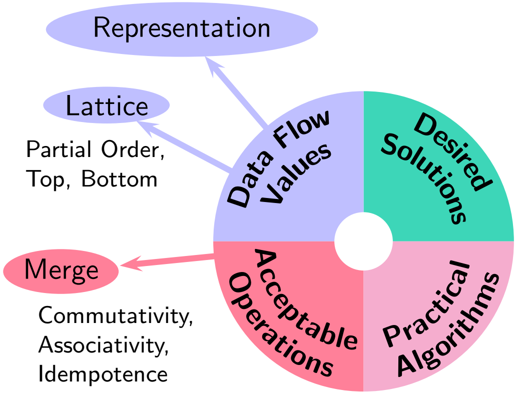
# Issues


Data Flow Values
Desired Solutions
Acceptable Operations
Practical Algorithms

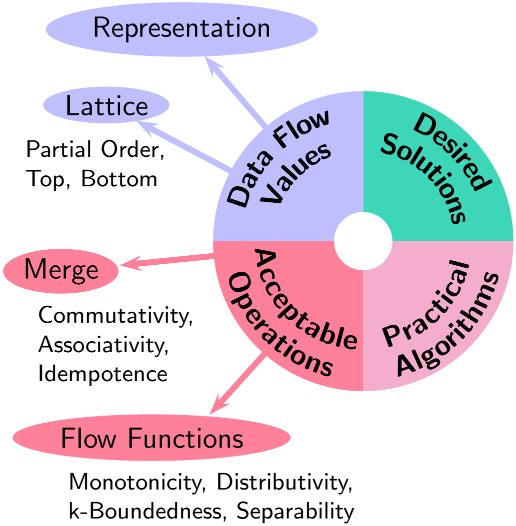
# Issues


Representation
Lattice
Partial Order,
Top, Bottom
Data Flow Values
Desired Solutions
Acceptable Operations
Practical Algorithms

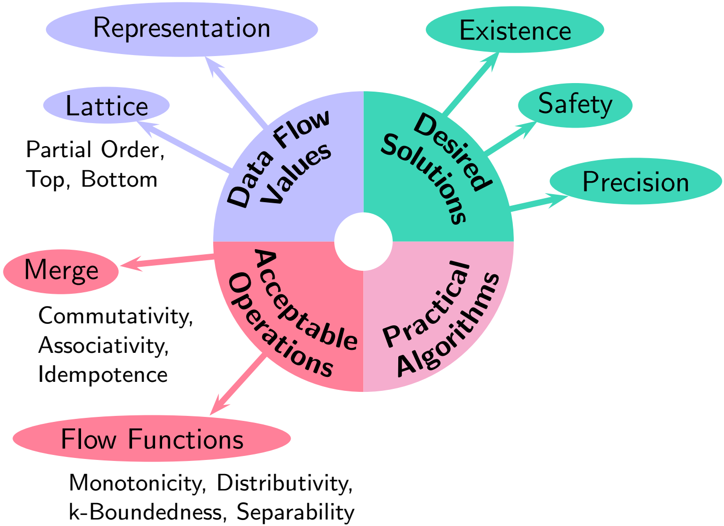
# Issues


Representation
Lattice
Partial Order,
Top, Bottom
Merge
Commutativity,
Associativity,
Idempotence
Data Flow Values
Desired Solutions
Acceptable Operations
Practical Algorithms

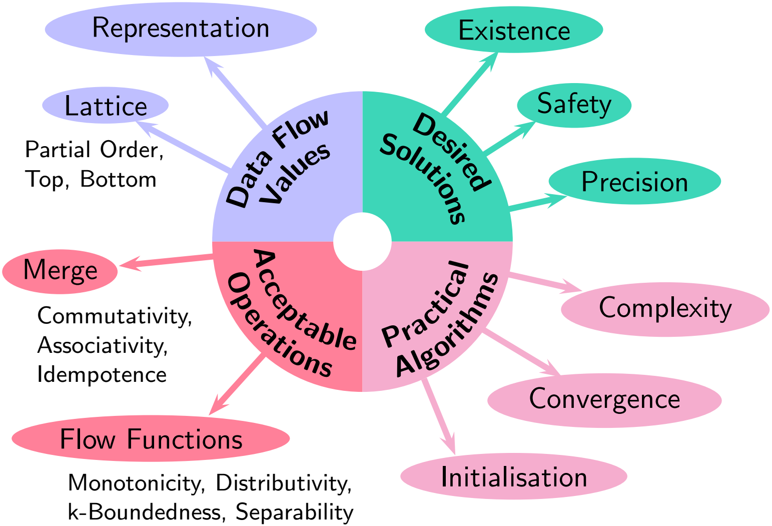
# Issues


Representation
Lattice
Partial Order,
Top, Bottom
Data Flow Values
Desired Solutions
Merge
Commutativity,
Associativity,
Idempotence
Acceptable Operations
Practical Algorithms
Flow Functions
Monotonicity, Distributivity,
k-Boundedness, Separability

# Issues


Representation
Lattice
Partial Order, Top, Bottom
Data Flow Values
Existence
Safety
Precision
Desired Solutions
Merge
Commutativity, Associativity, Idempotence
Acceptable Operations
Practical Algorithms
Flow Functions
Monotonicity, Distributivity, k-Boundedness, Separability

# Issues


Representation
Lattice
Partial Order, Top, Bottom
Data Flow Values
Existence
Safety
Precision
Desired Solutions
Merge
Commutativity, Associativity, Idempotence
Acceptable Operations
Flow Functions
Monotonicity, Distributivity, k-Boundedness, Separability
Practical Algorithms
Complexity
Convergence
Initialisation

*Part 4*

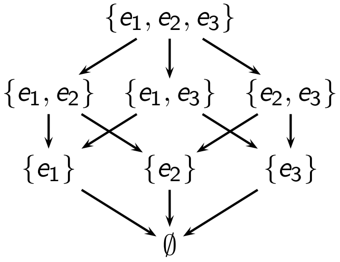
# *Data Flow Values*

# The Set of Data Flow Values

- Properties of the data flow values
- The notion of approximations
- Combining data flow values

# The Set of Data Flow Values

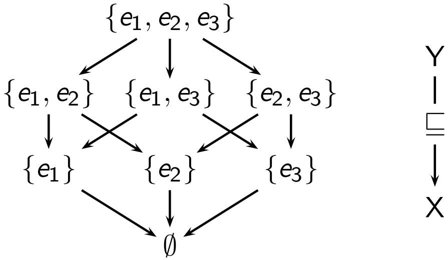
- Can be looked upon as a partially ordered set
  For available expressions analysis,
  - The powerset of the universal set of expressions
  - Partial order is the subset relation




Set View of the Lattice

# The Set of Data Flow Values

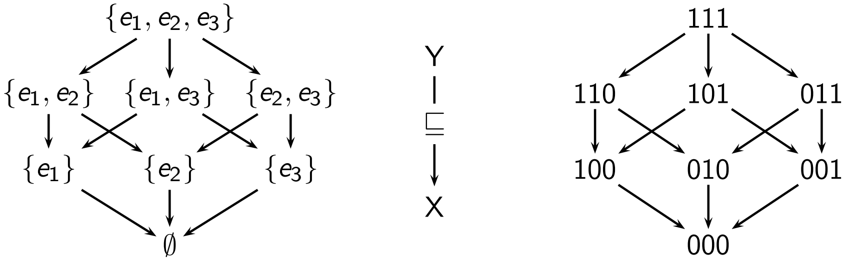
- Can be looked upon as a partially ordered set
  For available expressions analysis,
  - The powerset of the universal set of expressions
  - Partial order is the subset relation




Set View of the Lattice

# The Set of Data Flow Values

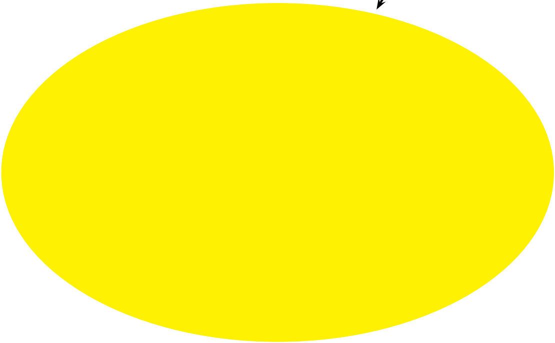
- Can be looked upon as a partially ordered set
  For available expressions analysis,
  - The powerset of the universal set of expressions
  - Partial order is the subset relation




Set View of the Lattice

Bit Vector View

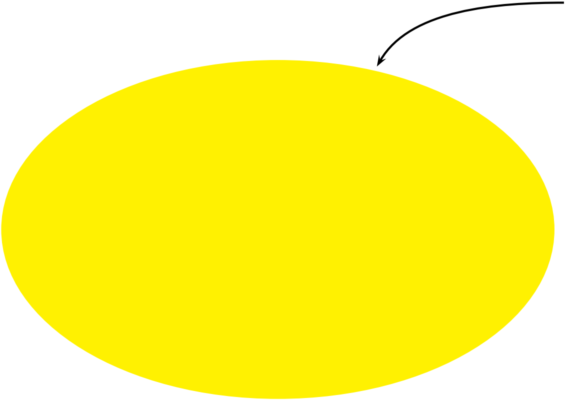
# An Aside on Lattices



Partially ordered sets

Partial order $\sqsubseteq$ is reflexive, transitive, and antisymmetric

# An Aside on Lattices



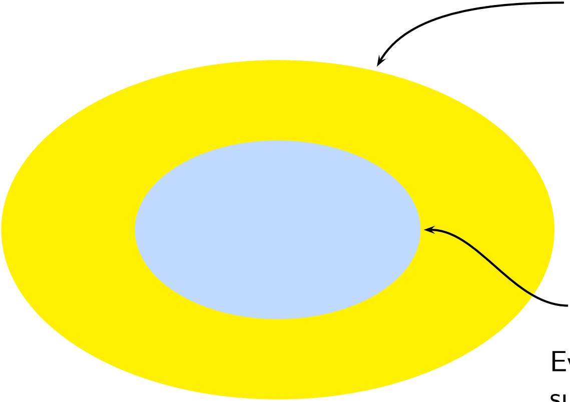
Partially ordered sets

Partial order $\sqsubseteq$ is reflexive, transitive, and antisymmetric

A lower bound of $x, y$ is $u$ s.t. $u \sqsubseteq x$ and $u \sqsubseteq y$

An upper bound of $x, y$ is $u$ s.t. $x \sqsubseteq u$ and $y \sqsubseteq u$

# An Aside on Lattices



Partially ordered sets

Partial order $\sqsubseteq$ is reflexive, transitive, and antisymmetric

Lattices

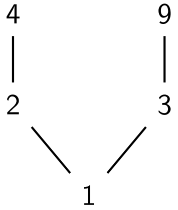
Every non-empty finite subset has a greatest lower bound (glb) and a least upper bound (lub)

# Partially Ordered Sets

Set $\{1, 2, 3, 4, 9\}$ with $\sqsubseteq$ relation as "divides" (i.e. $a \sqsubseteq b$ iff a divides b)

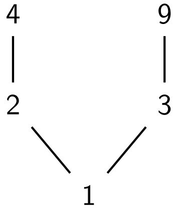
# Partially Ordered Sets

Set $\{1, 2, 3, 4, 9\}$ with $\sqsubseteq$ relation as "divides" (i.e. $a \sqsubseteq b$ iff a divides b)



# Partially Ordered Sets

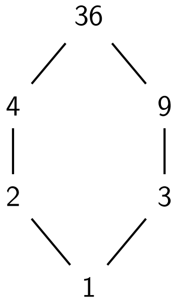
Set $\{1, 2, 3, 4, 9\}$ with $\sqsubseteq$ relation as "divides" (i.e. $a \sqsubseteq b$ iff a divides b)




Subsets $\{4, 9\}$ and $\{2, 3\}$ do not have an upper bound in the set

# Lattice

Set $\{1, 2, 3, 4, 9, 36\}$ with $\sqsubseteq$ relation as "divides" (i.e. $a \sqsubseteq b$ iff a divides b)



# Complete Lattice

- Lattice: Every non-empty finite subset has a glb and a lub.
  Example: Lattice of integers under $\leq$ relation. All finite subsets have a glb and a lub. Infinite subsets do not have a glb or a lub.

# Complete Lattice

- Lattice: Every non-empty finite subset has a glb and a lub.
  Example: Lattice of integers under $\leq$ relation. All finite subsets have a glb and a lub. Infinite subsets do not have a glb or a lub.
- Complete Lattice: Even empty and finite subsets have a glb and a lub.
  - Every finite lattice is complete.

  Example: Lattice of integers under $\leq$ relation with $\infty$ as $\top$ and $-\infty$ as $\perp$. Even infinite subsets have a glb and lub.

# Complete Lattice

- Lattice: Every non-empty finite subset has a glb and a lub.
  Example: Lattice of integers under $\leq$ relation. All finite subsets have a glb and a lub. Infinite subsets do not have a glb or a lub.
- Complete Lattice: Even empty and finite subsets have a glb and a lub.
  - Every finite lattice is complete.

  Example: Lattice of integers under $\leq$ relation with $\infty$ as $\top$ and $-\infty$ as $\bot$. Even infinite subsets have a glb and lub.
- Our discussion is restricted to complete lattices.
  - Each lattice is finite, or
  - glb and lub exists for each subset even if the lattice is infinite.

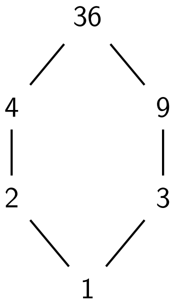
# Ascending and Descending Chains

- Strictly ascending chain. $x \sqsubset y \sqsubset \cdots \sqsubset z$
- Strictly descending chain. $x \sqsupset y \sqsupset \cdots \sqsupset z$
- If all strictly ascending and descending chains in $L$ are finite, then
  - $L$ has finite height, and
  - $L$ is complete.
- A complete lattice need not have finite height (i.e. strict chains may not be finite).
  Example:
  Lattice of integers under $\leq$ relation with $\infty$ as $\top$ and $-\infty$ as $\bot$.

**We require all descending chains to be finite.**

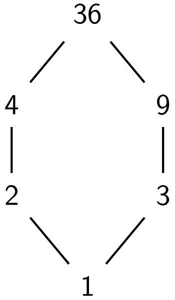
## Operations on Lattices

- Meet ($\sqcap$) and Join ($\sqcup$)



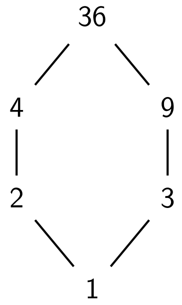
# Operations on Lattices

- Meet ($\sqcap$) and Join ($\sqcup$)
  - $x \sqcap y$ computes the glb of $x$ and $y$.
    $z = x \sqcap y \Rightarrow z \sqsubseteq x \wedge z \sqsubseteq y$



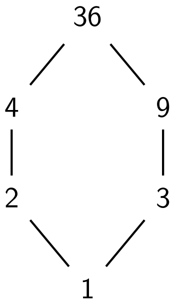
# Operations on Lattices

- Meet ($\sqcap$) and Join ($\sqcup$)
  - $x \sqcap y$ computes the glb of $x$ and $y$.
    $z = x \sqcap y \Rightarrow z \sqsubseteq x \wedge z \sqsubseteq y$
  - $x \sqcup y$ computes the lub of $x$ and $y$.
    $z = x \sqcup y \Rightarrow z \sqsupseteq x \wedge z \sqsupseteq y$



# Operations on Lattices

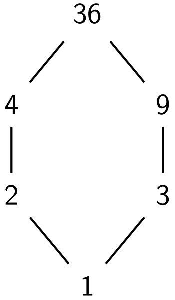
- Meet ($\sqcap$) and Join ($\sqcup$)
  - $x \sqcap y$ computes the glb of $x$ and $y$.
    $z = x \sqcap y \Rightarrow z \sqsubseteq x \wedge z \sqsubseteq y$
  - $x \sqcup y$ computes the lub of $x$ and $y$.
    $z = x \sqcup y \Rightarrow z \sqsupseteq x \wedge z \sqsupseteq y$
  - $\sqcap$ and $\sqcup$ are commutative, associative, and idempotent.



# Operations on Lattices

- Meet ($\sqcap$) and Join ($\sqcup$)
  - $x \sqcap y$ computes the glb of $x$ and $y$.
    $z = x \sqcap y \Rightarrow z \sqsubseteq x \wedge z \sqsubseteq y$
  - $x \sqcup y$ computes the lub of $x$ and $y$.
    $z = x \sqcup y \Rightarrow z \sqsupseteq x \wedge z \sqsupseteq y$
  - $\sqcap$ and $\sqcup$ are commutative, associative, and idempotent.
- Top ($\top$) and Bottom ($\bot$) elements

$$
\begin{array}{rcl}
\forall x \in L,\ x \sqcap \top & = & x \\
\forall x \in L,\ x \sqcup \top & = & \top \\
\forall x \in L,\ x \sqcap \bot & = & \bot \\
\forall x \in L,\ x \sqcup \bot & = & x
\end{array}
$$

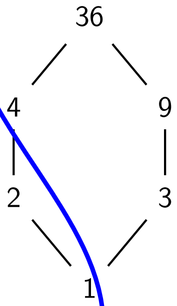


# Operations on Lattices

Greatest common divisor (or highest common factor) **in the lattice**

- Meet ($\sqcap$) and Join ($\sqcup$)
  - $x \sqcap y$ computes the glb of $x$ and $y$.
    $z = x \sqcap y \Rightarrow z \sqsubseteq x \wedge z \sqsubseteq y$
  - $x \sqcup y$ computes the lub of $x$ and $y$.
    $z = x \sqcup y \Rightarrow z \sqsupseteq x \wedge z \sqsupseteq y$
  - $\sqcap$ and $\sqcup$ are commutative, associative, and idempotent.
- Top ($\top$) and Bottom ($\perp$) elements

$$
\begin{aligned}
\forall x \in L,\ x \sqcap \top &= x \\
\forall x \in L,\ x \sqcup \top &= \top \\
\forall x \in L,\ x \sqcap \perp &= \perp \\
\forall x \in L,\ x \sqcup \perp &= x
\end{aligned}
$$



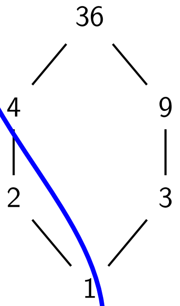

$x \sqcap y = gcd'(x, y)$

# Operations on Lattices

Greatest common divisor (or highest common factor) **in the lattice**

- Meet ($\sqcap$) and Join ($\sqcup$)
  - $x \sqcap y$ computes the glb of $x$ and $y$.
    $z = x \sqcap y \Rightarrow z \sqsubseteq x \wedge z \sqsubseteq y$
  - $x \sqcup y$ computes the lub of $x$ and $y$.
    $z = x \sqcup y \Rightarrow z \sqsupseteq x \wedge z \sqsupseteq y$
  - $\sqcap$ and $\sqcup$ are commutative, associative, and idempotent.
- Top ($\top$) and Bottom ($\bot$) elements

$$
\begin{array}{rcl}
\forall x \in L,\ x \sqcap \top & = & x \\
\forall x \in L,\ x \sqcup \top & = & \top \\
\forall x \in L,\ x \sqcap \bot & = & \bot \\
\forall x \in L,\ x \sqcup \bot & = & x
\end{array}
$$

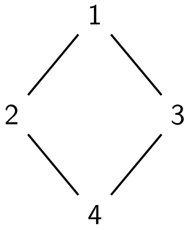



$x \sqcap y = gcd'(x, y)$

$x \sqcup y = lcm'(x, y)$

Lowest common multiple **in the lattice**

# Cartesian Product of Lattice



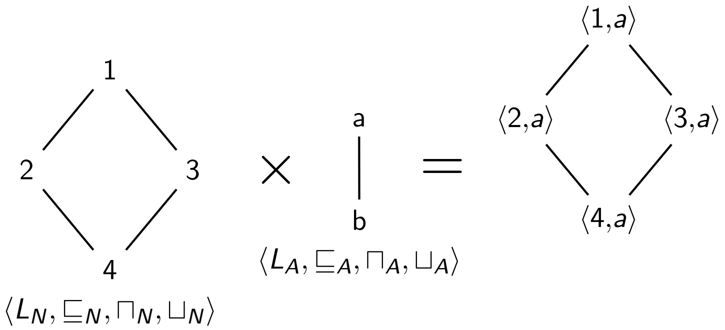

$\langle L_N, \sqsubseteq_N, \sqcap_N, \sqcup_N \rangle$

$\times$

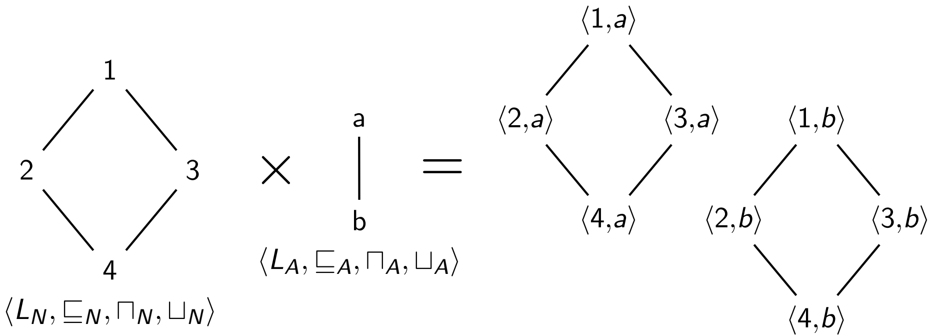

$\langle L_A, \sqsubseteq_A, \sqcap_A, \sqcup_A \rangle$

$=$

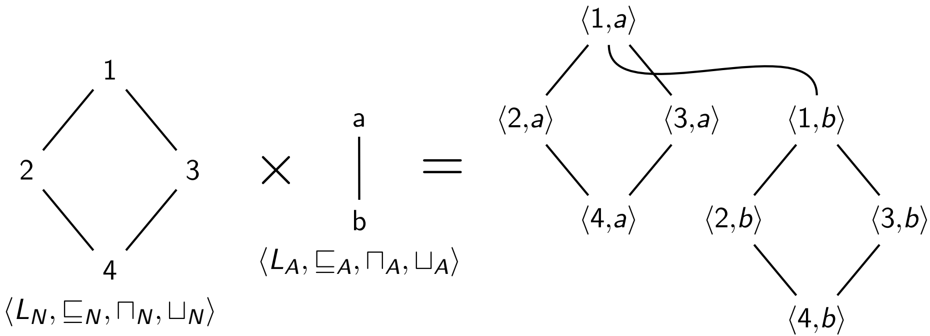
# Cartesian Product of Lattice



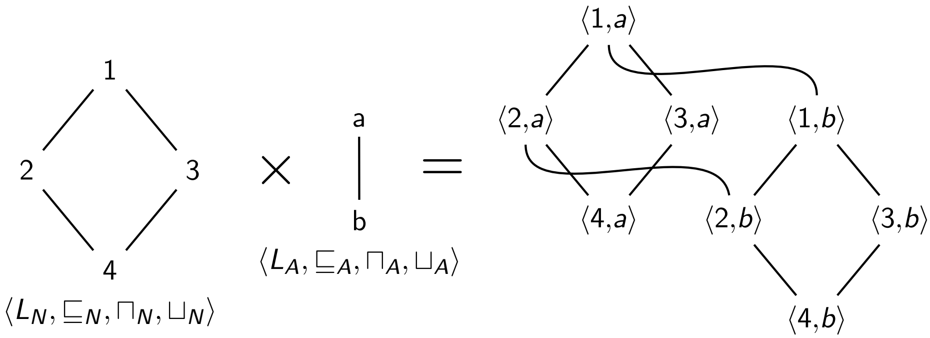
# Cartesian Product of Lattice



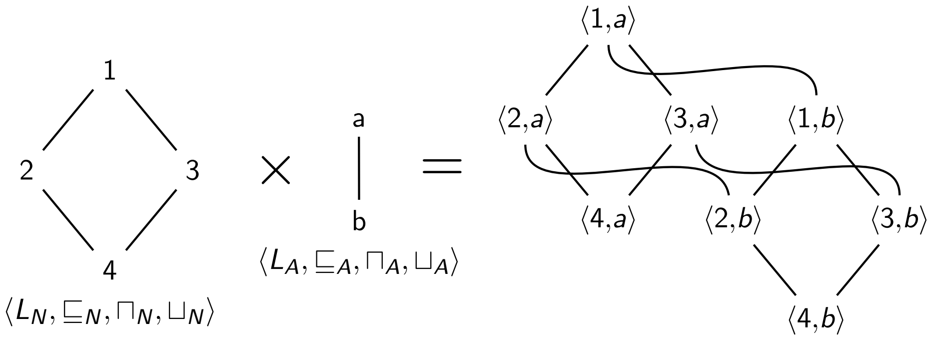
# Cartesian Product of Lattice



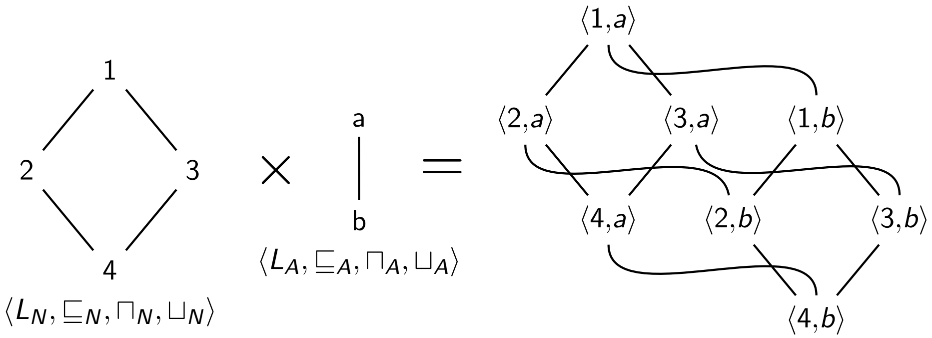
# Cartesian Product of Lattice



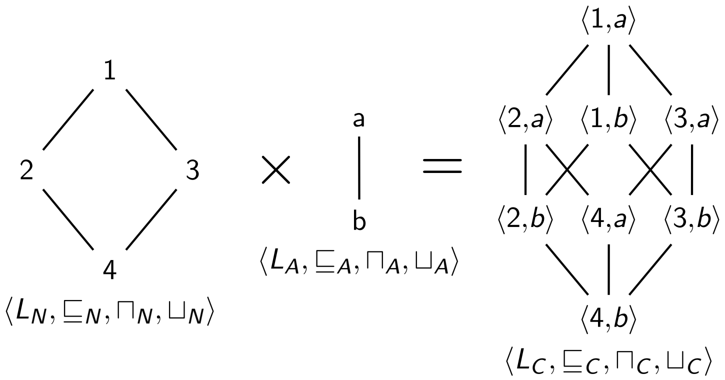
# Cartesian Product of Lattice



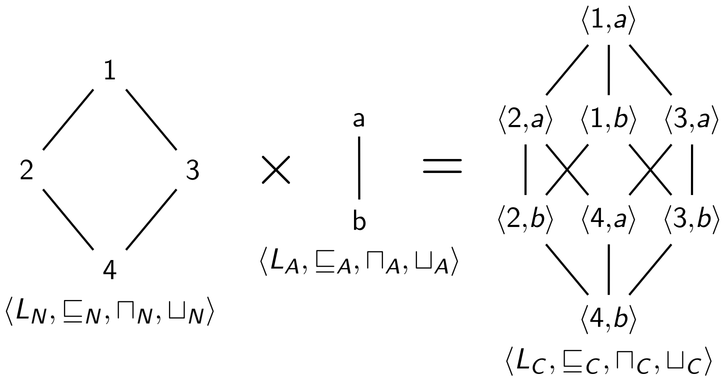
# Cartesian Product of Lattice



# Cartesian Product of Lattice



# Cartesian Product of Lattice




$$\langle x_1, y_1 \rangle \sqsubseteq_C \langle x_2, y_2 \rangle \quad \Leftrightarrow \quad x_1 \sqsubseteq_N x_2 \wedge y_1 \sqsubseteq_A y_2$$

$$\langle x_1, y_1 \rangle \sqcap_C \langle x_2, y_2 \rangle \quad = \quad \langle x_1 \sqcap_N x_2, y_1 \sqcap_A y_2 \rangle$$

$$\langle x_1, y_1 \rangle \sqcup_C \langle x_2, y_2 \rangle \quad = \quad \langle x_1 \sqcup_N x_2, y_1 \sqcup_A y_2 \rangle$$

# The Concept of Approximation

- $x$ approximates $y$ *iff*

  $x$ can be used in place of $y$ without causing any problems.
- Validity of approximation is context specific

  $x$ may be approximated by $y$ in one context and by $z$ in another
  - Earnings : Rs. 1050 can be safely approximated by Rs. 1000.
  - Expenses : Rs. 1050 can be safely approximated by Rs. 1100.

# Two Important Objectives in Data Flow Analysis

- The discovered data flow information should be
  - *Exhaustive*. No optimization opportunity should be missed.
  - *Safe*. Optimizations which do not preserve semantics should not be enabled.

# Two Important Objectives in Data Flow Analysis

- The discovered data flow information should be
  - *Exhaustive*. No optimization opportunity should be missed.
  - *Safe*. Optimizations which do not preserve semantics should not be enabled.
- Conservative approximations of these objectives are allowed

# Two Important Objectives in Data Flow Analysis

- The discovered data flow information should be
  - *Exhaustive*. No optimization opportunity should be missed.
  - *Safe*. Optimizations which do not preserve semantics should not be enabled.
- Conservative approximations of these objectives are allowed
- The intended use of data flow information ($\equiv$ context) determines validity of approximations

# Context Determines the Validity of Approximations

May prohibit correct optimization

May enable wrong optimization

| Analysis | Application | Safe Approximation | Exhaustive Approximation |
|---|---|---|---|

# Context Determines the Validity of Approximations

May prohibit correct optimization

May enable wrong optimization

| Analysis | Application | Safe Approximation | Exhaustive Approximation |
|---|---|---|---|
| Live variables | Dead code elimination | A dead variable is erroneously considered live | A live variable is erroneously considered dead |

# Context Determines the Validity of Approximations

May prohibit correct optimization

May enable wrong optimization

| Analysis | Application | Safe Approximation | Exhaustive Approximation |
|---|---|---|---|
| Live variables | Dead code elimination | A dead variable is erroneously considered live | A live variable is erroneously considered dead |
| Available expressions | Common subexpression elimination | An available expression is erroneously considered non-available | A non-available expression is erroneously considered available |

# Context Determines the Validity of Approximations

May prohibit correct optimization

May enable wrong optimization

| Analysis | Application | Safe Approximation | Exhaustive Approximation |
|---|---|---|---|
| Live variables | Dead code elimination | A dead variable is erroneously considered live | A live variable is erroneously considered dead |
| Available expressions | Common subexpression elimination | An available expression is erroneously considered non-available | A non-available expression is erroneously considered available |

**Spurious Inclusion**

**Spurious Exclusion**

# Partial Order Captures Approximation

- $\sqsubseteq$ captures valid approximations for safety

  $x \sqsubseteq y \;\Rightarrow\; x$ is *weaker than y*

  - The data flow information represented by x can be safely used in place of the data flow information represented by y
  - It may be imprecise, though.

# Partial Order Captures Approximation

- $\sqsubseteq$ captures valid approximations for safety

  $x \sqsubseteq y \;\Rightarrow\; x$ is *weaker than* $y$

  - The data flow information represented by x can be safely used in place of the data flow information represented by y
  - It may be imprecise, though.

- $\sqsupseteq$ captures valid approximations for exhaustiveness

  $x \sqsupseteq y \;\Rightarrow\; x$ is *stronger than* $y$

  - The data flow information represented by x contains every value contained in the data flow information represented by y
  - It may be unsafe, though.

# Partial Order Captures Approximation

- $\sqsubseteq$ captures valid approximations for safety

  $x \sqsubseteq y \;\Rightarrow\; x$ is *weaker than* $y$

  - The data flow information represented by x can be safely used in place of the data flow information represented by y
  - It may be imprecise, though.

- $\sqsupseteq$ captures valid approximations for exhaustiveness

  $x \sqsupseteq y \;\Rightarrow\; x$ is *stronger than* $y$

  - The data flow information represented by x contains every value contained in the data flow information represented by y
  - It may be unsafe, though.

*We want most exhaustive information which is also safe.*

# Most Approximate Values in a Complete Lattice

- *Top*. $\forall x \in L,\ x \sqsubseteq \top$. The most exhaustive value.

- *Bottom*. $\forall x \in L,\ \bot \sqsubseteq x$. The safest value.

# Most Approximate Values in a Complete Lattice

- *Top*. $\forall x \in L,\ x \sqsubseteq \top$. The most exhaustive value.
  - Using $\top$ in place of any data flow value will never miss out (or rule out) any possible value.

- *Bottom*. $\forall x \in L,\ \bot \sqsubseteq x$. The safest value.

# Most Approximate Values in a Complete Lattice

- *Top*. $\forall x \in L,\ x \sqsubseteq \top$. The most exhaustive value.
  - Using $\top$ in place of any data flow value will never miss out (or rule out) any possible value.
  - The consequences may be sematically *unsafe*, or *incorrect*.
- *Bottom*. $\forall x \in L,\ \bot \sqsubseteq x$. The safest value.

# Most Approximate Values in a Complete Lattice

- *Top*. $\forall x \in L,\ x \sqsubseteq \top$. The most exhaustive value.
  - Using $\top$ in place of any data flow value will never miss out (or rule out) any possible value.
  - The consequences may be sematically *unsafe*, or *incorrect*.
- *Bottom*. $\forall x \in L,\ \bot \sqsubseteq x$. The safest value.
  - Using $\bot$ in place of any data flow value will never be *unsafe*, or *incorrect*.

# Most Approximate Values in a Complete Lattice

- *Top*. $\forall x \in L,\ x \sqsubseteq \top$. The most exhaustive value.
  - Using $\top$ in place of any data flow value will never miss out (or rule out) any possible value.
  - The consequences may be sematically *unsafe*, or *incorrect*.
- *Bottom*. $\forall x \in L,\ \bot \sqsubseteq x$. The safest value.
  - Using $\bot$ in place of any data flow value will never be *unsafe*, or *incorrect*.
  - The consequences may be *undefined* or *useless* because this replacement might miss out valid values.

# Most Approximate Values in a Complete Lattice

- *Top*. $\forall x \in L,\ x \sqsubseteq \top$. The most exhaustive value.
  - Using $\top$ in place of any data flow value will never miss out (or rule out) any possible value.
  - The consequences may be sematically *unsafe*, or *incorrect*.
- *Bottom*. $\forall x \in L,\ \perp \sqsubseteq x$. The safest value.
  - Using $\perp$ in place of any data flow value will never be *unsafe*, or *incorrect*.
  - The consequences may be *undefined* or *useless* because this replacement might miss out valid values.

*Appropriate orientation chosen by design.*

# Setting Up Lattices

## Available Expressions Analysis

$$
\begin{tikzcd}[column sep=small]
 & \{e_1, e_2, e_3\} \arrow[dl] \arrow[d] \arrow[dr] & \\
\{e_1, e_2\} \arrow[d] \arrow[dr] & \{e_1, e_3\} \arrow[dl] \arrow[dr] & \{e_2, e_3\} \arrow[dl] \arrow[d] \\
\{e_1\} \arrow[dr] & \{e_2\} \arrow[d] & \{e_3\} \arrow[dl] \\
 & \emptyset &
\end{tikzcd}
$$

$\sqsubseteq$ is $\subseteq$

$\sqcap$ is $\cap$

## Live Variables Analysis

$$
\begin{tikzcd}[column sep=small]
 & \emptyset \arrow[dl] \arrow[d] \arrow[dr] & \\
\{v_1\} \arrow[d] \arrow[dr] & \{v_2\} \arrow[dl] \arrow[dr] & \{v_3\} \arrow[dl] \arrow[d] \\
\{v_1, v_2\} \arrow[dr] & \{v_1, v_3\} \arrow[d] & \{v_2, v_3\} \arrow[dl] \\
 & \{v_1, v_2, v_3\} &
\end{tikzcd}
$$

$\sqsubseteq$ is $\supseteq$

$\sqcap$ is $\cup$

# Partial Order Relation

| | |
|---|---|
| Reflexive | $x \sqsubseteq x$ |
| Transitive | $x \sqsubseteq y, y \sqsubseteq z$<br>$\Rightarrow x \sqsubseteq z$ |
| Antisymmetric | $x \sqsubseteq y, y \sqsubseteq x$<br>$\Leftrightarrow x = y$ |

# Partial Order Relation

| | | |
|---|---|---|
| Reflexive | $x \sqsubseteq x$ | $x$ can be safely used in place of $x$ |
| Transitive | $x \sqsubseteq y, y \sqsubseteq z \Rightarrow x \sqsubseteq z$ | If $x$ can be safely used in place of $y$ and $y$ can be safely used in place of $z$, then $x$ can be safely used in place of $z$ |
| Antisymmetric | $x \sqsubseteq y, y \sqsubseteq x \Leftrightarrow x = y$ | If $x$ can be safely used in place of $y$ and $y$ can be safely used in place of $x$, then $x$ must be same as $y$ |

# Merging Information

- $x \sqcap y$ computes the *greatest lower bound* of $x$ and $y$ i.e. largest $z$ such that $z \sqsubseteq x$ and $z \sqsubseteq y$

  The largest safe approximation of combining data flow information $x$ and $y$

# Merging Information

- $x \sqcap y$ computes the *greatest lower bound* of $x$ and $y$ i.e. largest $z$ such that $z \sqsubseteq x$ and $z \sqsubseteq y$

  The largest safe approximation of combining data flow information $x$ and $y$

- Commutative $x \sqcap y = y \sqcap x$

  Associative $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$

  Idempotent $x \sqcap x = x$

# Merging Information

- $x \sqcap y$ computes the *greatest lower bound* of $x$ and $y$ i.e. largest $z$ such that $z \sqsubseteq x$ and $z \sqsubseteq y$

  The largest safe approximation of combining data flow information $x$ and $y$

- | | | |
  |---|---|---|
  | Commutative | $x \sqcap y = y \sqcap x$ | The order in which the data flow information is merged, does not matter |
  | Associative | $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$ | Allow n-ary merging without any restriction on the order |
  | Idempotent | $x \sqcap x = x$ | No loss of information if $x$ is merged with itself |

# Merging Information

- $x \sqcap y$ computes the *greatest lower bound* of $x$ and $y$ i.e. largest $z$ such that $z \sqsubseteq x$ and $z \sqsubseteq y$

  The largest safe approximation of combining data flow information $x$ and $y$

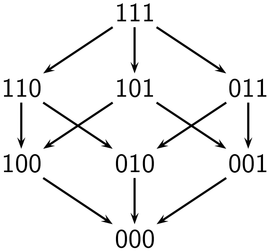
- Commutative $x \sqcap y = y \sqcap x$ The order in which the data flow information is merged, does not matter

  Associative $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$ Allow n-ary merging without any restriction on the order

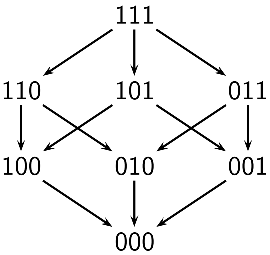
  Idempotent $x \sqcap x = x$ No loss of information if $x$ is merged with itself

- $x \sqcap \top = x$ (ensures exhaustiveness)
  $x \sqcap \bot = \bot$ (ensures safety)

# More on Lattices in Data Flow Analysis

| $L$ = Lattice for all expressions | $\widehat{L}$ = Lattice for a single expression |
|---|---|
|   | (Expression $e$ is available)<br>1 or $\{e\}$<br>$\downarrow$<br>0 or $\emptyset$<br>(Expressions $e$ is not available) |

# More on Lattices in Data Flow Analysis

| $L$ = Lattice for all expressions | $\widehat{L}$ = Lattice for a single expression |
|---|---|
|  | (Expression $e$ is available)<br>1 or $\{e\}$<br>$\downarrow$<br>0 or $\emptyset$<br>(Expressions $e$ is not available) |



Cartesian products if sets are used, vectors (or tuples) if bit are used.

- $L = \widehat{L} \times \widehat{L} \times \widehat{L}$ and $x = \langle \widehat{x}_1, \widehat{x}_2, \widehat{x}_3 \rangle \in L$ where $\widehat{x}_i \in \widehat{L}$
- $\sqsubseteq = \widehat{\sqsubseteq} \times \widehat{\sqsubseteq} \times \widehat{\sqsubseteq}$ and $\sqcap = \widehat{\sqcap} \times \widehat{\sqcap} \times \widehat{\sqcap}$
- $\top = \widehat{\top} \times \widehat{\top} \times \widehat{\top}$ and $\bot = \widehat{\bot} \times \widehat{\bot} \times \widehat{\bot}$

# Component Lattice for Data Flow Information Represented By Bit Vectors

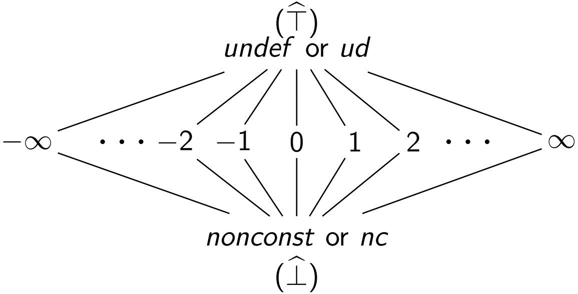
$(\widehat{\top})$
1
|
0
$(\widehat{\bot})$

$\sqcap$ is $\cap$ or Boolean AND

$(\widehat{\top})$
0
|
1
$(\widehat{\bot})$

$\sqcap$ is $\cup$ or Boolean OR

# Component Lattice for Integer Constant Propagation




- Overall lattice $L$ is the product of $\widehat{L}$ for all variables.
- $\sqcap$ and $\widehat{\sqcap}$ get defined by $\sqsubseteq$ and $\widehat{\sqsubseteq}$.

| $\widehat{\sqcap}$ | $\langle a, ud \rangle$ | $\langle a, nc \rangle$ | $\langle a, c_1 \rangle$ |
|---|---|---|---|
| $\langle a, ud \rangle$ | $\langle a, ud \rangle$ | $\langle a, nc \rangle$ | $\langle a, c_1 \rangle$ |
| $\langle a, nc \rangle$ | $\langle a, nc \rangle$ | $\langle a, nc \rangle$ | $\langle a, nc \rangle$ |
| $\langle a, c_2 \rangle$ | $\langle a, c_2 \rangle$ | $\langle a, nc \rangle$ | If $c_1 = c_2$ then $\langle a, c_1 \rangle$ else $\langle a, nc \rangle$ |

*Part 5*

# *Flow Functions*

# Flow Functions

- Computing data flow values from local effects
- Properties of flow functions

# The Set of Flow Functions

- $F$ is the set of functions $f : L \mapsto L$ such that
  - $F$ contains an identity function

    To model "empty" statements, i.e. statements which do not influence the data flow information
  - $F$ is closed under composition

    Cumulative effect of statements should generate data flow information from the same set.
- Properties of $f$
  - Monotonicity and Distributivity
  - Loop Closure Boundedness

# Flow Functions in Bit Vector Data Flow Frameworks

- Bit Vector Frameworks: Available Expressions Analysis, Reaching Definitions Analysis Live variable Analysis, Very Busy Expressions Analysis, Partial Redundancy Elimination etc.
  - All functions can be defined in terms of constant Gen and Kill

$$f(x) = \text{Gen} \cup (x - \text{Kill})$$

  - Lattices are powersets with partial orders as $\subseteq$ or $\supseteq$ relations
  - Information is merged using $\cap$ or $\cup$

# Flow Functions in Bit Vector Data Flow Frameworks

- Bit Vector Frameworks: Available Expressions Analysis, Reaching Definitions Analysis Live variable Analysis, Very Busy Expressions Analysis, Partial Redundancy Elimination etc.
  - All functions can be defined in terms of constant Gen and Kill

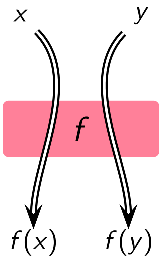
$$f(x) = \text{Gen} \cup (x - \text{Kill})$$

  - Lattices are powersets with partial orders as $\subseteq$ or $\supseteq$ relations
  - Information is merged using $\cap$ or $\cup$
- Flow functions in Faint Variables Analysis, Pointer Analyses, Constant Propagation, Possibly Uninitialized Variables cannot be expressed using constant Gen and Kill.

  Local context alone is not sufficient to describe the effect of statements fully.

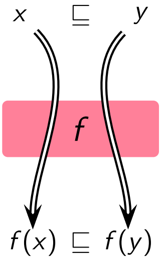
# Monotonicity of Flow Functions

- Partial order is preserved: If $x$ can be safely used in place of $y$ then $f(x)$ can be safely used in place of $f(y)$



# Monotonicity of Flow Functions

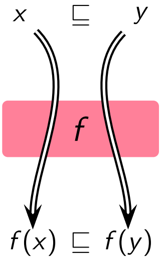
- Partial order is preserved: If $x$ can be safely used in place of $y$ then $f(x)$ can be safely used in place of $f(y)$



# Monotonicity of Flow Functions

- Partial order is preserved: If $x$ can be safely used in place of $y$ then $f(x)$ can be safely used in place of $f(y)$

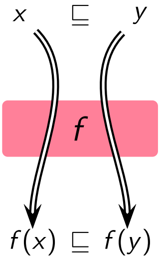
$$\forall x, y \in L, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$



# Monotonicity of Flow Functions

- Partial order is preserved: If $x$ can be safely used in place of $y$ then $f(x)$ can be safely used in place of $f(y)$

$$\forall x, y \in L, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$



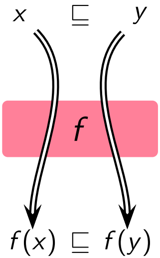

- Alternative definition

$$\forall x, y \in L, f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)$$

# Monotonicity of Flow Functions

- Partial order is preserved: If $x$ can be safely used in place of $y$ then $f(x)$ can be safely used in place of $f(y)$

$$\forall x, y \in L, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$



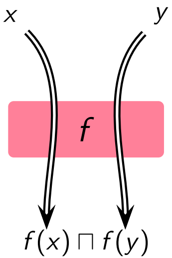

- Alternative definition

$$\forall x, y \in L, f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)$$

- Merging at intermediate points in shared segments of paths is safe (However, it may lead to imprecision).

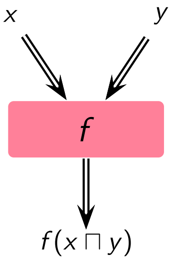
# Distributivity of Flow Functions

- Merging distributes over function application



# Distributivity of Flow Functions

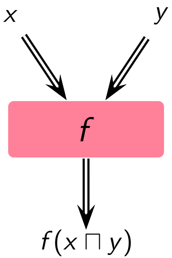
- Merging distributes over function application



# Distributivity of Flow Functions

- Merging distributes over function application

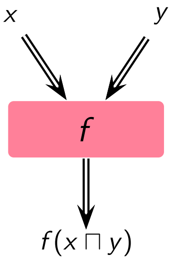
$$\forall x, y \in L, x \sqsubseteq y \Rightarrow f(x \sqcap y) = f(x) \sqcap f(y)$$



# Distributivity of Flow Functions

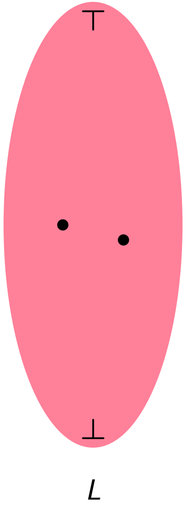
- Merging distributes over function application

$$\forall x, y \in L, x \sqsubseteq y \Rightarrow f(x \sqcap y) = f(x) \sqcap f(y)$$

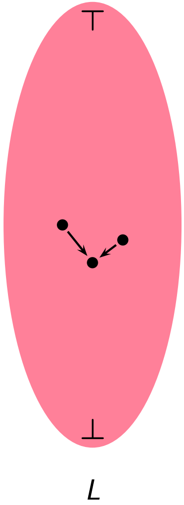



- Merging at intermediate points in shared segments of paths does not lead to imprecision.

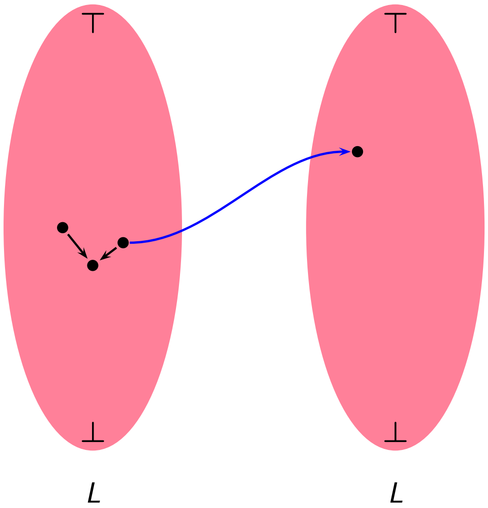
# Monotonicity and Distributivity



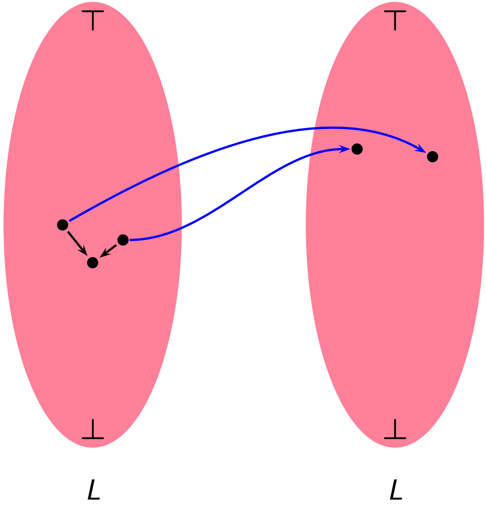
# Monotonicity and Distributivity



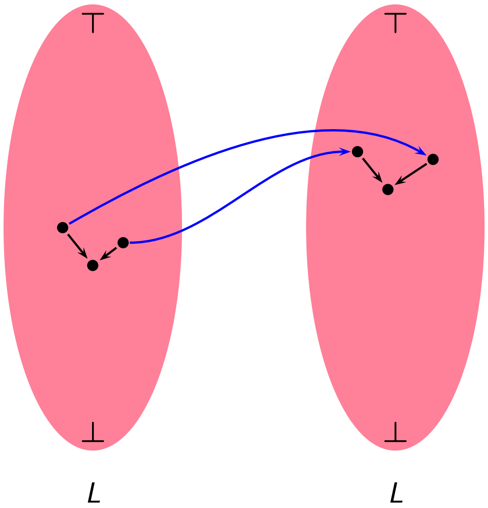
# Monotonicity and Distributivity



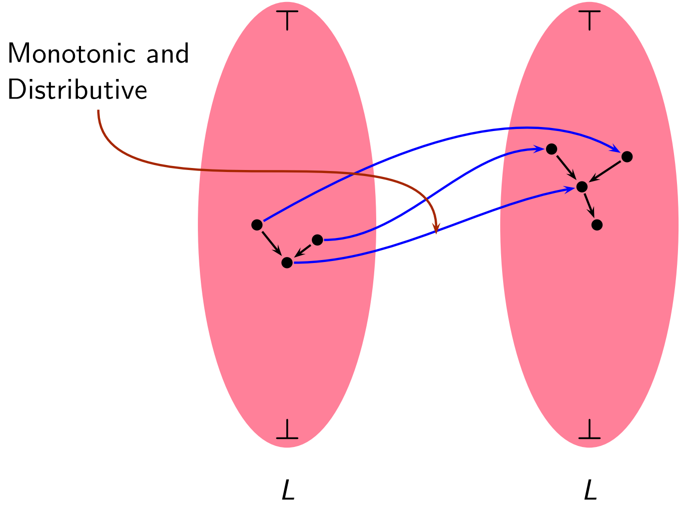
# Monotonicity and Distributivity



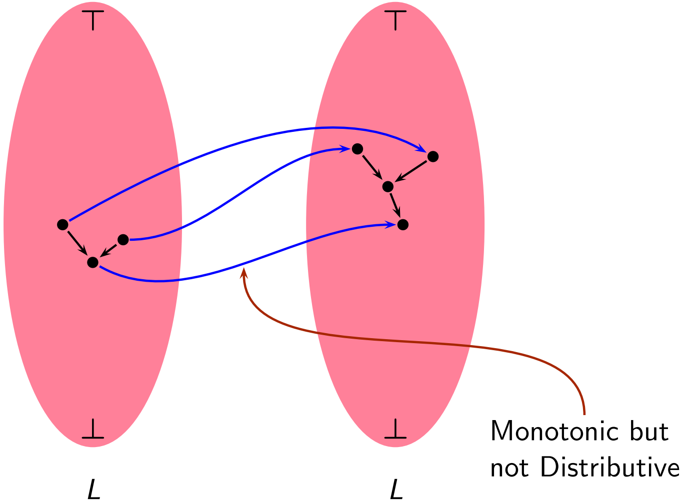
# Monotonicity and Distributivity



# Monotonicity and Distributivity



# Monotonicity and Distributivity



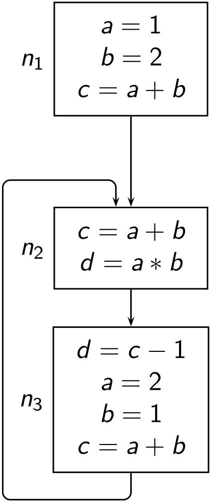
# Distributivity of Bit Vector Frameworks

$$\begin{aligned} f(x) &= \text{Gen} \cup (x - \text{Kill}) \\ f(y) &= \text{Gen} \cup (y - \text{Kill}) \end{aligned}$$

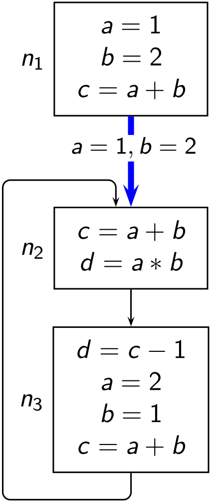
$$\begin{aligned} f(x \cup y) &= \text{Gen} \cup ((x \cup y) - \text{Kill}) \\ &= \text{Gen} \cup ((x - \text{Kill}) \cup (y - \text{Kill})) \\ &= (\text{Gen} \cup (x - \text{Kill}) \cup \text{Gen} \cup (y - \text{Kill})) \\ &= f(x) \cup f(y) \end{aligned}$$

$$\begin{aligned} f(x \cap y) &= \text{Gen} \cup ((x \cap y) - \text{Kill}) \\ &= \text{Gen} \cup ((x - \text{Kill}) \cap (y - \text{Kill})) \\ &= (\text{Gen} \cup (x - \text{Kill}) \cap \text{Gen} \cup (y - \text{Kill})) \\ &= f(x) \cap f(y) \end{aligned}$$

# Non-Distributivity of Constant Propagation

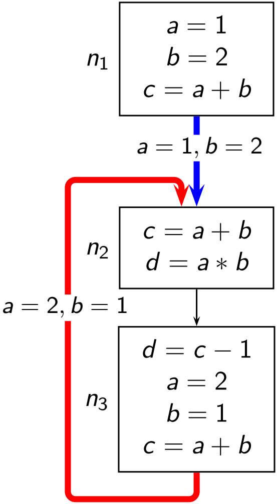


# Non-Distributivity of Constant Propagation



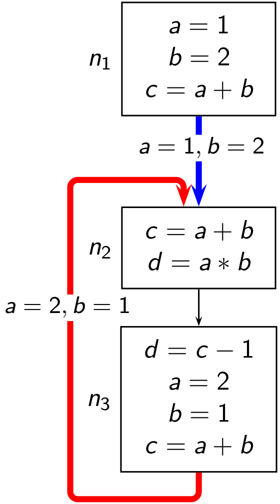

- $x = \langle 1, 2, 3, ? \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)

# Non-Distributivity of Constant Propagation




- $x = \langle 1, 2, 3, ? \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)
- $y = \langle 2, 1, 3, 2 \rangle$ (Along $Out_{n_3} \rightarrow In_{n_2}$)

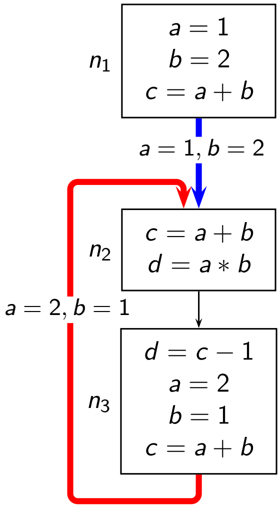
# Non-Distributivity of Constant Propagation




- $x = \langle 1, 2, 3, ? \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)
- $y = \langle 2, 1, 3, 2 \rangle$ (Along $Out_{n_3} \rightarrow In_{n_2}$)
- Function application before merging

$$\begin{aligned} f(x) \sqcap f(y) &= f(\langle 1, 2, 3, ? \rangle) \sqcap f(\langle 2, 1, 3, 2 \rangle) \\ &= \langle 1, 2, 3, 2 \rangle \sqcap \langle 2, 1, 3, 2 \rangle \\ &= \langle \widehat{\perp}, \widehat{\perp}, 3, 2 \rangle \end{aligned}$$

# Non-Distributivity of Constant Propagation



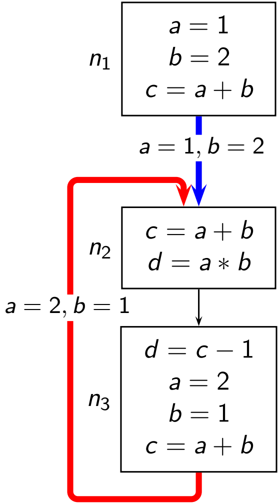

- $x = \langle 1, 2, 3, ? \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)
- $y = \langle 2, 1, 3, 2 \rangle$ (Along $Out_{n_3} \rightarrow In_{n_2}$)
- Function application before merging

$$\begin{aligned} f(x) \sqcap f(y) &= f(\langle 1, 2, 3, ? \rangle) \sqcap f(\langle 2, 1, 3, 2 \rangle) \\ &= \langle 1, 2, 3, 2 \rangle \sqcap \langle 2, 1, 3, 2 \rangle \\ &= \langle \widehat{\perp}, \widehat{\perp}, 3, 2 \rangle \end{aligned}$$

- Function application after merging

$$\begin{aligned} f(x \sqcap y) &= f(\langle 1, 2, 3, ? \rangle \sqcap \langle 2, 1, 3, 2 \rangle) \\ &= f(\langle \widehat{\perp}, \widehat{\perp}, 3, 2 \rangle) \\ &= \langle \widehat{\perp}, \widehat{\perp}, \widehat{\perp}, \widehat{\perp} \rangle \end{aligned}$$

# Non-Distributivity of Constant Propagation




- $x = \langle 1, 2, 3, ? \rangle$ (Along $Out_{n_1} \rightarrow In_{n_2}$)
- $y = \langle 2, 1, 3, 2 \rangle$ (Along $Out_{n_3} \rightarrow In_{n_2}$)
- Function application before merging

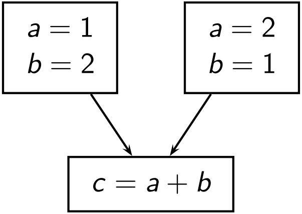
$$\begin{aligned} f(x) \sqcap f(y) &= f(\langle 1, 2, 3, ? \rangle) \sqcap f(\langle 2, 1, 3, 2 \rangle) \\ &= \langle 1, 2, 3, 2 \rangle \sqcap \langle 2, 1, 3, 2 \rangle \\ &= \langle \widehat{\perp}, \widehat{\perp}, 3, 2 \rangle \end{aligned}$$

- Function application after merging

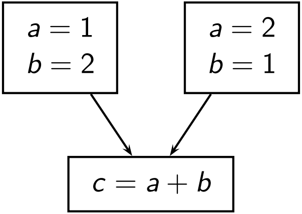
$$\begin{aligned} f(x \sqcap y) &= f(\langle 1, 2, 3, ? \rangle \sqcap \langle 2, 1, 3, 2 \rangle) \\ &= f(\langle \widehat{\perp}, \widehat{\perp}, 3, 2 \rangle) \\ &= \langle \widehat{\perp}, \widehat{\perp}, \widehat{\perp}, \widehat{\perp} \rangle \end{aligned}$$

- $f(x \sqcap y) \sqsubset f(x) \sqcap f(y)$

# Why is Constant Propagation Non-Distribitive?



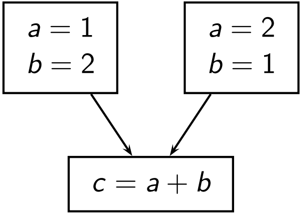
# Why is Constant Propagation Non-Distributive?



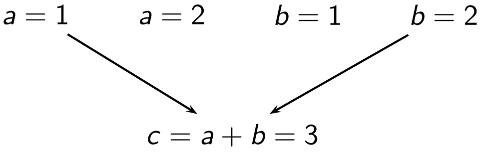

Possible combinations due to merging

$a = 1$ $a = 2$ $b = 1$ $b = 2$

# Why is Constant Propagation Non-Distribitive?

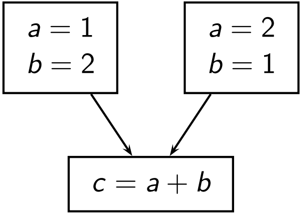



Possible combinations due to merging




- Correct combination.

# Why is Constant Propagation Non-Distribitive?



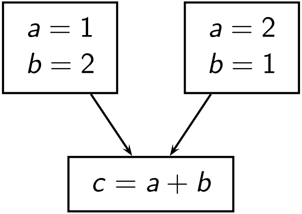

Possible combinations due to merging

$a = 1$ $\quad a = 2$ $\quad b = 1$ $\quad b = 2$

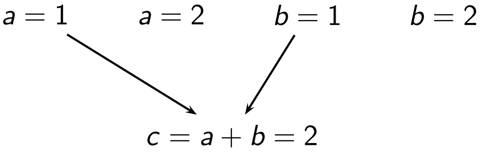
$c = a + b = 3$

- Correct combination.

# Why is Constant Propagation Non-Distribitive?

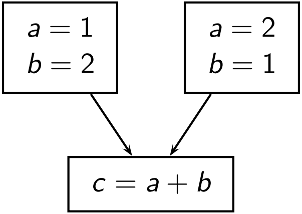



Possible combinations due to merging

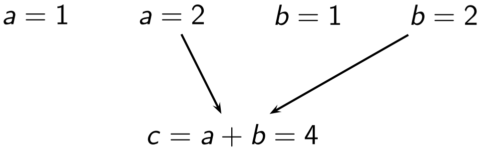



- Wrong combination.
- Mutually exclusive information.
- No execution path along which this information holds.

# Why is Constant Propagation Non-Distribitive?




Possible combinations due to merging




- Wrong combination.
- Mutually exclusive information.
- No execution path along which this information holds.

Part 6

# Solutions of Data Flow Analysis

# Solutions of Data Flow Analysis

- Characterizing solutions

  Desirable solutions and computable solutions

- Existence and computability

# Solutions of Data Flow Analysis

- An assignment $A$ associates data flow values with program points. $A \sqsubseteq B$ if for all program points $p$, $A(p) \sqsubseteq B(p)$
- Performing data flow analysis

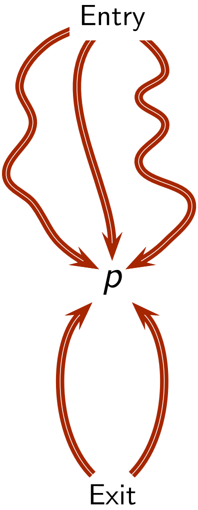
  Given

  - A set of flow functions, a lattice, and merge operation
  - A program flow graph with a mapping from nodes to flow functions

  Find out

  - An assignment $A$ which is as exhaustive as possible and is safe

# Meet Over Paths (MoP) Assignment

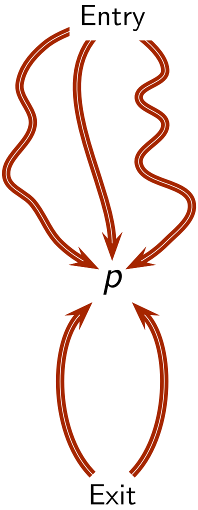



- The largest safe approximation of the information reaching a program point along all information flow paths.

$$MoP(p) \quad = \bigsqcap_{\rho \in Paths(p)} f_\rho(BoundaryInfo)$$

  - $f_\rho$ represents the compositions of flow functions along $\rho$.
  - *BoundaryInfo* refers to the relevant information from the calling context.
  - All execution paths are considered potentially executable by ignoring the results of conditionals.

# Meet Over Paths (MoP) Assignment




- The largest safe approximation of the information reaching a program point along all information flow paths.

$$MoP(p) \quad = \bigsqcap_{\rho \in Paths(p)} f_\rho(BoundaryInfo)$$

  - $f_\rho$ represents the compositions of flow functions along $\rho$.
  - *BoundaryInfo* refers to the relevant information from the calling context.
  - All execution paths are considered potentially executable by ignoring the results of conditionals.

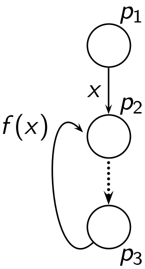
- Any $Info(p) \sqsubseteq MoP(p)$ is safe.

# Existence of an MoP Assignment

$$MoP(p) \quad = \bigsqcap_{\rho \in Paths(p)} f_\rho(BoundaryInfo)$$

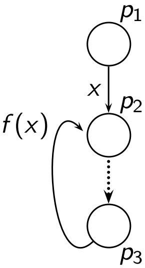
- If all paths reaching $p$ are acyclic, then existence of solution trivially follows from the definition of the function space.
- If cyclic paths also reach $p$, then there are an infinite number of unbounded paths.
  $\Rightarrow$ Need to define loop closures.

# Loop Closures of Flow Functions




| Paths Terminating at $p_2$ | Data Flow Value |
|---|---|
| $p_1, p_2$ | $x$ |
| $p_1, p_2, p_3, p_2$ | $f(x)$ |
| $p_1, p_2, p_3, p_2, p_3, p_2$ | $f(f(x)) = f^2(x)$ |
| $p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$ | $f(f(f(x))) = f^3(x)$ |
| $\ldots$ | $\ldots$ |

# Loop Closures of Flow Functions

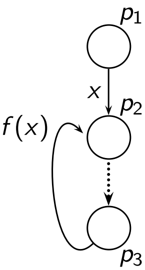



| Paths Terminating at $p_2$ | Data Flow Value |
|---|---|
| $p_1, p_2$ | $x$ |
| $p_1, p_2, p_3, p_2$ | $f(x)$ |
| $p_1, p_2, p_3, p_2, p_3, p_2$ | $f(f(x)) = f^2(x)$ |
| $p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$ | $f(f(f(x))) = f^3(x)$ |
| ... | ... |

- For static analysis we need to summarize the value at $p_2$ by a value which is safe after any iteration.

$$f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap f^4(x) \sqcap \ldots$$

# Loop Closures of Flow Functions




| Paths Terminating at $p_2$ | Data Flow Value |
|---|---|
| $p_1, p_2$ | $x$ |
| $p_1, p_2, p_3, p_2$ | $f(x)$ |
| $p_1, p_2, p_3, p_2, p_3, p_2$ | $f(f(x)) = f^2(x)$ |
| $p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_2$ | $f(f(f(x))) = f^3(x)$ |
| ... | ... |

- For static analysis we need to summarize the value at $p_2$ by a value which is safe after any iteration.

$$f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap f^4(x) \sqcap \ldots$$

- $f^*$ is called the loop closure of $f$.

# Bounded Loop Closures

- The loop closure of $f$ is bounded if there exists a $k$ such that

$$\forall x \in L,\; f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap \ldots \sqcap f^{k-1}(x)$$

# Bounded Loop Closures

- The loop closure of $f$ is bounded if there exists a $k$ such that

$$\forall x \in L,\ f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap \ldots \sqcap f^{k-1}(x)$$

- If all descending chains are finite, then loop closures are bounded.

  The lattice may have infinite elements, e.g. in Constant Propagation

# Maximum Fixed Point (MFP) Assignment

- Difficulties in computing MoP assignment

# Maximum Fixed Point (MFP) Assignment

- Difficulties in computing MoP assignment
  - In the presence of cycles there are infinite paths

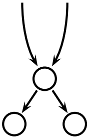
    If all paths need to be traversed $\Rightarrow$ Undecidability

# Maximum Fixed Point (MFP) Assignment

- Difficulties in computing MoP assignment
  - In the presence of cycles there are infinite paths

    If all paths need to be traversed ⇒ Undecidability
  - Even if a program is acyclic, every conditional multiplies the number of paths by two

    If all paths need to be traversed ⇒ Intractability



# Maximum Fixed Point (MFP) Assignment

- Difficulties in computing MoP assignment
  - In the presence of cycles there are infinite paths

    If all paths need to be traversed ⇒ Undecidability
  - Even if a program is acyclic, every conditional multiplies the number of paths by two

    If all paths need to be traversed ⇒ Intractability
- Why not merge information at intermediate points?
  - Merging is safe but may lead to imprecision.
  - Computes fixed point solutions of data flow equations.

# Maximum Fixed Point (MFP) Assignment

- Difficulties in computing MoP assignment
  - In the presence of cycles there are infinite paths

    If all paths need to be traversed ⇒ Undecidability
  - Even if a program is acyclic, every conditional multiplies the number of paths by two

    If all paths need to be traversed ⇒ Intractability
- Why not merge information at intermediate points?
  - Merging is safe but may lead to imprecision.
  - Computes fixed point solutions of data flow equations.

Path based specification

Edge based specifications

# Undecidability of Data Flow Analysis

- Reducing MPCP (Modified Post's Correspondence Problem) to constant propagation
- MPCP is known to be undecidable
- If an algorithm exists for detecting all constants
  $\Rightarrow$ MPCP would be decidable
- Since MPCP is undecidable
  $\Rightarrow$ There does not exist an algorithm for detecting all constants
  $\Rightarrow$ Static analysis is undecidable

# Fixed Points Computation: Flow Functions Vs. Equations

- Consider a CFG with N nodes.
  Let $\mathcal{X}$ be a vector $\langle In_1, Out_1, \ldots, In_N, Out_N \rangle$

# Fixed Points Computation: Flow Functions Vs. Equations

- Consider a CFG with N nodes.
  Let $\mathcal{X}$ be a vector $\langle In_1, Out_1, \ldots, In_N, Out_N \rangle$
- The data flow equations are:

$$
\begin{array}{lcllcl}
In_1 & = & f_{In_1}(\mathcal{X}) & Out_1 & = & f_{Out_1}(\mathcal{X}) \\
In_2 & = & f_{In_2}(\mathcal{X}) & Out_2 & = & f_{Out_2}(\mathcal{X}) \\
\ldots & & & & & \\
In_N & = & f_{In_N}(\mathcal{X}) & Out_N & = & f_{Out_N}(\mathcal{X})
\end{array}
$$

where $f_p : \mathcal{L} \mapsto L$ and $\mathcal{L}$ is an 2N-way product $L \times L \times \ldots \times L$

# Fixed Points Computation: Flow Functions Vs. Equations

- Consider a CFG with N nodes.
  Let $\mathcal{X}$ be a vector $\langle In_1, Out_1, \ldots, In_N, Out_N \rangle$
- The data flow equations are:

$$\begin{array}{rclcrcl} In_1 & = & f_{In_1}(\mathcal{X}) & \quad & Out_1 & = & f_{Out_1}(\mathcal{X}) \\ In_2 & = & f_{In_2}(\mathcal{X}) & & Out_2 & = & f_{Out_2}(\mathcal{X}) \\ \ldots & & & & & & \\ In_N & = & f_{In_N}(\mathcal{X}) & & Out_N & = & f_{Out_N}(\mathcal{X}) \end{array}$$

  where $f_p : \mathcal{L} \mapsto L$ and $\mathcal{L}$ is an 2N-way product $L \times L \times \ldots \times L$
- They can be rewritten as $\mathcal{X} = \mathcal{F}(\mathcal{X})$ where $\mathcal{F} : \mathcal{L} \mapsto \mathcal{L}$ is

$$\mathcal{F}(\mathcal{X}) = \langle f_{In_1}(\mathcal{X}), f_{Out_1}(\mathcal{X}), \ldots, f_{In_N}(\mathcal{X}), f_{Out_N}(\mathcal{X}) \rangle$$

# Fixed Points Computation: Flow Functions Vs. Equations

- Consider a CFG with N nodes.
  Let $\mathcal{X}$ be a vector $\langle In_1, Out_1, \ldots, In_N, Out_N \rangle$
- The data flow equations are:

$$\begin{array}{rclcrcl} In_1 & = & f_{In_1}(\mathcal{X}) & & Out_1 & = & f_{Out_1}(\mathcal{X}) \\ In_2 & = & f_{In_2}(\mathcal{X}) & & Out_2 & = & f_{Out_2}(\mathcal{X}) \\ \ldots & & & & & & \\ In_N & = & f_{In_N}(\mathcal{X}) & & Out_N & = & f_{Out_N}(\mathcal{X}) \end{array}$$

  where $f_p : \mathcal{L} \mapsto L$ and $\mathcal{L}$ is an 2N-way product $L \times L \times \ldots \times L$
- They can be rewritten as $\mathcal{X} = \mathcal{F}(\mathcal{X})$ where $\mathcal{F} : \mathcal{L} \mapsto \mathcal{L}$ is

$$\mathcal{F}(\mathcal{X}) = \langle f_{In_1}(\mathcal{X}), f_{Out_1}(\mathcal{X}), \ldots, f_{In_N}(\mathcal{X}), f_{Out_N}(\mathcal{X}) \rangle$$

- We compute the fixed points of equation $\mathcal{X} = \mathcal{F}(\mathcal{X})$

# Tarski's Fixed Point Theorem

**f is not a flow function**

Given monotonic $f : L \mapsto L$ where $L$ is a complete lattice

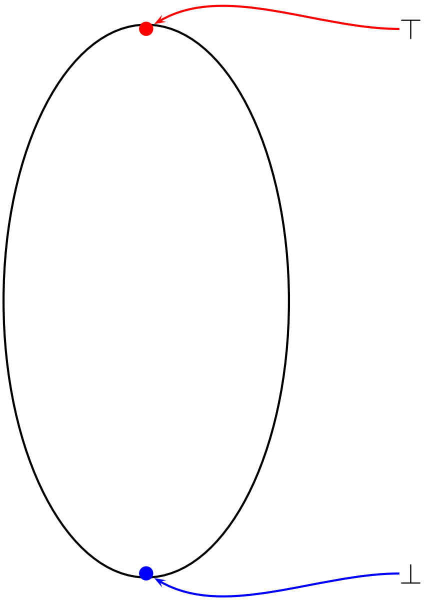
Define

$$
\begin{array}{lrcl}
p \text{ is a fixed point of } f: & Fix(f) & = & \{p \mid f(p) = p\} \\
f \text{ is reductive at } p: & Red(f) & = & \{p \mid f(p) \sqsubseteq p\} \\
f \text{ is extensive at } p: & Ext(f) & = & \{p \mid f(p) \sqsupseteq p\}
\end{array}
$$

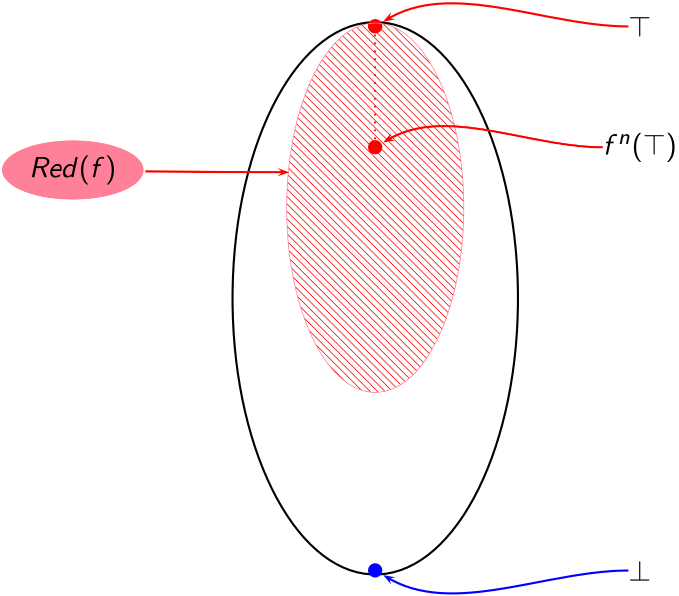
Then

$$
\begin{array}{rcl}
LFP(f) & = & \bigsqcap Red(f) \in Fix(f) \\
MFP(f) & = & \bigsqcup Ext(f) \in Fix(f)
\end{array}
$$

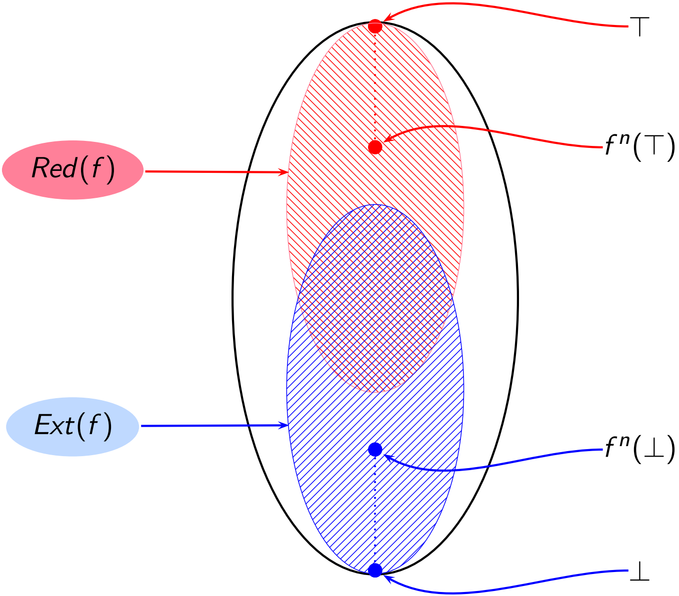
# Fixed Points of a Function



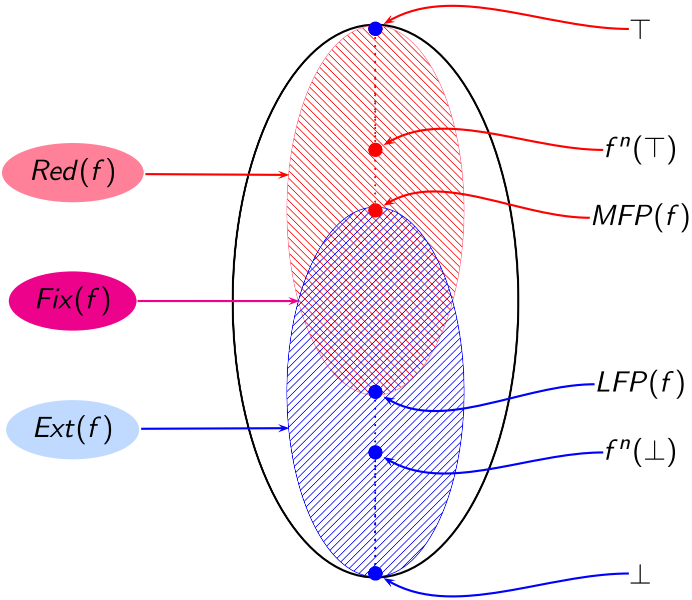
# Fixed Points of a Function



# Fixed Points of a Function

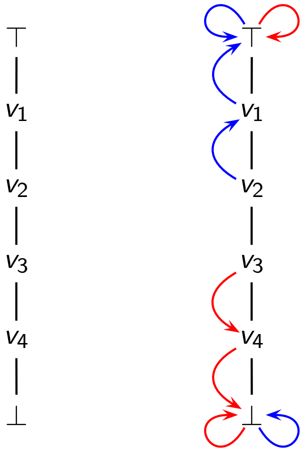


# Fixed Points of a Function



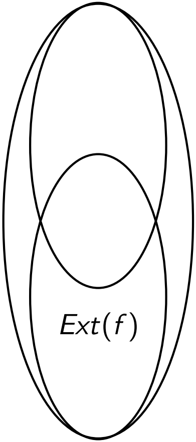
# Examples of Reductive and Extensive Sets

Finite $L$ Monotonic $f : L \mapsto L$

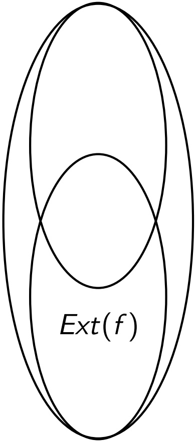



$$
\begin{aligned}
Red(f) &= \{\top, v_3, v_4, \bot\} \\
Ext(f) &= \{\top, v_1, v_2, \bot\} \\
Fix(f) &= Red(f) \cap Ext(f) \\
&= \{\top, \bot\} \\
MFP(f) &= lub\,(Ext(f)) \\
&= lub\,(Fix(f)) \\
&= \top \\
LFP(f) &= glb\,(Red(f)) \\
&= glb\,(Fix(f)) \\
&= \bot
\end{aligned}
$$

# Existence of MFP: Proof of Tarski's Fixed Point Theorem

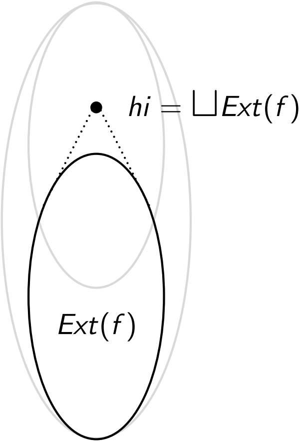


# Existence of MFP: Proof of Tarski's Fixed Point Theorem



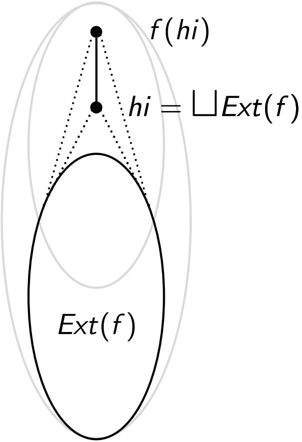

- Claim 1: Let $X \subseteq L$.
  $p \sqsupseteq x,\ \forall x \in X \Rightarrow p \sqsupseteq \bigsqcup(X)$.

# Existence of MFP: Proof of Tarski's Fixed Point Theorem



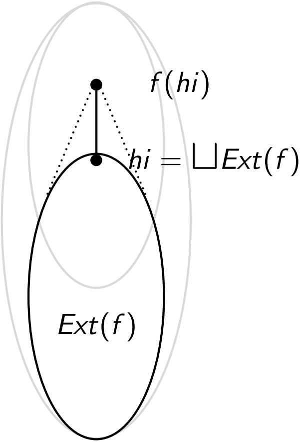

- Claim 1: Let $X \subseteq L$.
  $p \sqsupseteq x,\ \forall x \in X \Rightarrow p \sqsupseteq \bigsqcup(X)$.
- $\forall q \in Ext(f),\ hi \sqsupseteq q$

# Existence of MFP: Proof of Tarski's Fixed Point Theorem



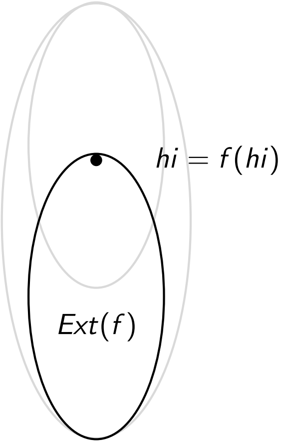

- Claim 1: Let $X \subseteq L$.
  $p \sqsupseteq x,\ \forall x \in X \Rightarrow p \sqsupseteq \bigsqcup(X)$.
- $\forall q \in Ext(f),\ hi \sqsupseteq q$
- $hi \sqsupseteq q$
  $\Rightarrow f(hi) \sqsupseteq f(q) \sqsupseteq q$ (monotonicity)
  $\Rightarrow f(hi) \sqsupseteq hi$ (claim 1, $hi$ is $\bigsqcup$)

# Existence of MFP: Proof of Tarski's Fixed Point Theorem



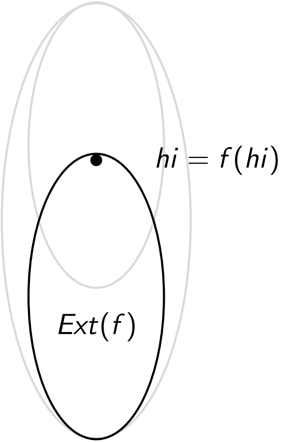

- Claim 1: Let $X \subseteq L$.
  $p \sqsupseteq x,\ \forall x \in X \Rightarrow p \sqsupseteq \bigsqcup(X)$.
- $\forall q \in Ext(f),\ hi \sqsupseteq q$
- $hi \sqsupseteq q$
  $\Rightarrow f(hi) \sqsupseteq f(q) \sqsupseteq q$ (monotonicity)
  $\Rightarrow f(hi) \sqsupseteq hi$ (claim 1, $hi$ is $\bigsqcup$)
- $f$ is extensive at $hi$ also: $hi \in Ext(f)$

# Existence of MFP: Proof of Tarski's Fixed Point Theorem




- Claim 1: Let $X \subseteq L$.
  $p \sqsupseteq x,\ \forall x \in X \Rightarrow p \sqsupseteq \bigsqcup(X)$.
- $\forall q \in Ext(f),\ hi \sqsupseteq q$
- $hi \sqsupseteq q$
  $\Rightarrow f(hi) \sqsupseteq f(q) \sqsupseteq q$ (monotonicity)
  $\Rightarrow f(hi) \sqsupseteq hi$ (claim 1, $hi$ is $\bigsqcup$)
- $f$ is extensive at $hi$ also: $hi \in Ext(f)$
- $f(hi) \sqsupseteq hi \Rightarrow f^2(hi) \sqsupseteq f(hi)$
  $\Rightarrow f(hi) \in Ext(f)$
  $\Rightarrow hi \sqsubseteq f(hi)$
  $\Rightarrow hi = f(hi) \Rightarrow hi \in Fix(f)$

# Existence of MFP: Proof of Tarski's Fixed Point Theorem



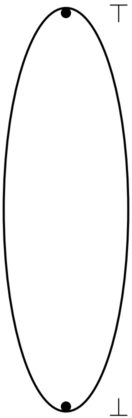

- Claim 1: Let $X \subseteq L$.
  $p \sqsupseteq x, \ \forall x \in X \Rightarrow p \sqsupseteq \bigsqcup(X)$.
- $\forall q \in Ext(f), \ hi \sqsupseteq q$
- $hi \sqsupseteq q$
  $\Rightarrow f(hi) \sqsupseteq f(q) \sqsupseteq q$ (monotonicity)
  $\Rightarrow f(hi) \sqsupseteq hi$ (claim 1, $hi$ is $\bigsqcup$)
- $f$ is extensive at $hi$ also: $hi \in Ext(f)$
- $f(hi) \sqsupseteq hi \Rightarrow f^2(hi) \sqsupseteq f(hi)$
  $\Rightarrow f(hi) \in Ext(f)$
  $\Rightarrow hi \sqsubseteq f(hi)$
  $\Rightarrow hi = f(hi) \Rightarrow hi \in Fix(f)$
- $Fix(f) \subseteq Ext(f) \Rightarrow hi \sqsupseteq p, \ \forall p \in Fix(f)$

# Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then $MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.

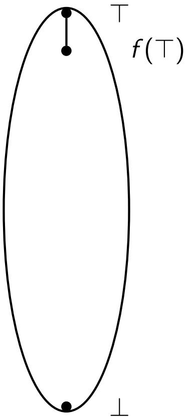
# Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then $MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.



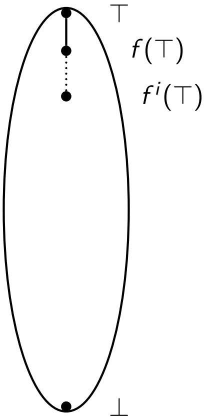
# Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then $MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.



# Computing the Maximum Fixed Point

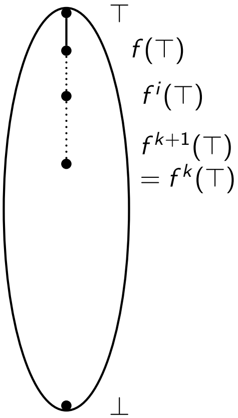
For monotonic $f : L \mapsto L$, if all descending chains are finite, then $MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.




- $\top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsupseteq f^3(\top) \sqsupseteq f^4(\top) \sqsupseteq \ldots$

# Computing the Maximum Fixed Point

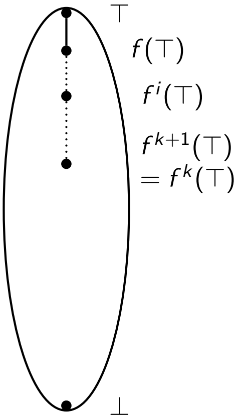
For monotonic $f : L \mapsto L$, if all descending chains are finite, then $MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.




- $\top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsupseteq f^3(\top) \sqsupseteq f^4(\top) \sqsupseteq \ldots$
- Since descending chains are finite, there must exist $f^k(\top)$ such that $f^{k+1}(\top) = f^k(\top)$ and $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.

# Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then
$MFP(f) = f^{k+1}(\top) = f^{k}(\top)$ such that $f^{j+1}(\top) \neq f^{j}(\top)$, $j < k$.

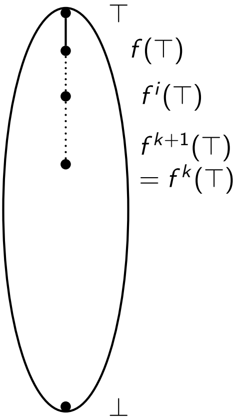



- $\top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsupseteq f^3(\top) \sqsupseteq f^4(\top) \sqsupseteq \ldots$
- Since descending chains are finite, there must exist $f^{k}(\top)$ such that $f^{k+1}(\top) = f^{k}(\top)$ and $f^{j+1}(\top) \neq f^{j}(\top)$, $j < k$.
- If $p$ is a fixed point of $f$ then $f^{k}(\top) \sqsupseteq p$.

  Proof strategy: Induction on $i$ for $f^{i}(\top)$

# Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then
$MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.



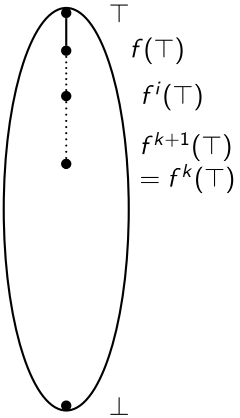

- $\top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsupseteq f^3(\top) \sqsupseteq f^4(\top) \sqsupseteq \ldots$
- Since descending chains are finite, there must exist $f^k(\top)$ such that $f^{k+1}(\top) = f^k(\top)$ and $f^{j+1}(\top) \neq f^j(\top)$, $j < k$.
- If $p$ is a fixed point of $f$ then $f^k(\top) \sqsupseteq p$.

  Proof strategy: Induction on $i$ for $f^i(\top)$

  - Basis ($i = 0$): $f^0(\top) = \top \sqsupseteq p$.
  - Inductive Hypothesis: Assume that $f^i(\top) \sqsupseteq p$.

# Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then
$MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top),\ j < k$.




- $\top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsupseteq f^3(\top) \sqsupseteq f^4(\top) \sqsupseteq \dots$
- Since descending chains are finite, there must exist $f^k(\top)$ such that $f^{k+1}(\top) = f^k(\top)$ and $f^{j+1}(\top) \neq f^j(\top),\ j < k$.
- If $p$ is a fixed point of $f$ then $f^k(\top) \sqsupseteq p$.

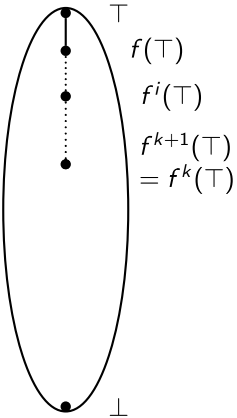
  Proof strategy: Induction on $i$ for $f^i(\top)$

  - Basis ($i = 0$): $f^0(\top) = \top \sqsupseteq p$.
  - Inductive Hypothesis: Assume that $f^i(\top) \sqsupseteq p$.
  - Proof:

$$
\begin{array}{rlll}
 & f(f^i(\top)) & \sqsupseteq & f(p) \quad (f \text{ is monotonic}) \\
\Rightarrow & f(f^i(\top)) & \sqsupseteq & p \quad\quad\ (f(p) = p) \\
\Rightarrow & f^{i+1}(\top) & \sqsupseteq & p
\end{array}
$$

# Computing the Maximum Fixed Point

For monotonic $f : L \mapsto L$, if all descending chains are finite, then $MFP(f) = f^{k+1}(\top) = f^k(\top)$ such that $f^{j+1}(\top) \neq f^j(\top),\ j < k$.




- $\top \sqsupseteq f(\top) \sqsupseteq f^2(\top) \sqsupseteq f^3(\top) \sqsupseteq f^4(\top) \sqsupseteq \ldots$
- Since descending chains are finite, there must exist $f^k(\top)$ such that $f^{k+1}(\top) = f^k(\top)$ and $f^{j+1}(\top) \neq f^j(\top),\ j < k$.
- If $p$ is a fixed point of $f$ then $f^k(\top) \sqsupseteq p$.

  Proof strategy: Induction on $i$ for $f^i(\top)$

  - Basis ($i = 0$): $f^0(\top) = \top \sqsupseteq p$.
  - Inductive Hypothesis: Assume that $f^i(\top) \sqsupseteq p$.
  - Proof:

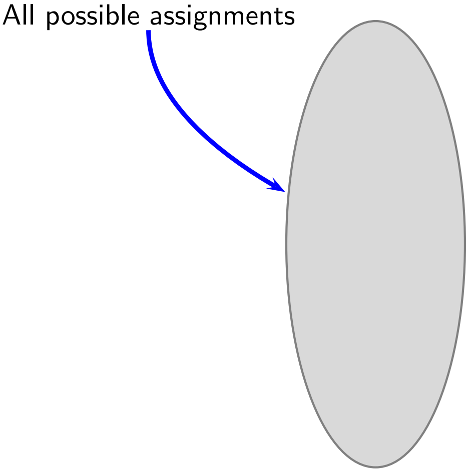
$$\begin{array}{cccll} & f(f^i(\top)) & \sqsupseteq & f(p) & (f \text{ is monotonic}) \\ \Rightarrow & f(f^i(\top)) & \sqsupseteq & p & (f(p) = p) \\ \Rightarrow & f^{i+1}(\top) & \sqsupseteq & p & \end{array}$$

- $\Rightarrow f^{k+1}(\top)$ is the MFP.

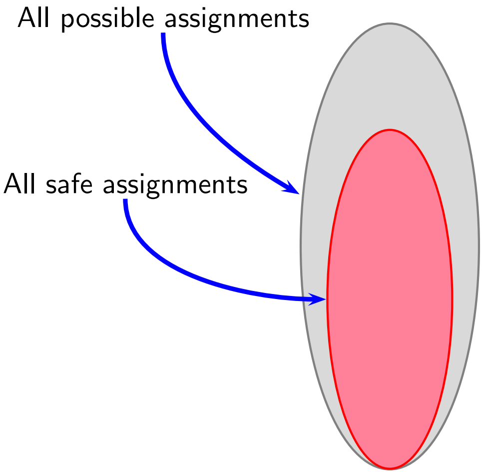
# Existence and Computation of the Maximum Fixed Point

- For monotonic $f : L \mapsto L$
  - Existence: $MFP(f) = \bigsqcup Ext(f) \in Fix(f)$
    Requires $L$ to be complete.
  - Computation: $MFP(f) = f^{k+1}(\top) = f^{k}(\top)$ such that $f^{j+1}(\top) \neq f^{j}(\top),\ j < k$.
    Requires all *strictly descending* chains to be finite.
- Finite strictly descending chains $\Rightarrow$ Completeness of lattice
  Completeness of lattice $\not\Rightarrow$ Finite strictly descending chains
  $\Rightarrow$ Even if MFP exists, it may not be reachable
  unless all strictly descending chains are finite.

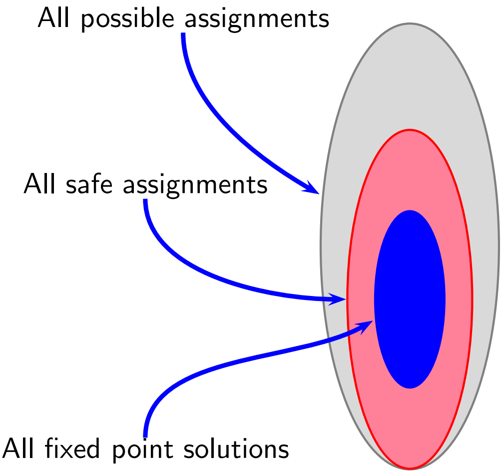
# Possible Assignments as Solutions of Data Flow Analyses



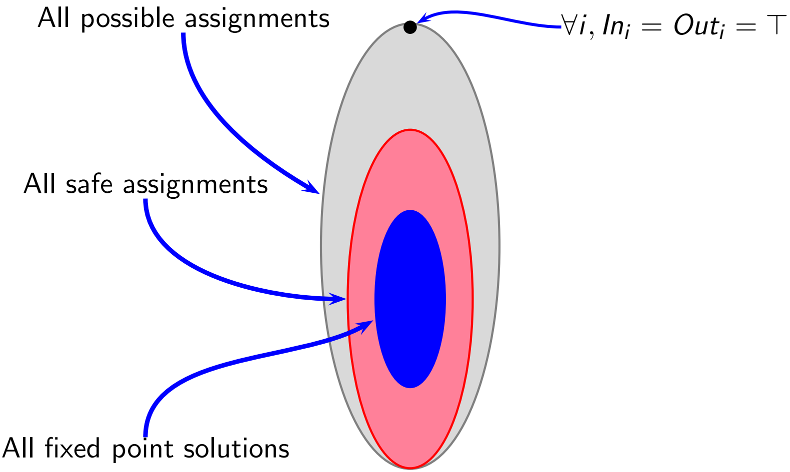
# Possible Assignments as Solutions of Data Flow Analyses



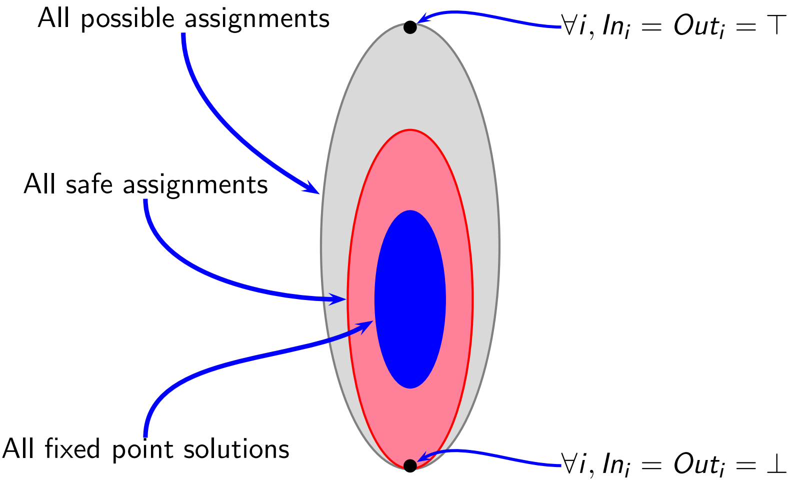
# Possible Assignments as Solutions of Data Flow Analyses



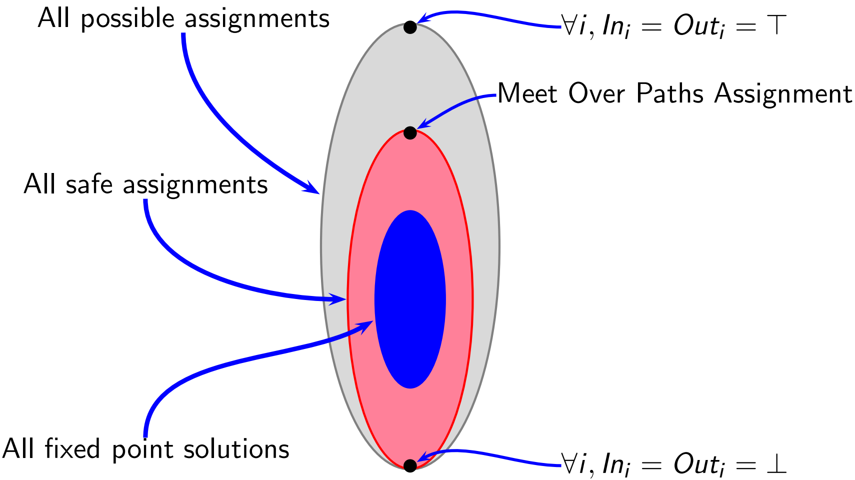
# Possible Assignments as Solutions of Data Flow Analyses



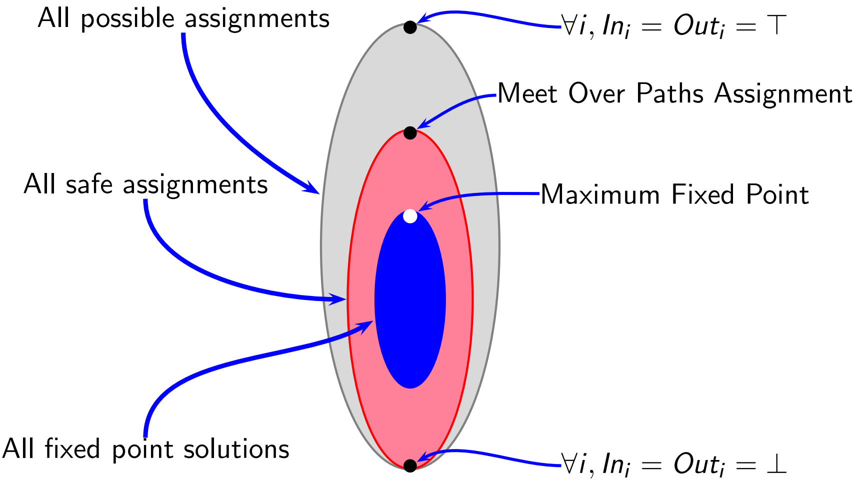
# Possible Assignments as Solutions of Data Flow Analyses



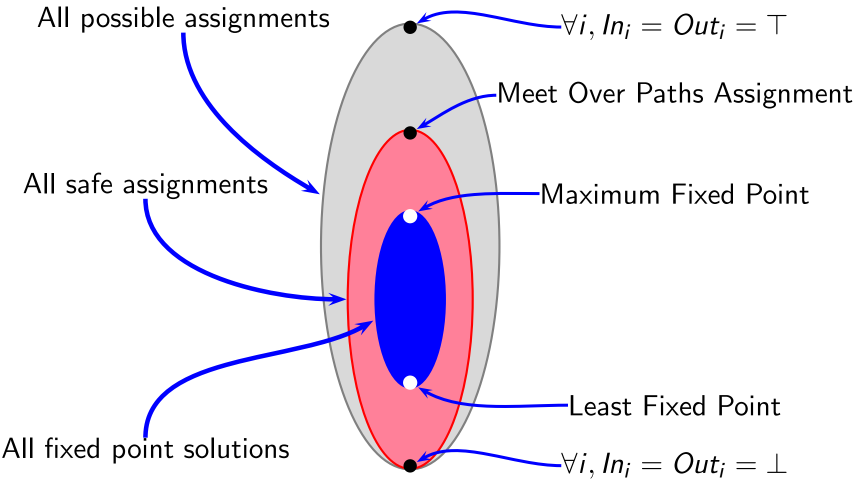
# Possible Assignments as Solutions of Data Flow Analyses



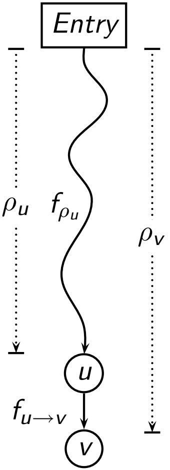
# Possible Assignments as Solutions of Data Flow Analyses



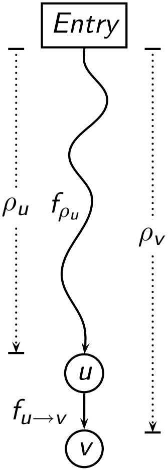
# Possible Assignments as Solutions of Data Flow Analyses



# Safety of MFP Solution: MFP $\sqsubseteq$ MoP

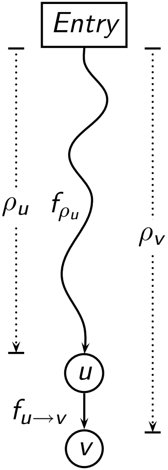


# Safety of MFP Solution: MFP $\sqsubseteq$ MoP



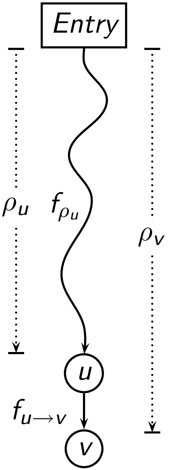

- $MoP(v) \;=\; \bigsqcap_{\rho \in Paths(v)} f_\rho(B = Boundary\_Info)$

# Safety of MFP Solution: MFP $\sqsubseteq$ MoP



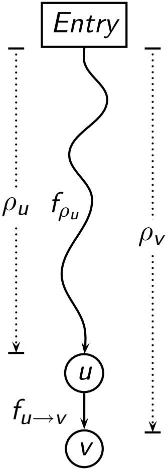

- $MoP(v) \quad = \bigsqcap_{\rho \in Paths(v)} f_\rho(B = Boundary\_Info)$
- Proof Obligation: $\forall \rho_v \; MFP(v) \sqsubseteq f_{\rho_v}(B)$

# Safety of MFP Solution: MFP $\sqsubseteq$ MoP




- $MoP(v) \quad = \displaystyle\bigsqcap_{\rho \in Paths(v)} f_\rho(B = Boundary\_Info)$
- Proof Obligation: $\forall \rho_v \; MFP(v) \sqsubseteq f_{\rho_v}(B)$
- Claim 1: $\forall u \rightarrow v, MFP(v) \sqsubseteq f_{u\rightarrow v}(MFP(u))$

# Safety of MFP Solution: MFP $\sqsubseteq$ MoP




- $MoP(v) \quad = \bigsqcap\limits_{\rho \in Paths(v)} f_\rho(B = Boundary\_Info)$
- Proof Obligation: $\forall \rho_v \; MFP(v) \sqsubseteq f_{\rho_v}(B)$
- Claim 1: $\forall u \rightarrow v, MFP(v) \sqsubseteq f_{u\rightarrow v}(MFP(u))$
- Proof Outline: Induction on path length

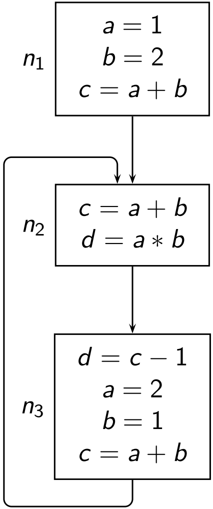
  Base case: Path of length 0.

  $MFP(Entry) = MoP(Entry) = B$

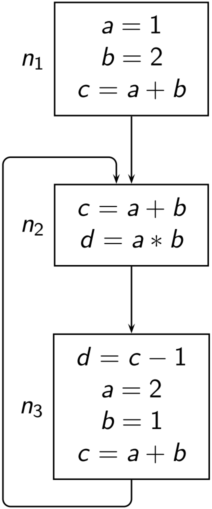
  Inductive hypothesis: Assume it holds for paths consisting of $k$ edges (say at $u$)

$$
\begin{array}{lll}
 & MFP(u) \sqsubseteq f_{\rho_u}(B) & \text{(Inductive hypothesis)} \\
 & MFP(v) \sqsubseteq f_{u\rightarrow v}(MFP(u)) & \text{(Claim 1)} \\
\Rightarrow & MFP(v) \sqsubseteq f_{u\rightarrow v}(f_{\rho_u}(B)) & \\
\Rightarrow & MFP(v) \sqsubseteq f_{\rho_v}(B) &
\end{array}
$$

# Assignments for Constant Propagation Example



# Assignments for Constant Propagation Example




MoP

$\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$

$\langle 1, 2, 3, \widehat{\top} \rangle$

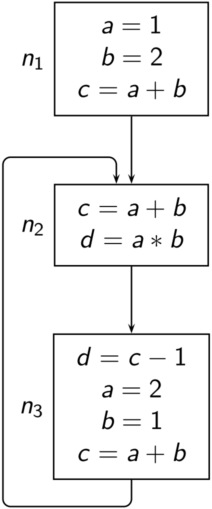
$\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$

$\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$

$\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$

$\langle 2, 1, 3, 2 \rangle$

# Assignments for Constant Propagation Example




| MoP | MFP |
|---|---|
| $\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$ | $\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$ |
| $\langle 1, 2, 3, \widehat{\top} \rangle$ | $\langle 1, 2, 3, \widehat{\top} \rangle$ |
| $\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$ | $\langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$ |
| $\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$ | $\langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$ |
| $\langle \widehat{\bot}, \widehat{\bot}, 3, 2 \rangle$ | $\langle \widehat{\bot}, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$ |
| $\langle 2, 1, 3, 2 \rangle$ | $\langle 2, 1, 3, \widehat{\bot} \rangle$ |

Part 7

# Performing Data Flow Analysis

# Performing Data Flow Analysis

- Algorithms for computing MFP solution
- Complexity of data flow analysis
- Factor affecting the complexity of data flow analysis

# Iterative Methods of Performing Data Flow Analysis

Successive recomputation after conservative initialization ($\top$)

- *Round Robin*. Repeated traversals over nodes in a fixed order

  Termination : After values stabilise

  + Simplest to understand and implement
  − May perform unnecessary computations

# Iterative Methods of Performing Data Flow Analysis

Successive recomputation after conservative initialization ($\top$)

- *Round Robin*. Repeated traversals over nodes in a fixed order

  Termination : After values stabilise

  + Simplest to understand and implement
  − May perform unnecessary computations

Our examples use this method.

# Iterative Methods of Performing Data Flow Analysis

Successive recomputation after conservative initialization ($\top$)

- *Round Robin*. Repeated traversals over nodes in a fixed order

  Termination : After values stabilise

  + Simplest to understand and implement
  − May perform unnecessary computations

Our examples use this method.

- *Work List*. Dynamic list of nodes which need recomputation

  Termination : When the list becomes empty

  + Demand driven. Avoid unnecessary computations.
  − Overheads of maintaining work list.

# Elimination Methods of Performing Data Flow Analysis

Delayed computations of dependent data flow values of dependent nodes.

Find suitable single-entry regions.

- *Interval Based Analysis*. Uses graph partitioning.
- $T_1$, $T_2$ *Based Analysis*. Uses graph parsing.

# Round Robin Iterative Algorithm for Computing MFP Assignment

$In_0 = \text{Boundary\_Info}$
**for** all $j \neq 0$, $In_j = \top$;
*change* $= $ *true*
**while** *change* **do**
$\{$ *change* $=$ *false*
  **for** $j = 1$ to $N - 1$ **do**
  $\{$ $temp = \displaystyle\prod_{p \in pred(j)} f_p(In_p)$
    **if** $temp \neq In_j$ **then**
    $\{$ $In_j = temp$
      *change* $=$ *true*
    $\}$
  $\}$
$\}$

# Complexity of Round Robin Iterative Algorithm

- Unidirectional bit vector frameworks
  - Construct a spaning tree $T$ of $G$ to identify postorder traversal
  - Traverse $G$ in reverse postorder for forward problems and Traverse $G$ in postorder for backward problems
  - Depth $d(G, T)$: Maximum number of back edges in any acyclic path

| Task | Number of iterations |
|---|---|
| Initialisation | 1 |
| Convergence (until *change* remains true) | $d(G, T)$ |
| Verifying convergence (*change* becomes false) | 1 |

# Complexity of Round Robin Iterative Algorithm

- Unidirectional bit vector frameworks
  - Construct a spaning tree $T$ of $G$ to identify postorder traversal
  - Traverse $G$ in reverse postorder for forward problems and Traverse $G$ in postorder for backward problems
  - Depth $d(G, T)$: Maximum number of back edges in any acyclic path

| Task | Number of iterations |
|---|---|
| Initialisation | 1 |
| Convergence<br>(until *change* remains true) | $d(G, T)$ |
| Verifying convergence<br>(*change* becomes false) | 1 |

- What about bidirectional bit vector frameworks?

# Complexity of Round Robin Iterative Algorithm

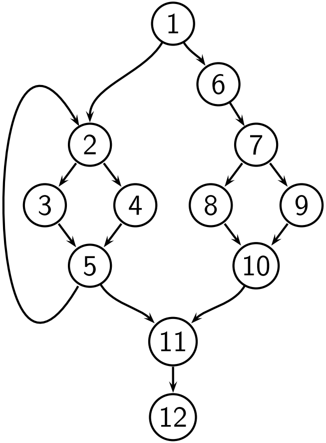
- Unidirectional bit vector frameworks
  - Construct a spaning tree $T$ of $G$ to identify postorder traversal
  - Traverse $G$ in reverse postorder for forward problems and Traverse $G$ in postorder for backward problems
  - Depth $d(G, T)$: Maximum number of back edges in any acyclic path

| Task | Number of iterations |
|---|---|
| Initialisation | 1 |
| Convergence (until *change* remains true) | $d(G, T)$ |
| Verifying convergence (*change* becomes false) | 1 |

- What about bidirectional bit vector frameworks?
- What about other frameworks?

# Complexity of Bidirectional Bit Vector Frameworks

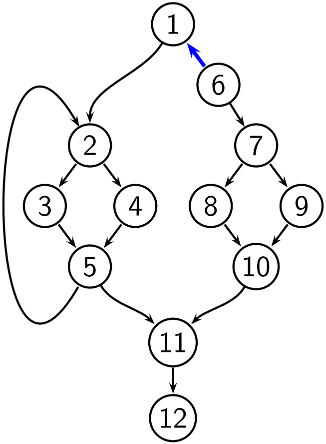
Example: Original formulation of PRE




- Information could flow along arbitrary paths

# Complexity of Bidirectional Bit Vector Frameworks

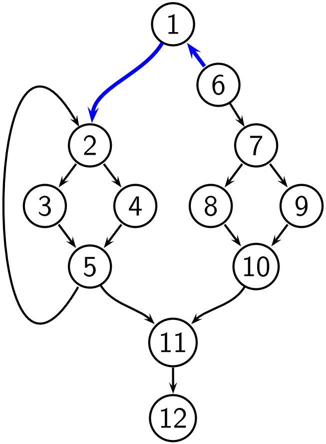
Example: Original formulation of PRE




- Information could flow along arbitrary paths

# Complexity of Bidirectional Bit Vector Frameworks

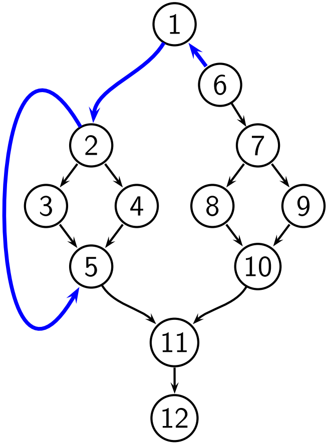
Example: Original formulation of PRE




- Information could flow along arbitrary paths

# Complexity of Bidirectional Bit Vector Frameworks

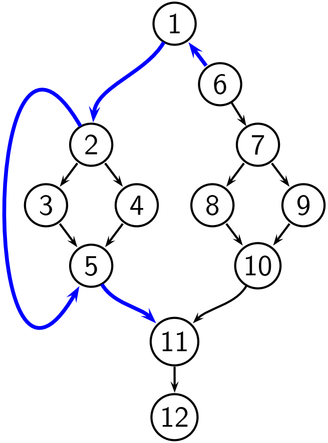
Example: Original formulation of PRE




- Information could flow along arbitrary paths

# Complexity of Bidirectional Bit Vector Frameworks

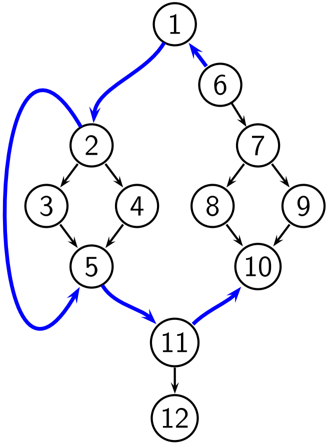
Example: Original formulation of PRE




- Information could flow along arbitrary paths

# Complexity of Bidirectional Bit Vector Frameworks

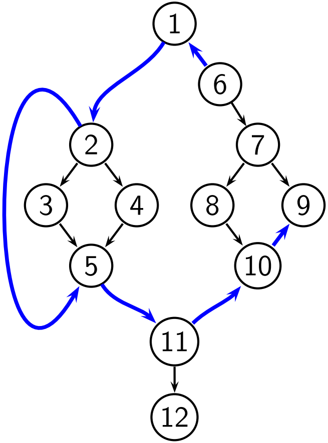
Example: Original formulation of PRE




- Information could flow along arbitrary paths

# Complexity of Bidirectional Bit Vector Frameworks

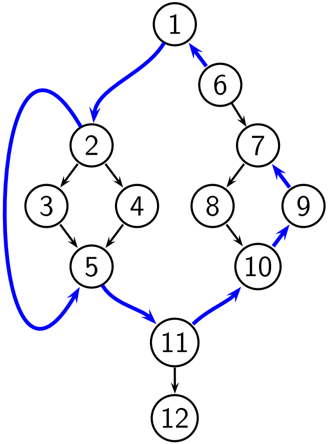
Example: Original formulation of PRE




- Information could flow along arbitrary paths

# Complexity of Bidirectional Bit Vector Frameworks

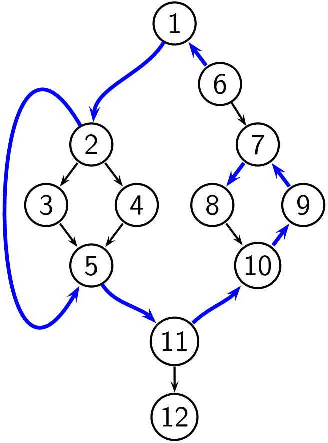
Example: Original formulation of PRE




- Information could flow along arbitrary paths

# Complexity of Bidirectional Bit Vector Frameworks

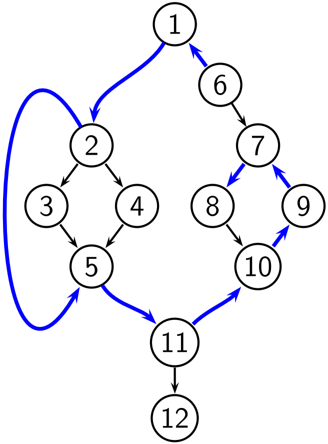
Example: Original formulation of PRE




- Information could flow along arbitrary paths

# Complexity of Bidirectional Bit Vector Frameworks

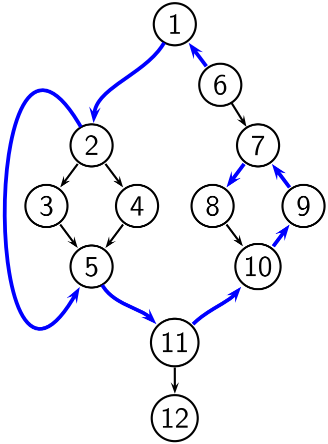
Example: Original formulation of PRE




- Information could flow along arbitrary paths
- Theoretically predicted number : 144

# Complexity of Bidirectional Bit Vector Frameworks

Example: Original formulation of PRE




- Information could flow along arbitrary paths
- Theoretically predicted number : 144
- Practical number : 5.

# Lacuna with PRE Complexity

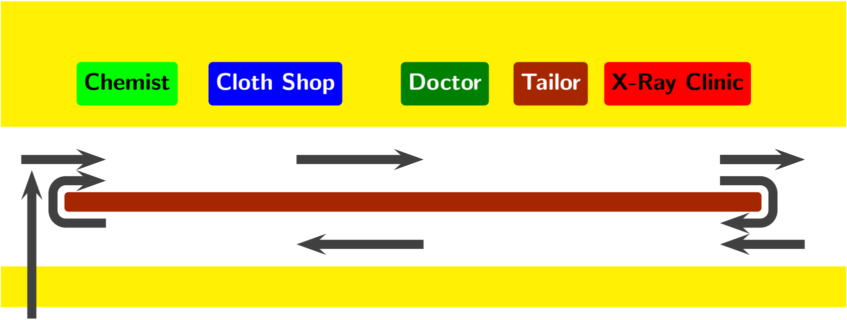
- Lacuna with PRE : Complexity $O(n^2)$ traversals.

  Practical graphs may have upto 50 nodes.

  - Predicted number of traversals : 2,500.
  - Practical number of traversals : $\leq$ 5.

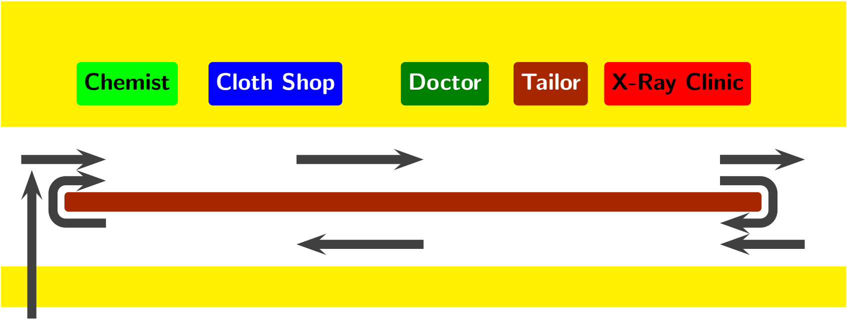
- No explanation for about 14 years despite dozens of efforts.
- Not much experimentation with performing advanced optimizations involving bidirectional dependency.

# Complexity of Round Robin Iterative Method



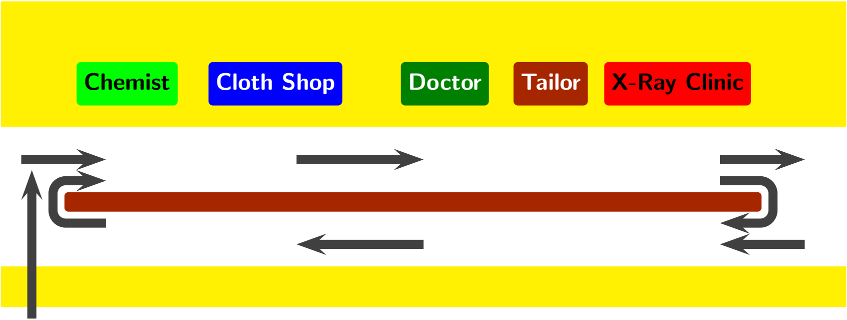

- Buy OTC (Over-The-Counter) medicine. No U-Turn 1 Trip

# Complexity of Round Robin Iterative Method



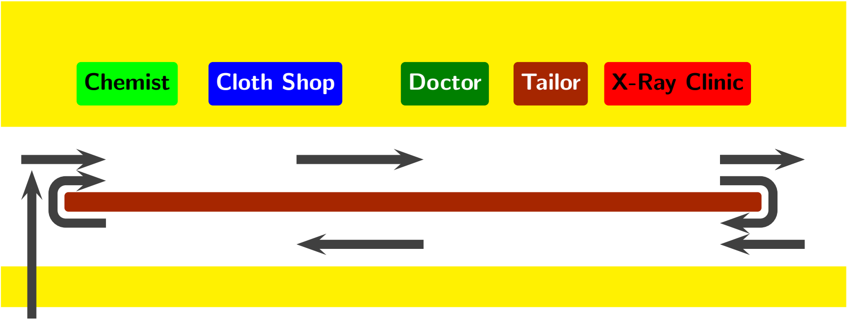

- Buy OTC (Over-The-Counter) medicine. No U-Turn 1 Trip
- Buy cloth. Give it to the tailor for stitching. No U-Turn 1 Trip

# Complexity of Round Robin Iterative Method



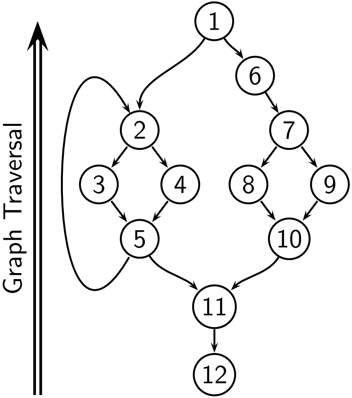

| | | |
|---|---|---|
| • Buy OTC (Over-The-Counter) medicine. | No U-Turn | 1 Trip |
| • Buy cloth. Give it to the tailor for stitching. | No U-Turn | 1 Trip |
| • Buy medicine with doctor's prescription. | 1 U-Turn | 2 Trips |

# Complexity of Round Robin Iterative Method



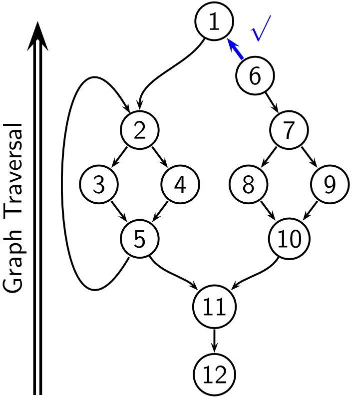

| | | |
|---|---|---|
| • Buy OTC (Over-The-Counter) medicine. | No U-Turn | 1 Trip |
| • Buy cloth. Give it to the tailor for stitching. | No U-Turn | 1 Trip |
| • Buy medicine with doctor's prescription. | 1 U-Turn | 2 Trips |
| • Buy medicine with doctor's prescription. The diagnosis requires X-Ray. | 2 U-Turns | 3 Trips |

# Complexity of Bidirectional Bit Vector Frameworks



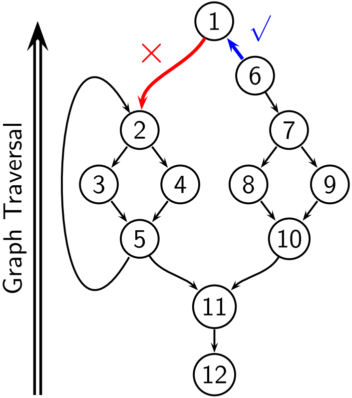

- Every "incompatible" edge traversal
  $\Rightarrow$ **One additional graph traversal**

# Complexity of Bidirectional Bit Vector Frameworks



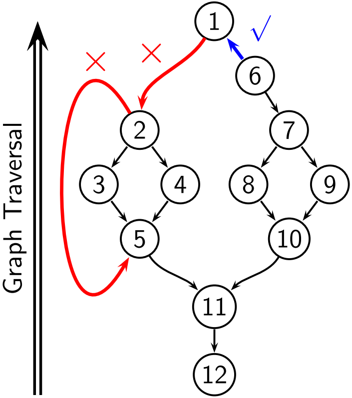

- Every "incompatible" edge traversal
  ⇒ **One additional graph traversal**
- Max. Incompatible edge traversals
  = *Width* of the graph = **0?**
- Maximum number of traversals =
  1 + Max. incompatible edge traversals

# Complexity of Bidirectional Bit Vector Frameworks




- $\boxed{\text{Every "incompatible" edge traversal } \Rightarrow \textbf{One additional graph traversal}}$
- Max. Incompatible edge traversals $=$ *Width* of the graph $=$ **1?**
- Maximum number of traversals $=$ $1 +$ Max. incompatible edge traversals

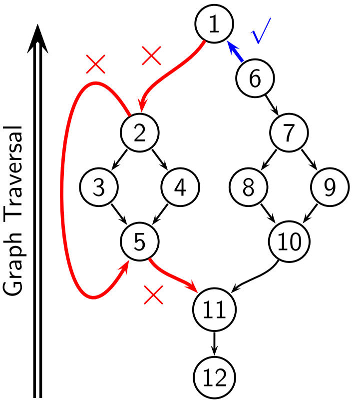
# Complexity of Bidirectional Bit Vector Frameworks




- | Every "incompatible" edge traversal<br>⇒ **One additional graph traversal** |
  |---|

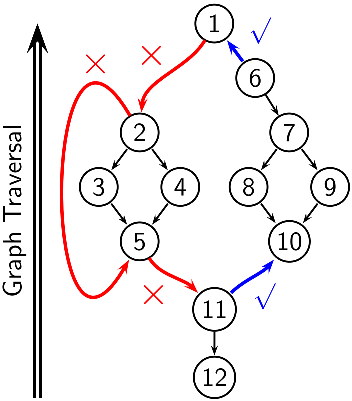
- Max. Incompatible edge traversals $=$ *Width* of the graph $=$ **2?**
- Maximum number of traversals $=$ $1 +$ Max. incompatible edge traversals

# Complexity of Bidirectional Bit Vector Frameworks



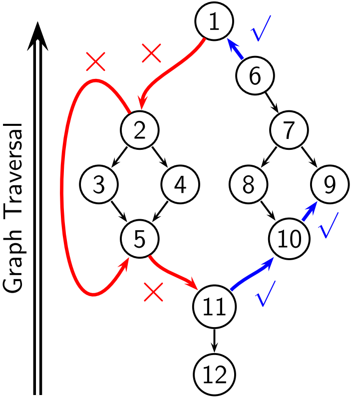

- **Every "incompatible" edge traversal $\Rightarrow$ One additional graph traversal**
- Max. Incompatible edge traversals = *Width* of the graph = **3?**
- Maximum number of traversals = 1 + Max. incompatible edge traversals

# Complexity of Bidirectional Bit Vector Frameworks



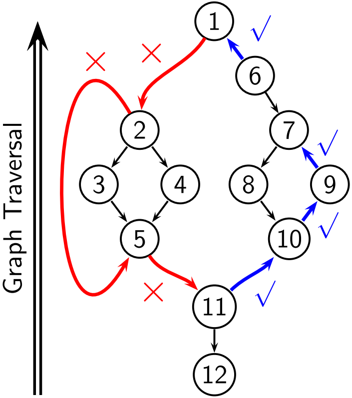

- Every "incompatible" edge traversal
  $\Rightarrow$ **One additional graph traversal**
- Max. Incompatible edge traversals
  $=$ *Width* of the graph $=$ **3?**
- Maximum number of traversals $=$
  $1 +$ Max. incompatible edge traversals

# Complexity of Bidirectional Bit Vector Frameworks



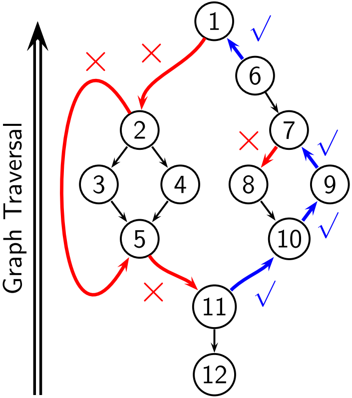

- Every "incompatible" edge traversal
  $\Rightarrow$ **One additional graph traversal**
- Max. Incompatible edge traversals
  $=$ *Width* of the graph $=$ **3?**
- Maximum number of traversals $=$
  $1 +$ Max. incompatible edge traversals

# Complexity of Bidirectional Bit Vector Frameworks



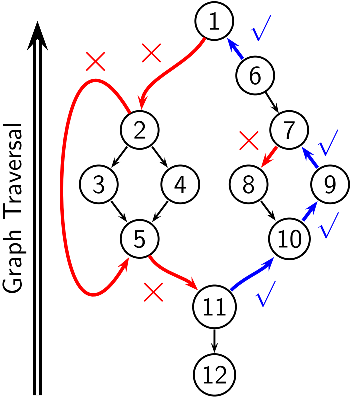

- Every "incompatible" edge traversal
  ⇒ **One additional graph traversal**
- Max. Incompatible edge traversals
  = *Width* of the graph = **3?**
- Maximum number of traversals =
  1 + Max. incompatible edge traversals

# Complexity of Bidirectional Bit Vector Frameworks




- **Every "incompatible" edge traversal**
  **⇒ One additional graph traversal**
- Max. Incompatible edge traversals
  = *Width* of the graph = **4**
- Maximum number of traversals =
  1 + Max. incompatible edge traversals

# Complexity of Bidirectional Bit Vector Frameworks




- Every "incompatible" edge traversal
  ⇒ **One additional graph traversal**
- Max. Incompatible edge traversals
  = *Width* of the graph = **4**
- Maximum number of traversals =
  1 + 4 = 5

# Framework Properties Influencing Complexity

Depends on the loop closure properties of the framework

$k$-Bounded Frameworks

$$f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap \ldots \sqcap f^{k-1}(x)$$

Necessary and sufficient

# Framework Properties Influencing Complexity

Depends on the loop closure properties of the framework

$k$-Bounded Frameworks

Fast Frameworks ($k = 2$)

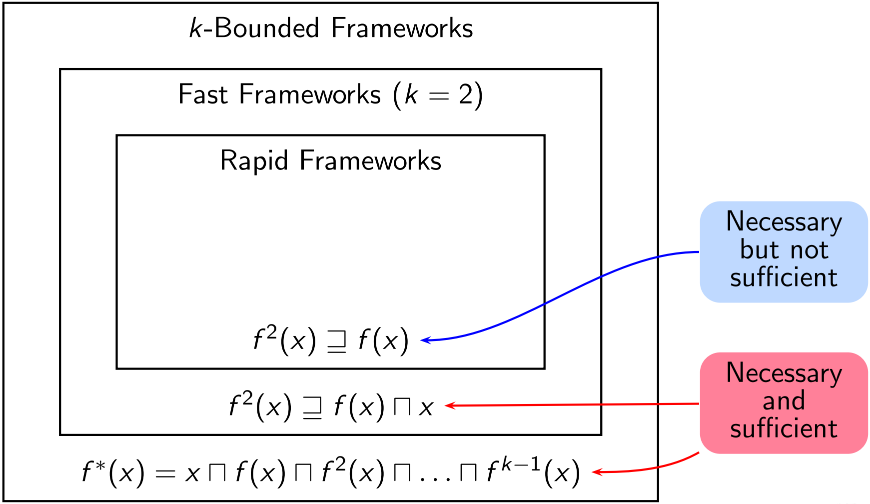
$f^2(x) \sqsupseteq f(x) \sqcap x$

$f^*(x) = x \sqcap f(x) \sqcap f^2(x) \sqcap \ldots \sqcap f^{k-1}(x)$

Necessary and sufficient

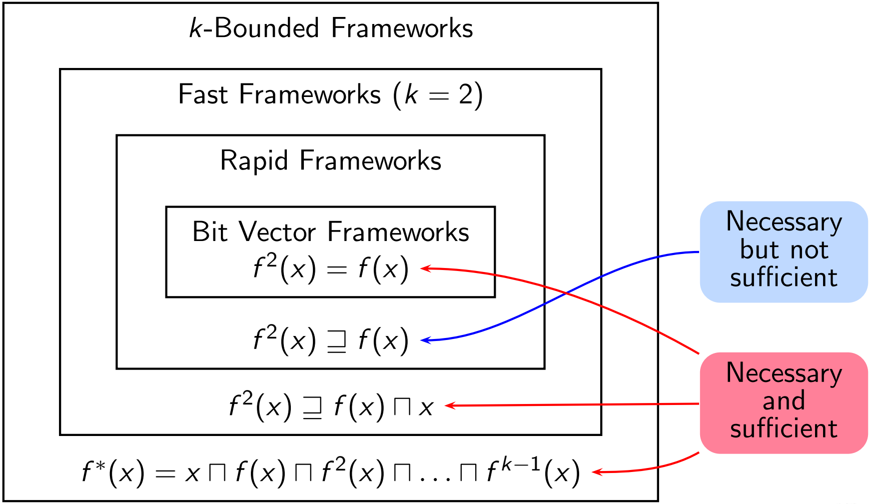
# Framework Properties Influencing Complexity

Depends on the loop closure properties of the framework



# Framework Properties Influencing Complexity

Depends on the loop closure properties of the framework



# Complexity of Round Robin Iterative Algorithm

- Unidirectional rapid frameworks

| Task | Number of iterations | |
|---|---|---|
| | Irreducible $G$ | Reducible $G$ |
| Initialisation | 1 | 1 |
| Convergence (until *change* remains true) | $d(G,T)+1$ | $d(G,T)$ |
| Verifying convergence (*change* becomes false) | 1 | 1 |

# Loop Closures in Bit Vector Frameworks

- Flow functions in bit vector frameworks have constant Gen and Kill

$$
\begin{aligned}
f^*(x) &= x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap \ldots \\
f^2(x) &= f\,(\mathsf{Gen} \cup (x - \mathsf{Kill})) \\
&= \mathsf{Gen} \cup ((\mathsf{Gen} \cup (x - \mathsf{Kill})) - \mathsf{Kill}) \\
&= \mathsf{Gen} \cup ((\mathsf{Gen} - \mathsf{Kill}) \cup (x - \mathsf{Kill})) \\
&= \mathsf{Gen} \cup (\mathsf{Gen} - \mathsf{Kill}) \cup (x - \mathsf{Kill}) \\
&= \mathsf{Gen} \cup (x - \mathsf{Kill}) \quad = \quad f(x) \\
f^*(x) &= x \sqcap f(x)
\end{aligned}
$$

# Loop Closures in Bit Vector Frameworks

- Flow functions in bit vector frameworks have constant Gen and Kill

$$
\begin{aligned}
f^*(x) &= x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap \ldots \\
f^2(x) &= f\,(\text{Gen} \cup (x - \text{Kill})) \\
&= \text{Gen} \cup ((\text{Gen} \cup (x - \text{Kill})) - \text{Kill}) \\
&= \text{Gen} \cup ((\text{Gen} - \text{Kill}) \cup (x - \text{Kill})) \\
&= \text{Gen} \cup (\text{Gen} - \text{Kill}) \cup (x - \text{Kill}) \\
&= \text{Gen} \cup (x - \text{Kill}) \;\; = \;\; f(x) \\
f^*(x) &= x \sqcap f(x)
\end{aligned}
$$

- *Loop Closures of Bit Vector Frameworks are 2-bounded.*

# Loop Closures in Bit Vector Frameworks

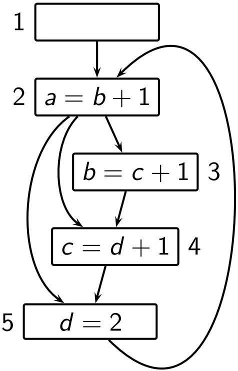
- Flow functions in bit vector frameworks have constant Gen and Kill

$$
\begin{aligned}
f^*(x) &= x \sqcap f(x) \sqcap f^2(x) \sqcap f^3(x) \sqcap \ldots \\
f^2(x) &= f\,(\text{Gen} \cup (x - \text{Kill})) \\
&= \text{Gen} \cup ((\text{Gen} \cup (x - \text{Kill})) - \text{Kill}) \\
&= \text{Gen} \cup ((\text{Gen} - \text{Kill}) \cup (x - \text{Kill})) \\
&= \text{Gen} \cup (\text{Gen} - \text{Kill}) \cup (x - \text{Kill}) \\
&= \text{Gen} \cup (x - \text{Kill}) \quad = \quad f(x) \\
f^*(x) &= x \sqcap f(x)
\end{aligned}
$$

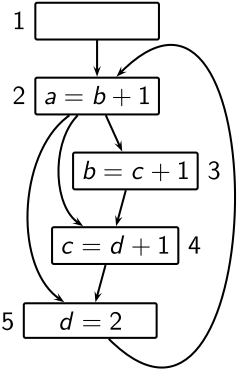
- *Loop Closures of Bit Vector Frameworks are 2-bounded.*
- Intuition: Since Gen and Kill are constant, same things are generated or killed in every application of $f$.

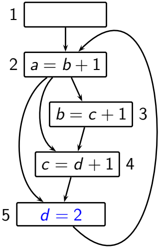
  Multiple applications of $f$ are not required unless the input value changes.

# What About Constant Propagation?



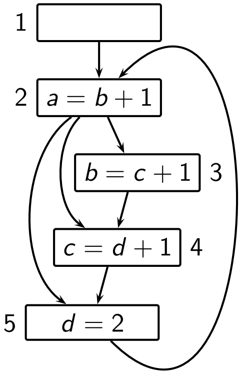
# What About Constant Propagation?

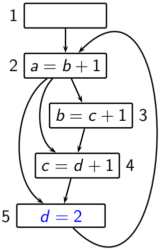


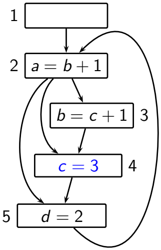

Iteration $\#1$

# What About Constant Propagation?



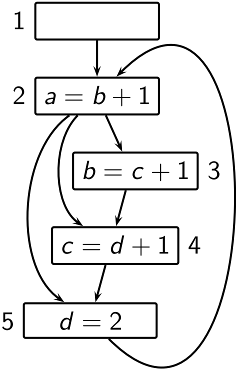



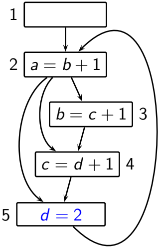

Iteration #1



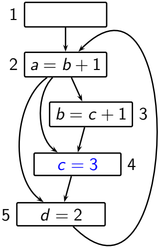

Iteration #2

# What About Constant Propagation?

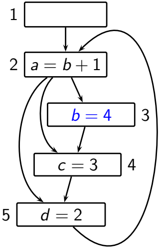



Iteration #1

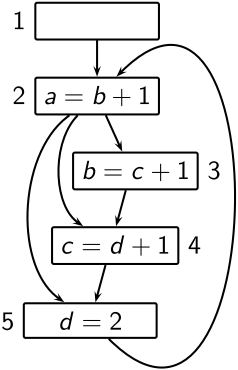


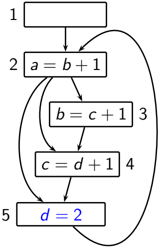

Iteration #2



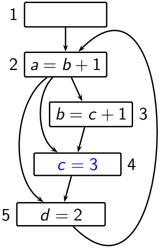

Iteration #3

# What About Constant Propagation?

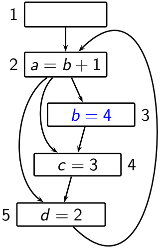


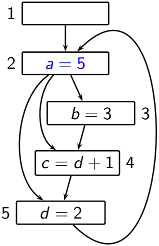

Iteration #1




Iteration #2




Iteration #3




Iteration #4

# Larger Values of Loop Closure Bounds

- Fast Frameworks (eg. bit vector frameworks)
  Data flow values of different entities are independent
  (Entities refer to expressions, variables etc.)
- Non-fast frameworks
  Data flow values of different entities are inter-dependent
  Loop closure bound depends on the number of entities

# Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

# Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

| Separable | Non-Separable |
|---|---|

# Separability

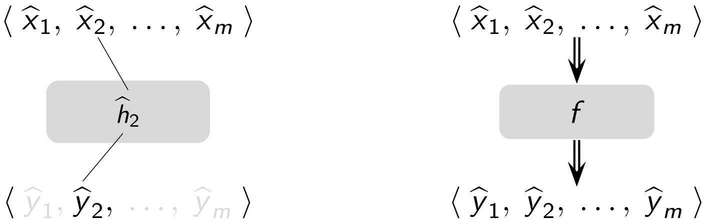
$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

| **Separable** | **Non-Separable** |
|---|---|
| $\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$ | $\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$ |
| $\Downarrow$ | $\Downarrow$ |
| $f$ | $f$ |
| $\Downarrow$ | $\Downarrow$ |
| $\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$ | $\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$ |

# Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

**Separable** | **Non-Separable**



# Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

**Separable**

$$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$$

$$\widehat{h}_2$$

$$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$$

$\widehat{h} : \widehat{L} \mapsto \widehat{L}$

**Non-Separable**

$$\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$$

$$\Downarrow$$

$$f$$

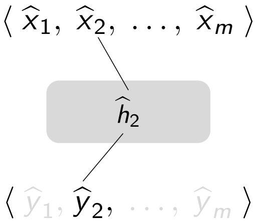
$$\Downarrow$$

$$\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$$

# Separability

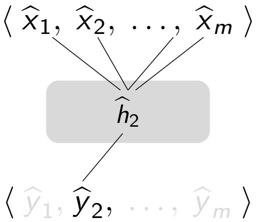
$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

**Separable**




$\widehat{h} : \widehat{L} \mapsto \widehat{L}$

**Non-Separable**



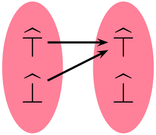
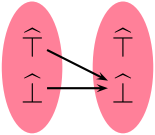
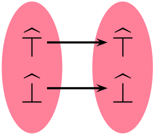
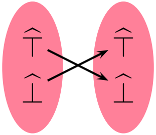
# Separability

$f : L \mapsto L$ is $\langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_m \rangle$ where $\widehat{h}_i$ computes the value of $\widehat{x}_i$

| Separable | Non-Separable |
|---|---|
| $\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$ | $\langle\ \widehat{x}_1,\ \widehat{x}_2,\ \ldots,\ \widehat{x}_m\ \rangle$ |
| $\widehat{h}_2$ | $\widehat{h}_2$ |
| $\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$ | $\langle\ \widehat{y}_1,\ \widehat{y}_2,\ \ldots,\ \widehat{y}_m\ \rangle$ |
| $\widehat{h} : \widehat{L} \mapsto \widehat{L}$ | $\widehat{h} : L \mapsto \widehat{L}$ |
| Example: All bit vector frameworks | Example: Constant Propagation |

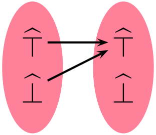
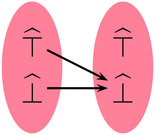
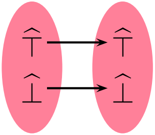
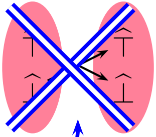
# Separability of Bit Vector Frameworks

- $\widehat{L}$ is $\{0,1\}$, $L$ is $\{0,1\}^m$
- $\widehat{\sqcap}$ is either boolean AND or boolean OR
- $\widehat{\top}$ and $\widehat{\bot}$ are 0 or 1 depending on $\widehat{\sqcap}$.
- $\widehat{h}$ is a *bit function* and could be one of the following:

| *Raise* | *Lower* | *Propagate* | *Negate* |
|---|---|---|---|
|  |  |  |  |

# Separability of Bit Vector Frameworks

- $\widehat{L}$ is $\{0,1\}$, $L$ is $\{0,1\}^m$
- $\widehat{\sqcap}$ is either boolean AND or boolean OR
- $\widehat{\top}$ and $\widehat{\bot}$ are 0 or 1 depending on $\widehat{\sqcap}$.
- $\widehat{h}$ is a *bit function* and could be one of the following:

| *Raise* | *Lower* | *Propagate* | *Negate* |
|---|---|---|---|
|  |  |  |  |

Non-monotonicity

# Data Flow Equations for Non-Separable Flows

- General flow functions can be written as

$$f_n(X) \quad = \quad (X - \mathsf{Kill}_n(X)) \cup \mathsf{Gen}_n(X)$$

where Gen and Kill have constant and dependent parts

$$\begin{aligned} \mathsf{Gen}_n(X) &= ConstGen_n \cup DepGen_n(X) \\ \mathsf{Kill}_n(X) &= ConstKill_n \cup DepKill_n(X) \end{aligned}$$

# Data Flow Equations for Non-Separable Flows

- General flow functions can be written as

$$f_n(X) \quad = \quad (X - \mathsf{Kill}_n(X)) \cup \mathsf{Gen}_n(X)$$

where Gen and Kill have constant and dependent parts

$$\begin{aligned} \mathsf{Gen}_n(X) &= ConstGen_n \cup DepGen_n(X) \\ \mathsf{Kill}_n(X) &= ConstKill_n \cup DepKill_n(X) \end{aligned}$$

- Bit vector frameworks are a special case

$$DepGen_n(X) = DepKill_n(X) = \emptyset$$

*Part 8*

# *Heap Reference Analysis*

# An Overview

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

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

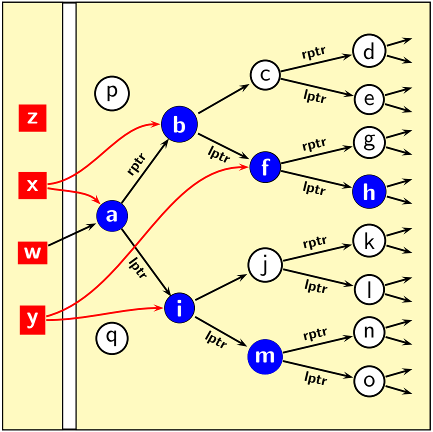
- A link may be accessed in multiple ways
- Setting links to NULL
  - *Alias Analysis*. Identify all possible ways of accessing a link
  - *Liveness Analysis*. For each program point, identify "dead" links (i.e. links which are not accessed after that program point)
  - *Availability and Anticipability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

# Assumptions

For simplicity of exposition

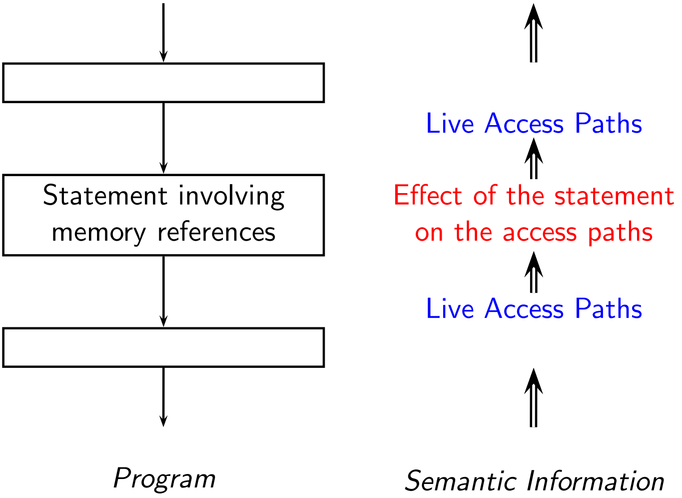
- Java model of heap access
  - Root variables are on stack and represent references to memory in heap.
  - Root variables cannot be pointed to by any reference.
- Simple extensions for C++
  - Root variables can be pointed to by other pointers.
  - Pointer arithmetic is not handled.

# Key Idea #1 : Access Paths Denote Links

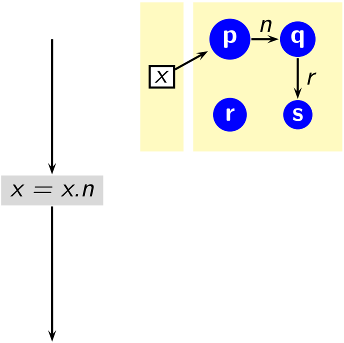



- Root variables : $x, y, z$
- Field names : rptr, lptr
- Access path : $x$➔rptr➔lptr
  Semantically, sequence of "links"
- Frontier : name of the last link
- Live access path : Iff the link corresponding to its frontier is used in future

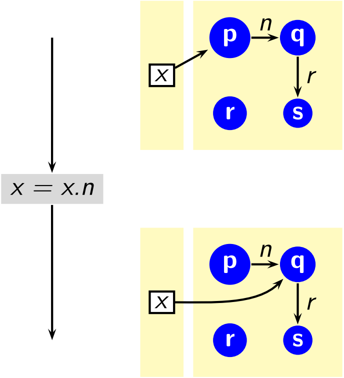
# Liveness Analysis



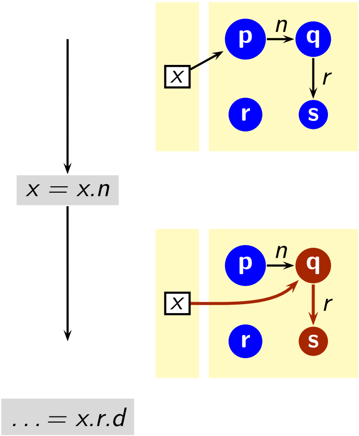
# Key Idea #2 : Transfer of Access Paths



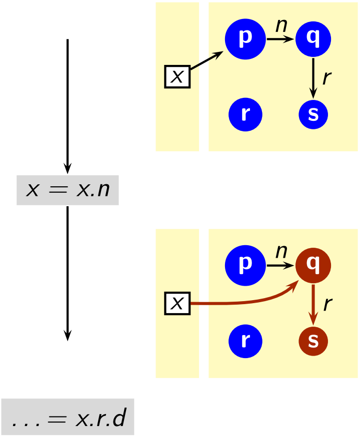
# Key Idea #2 : Transfer of Access Paths



# Key Idea #2 : Transfer of Access Paths

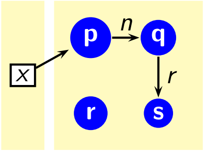


# Key Idea #2 : Transfer of Access Paths

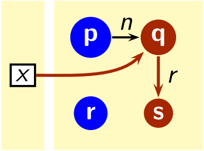



$\{x, x \rightarrow r\}$

# Key Idea #2 : Transfer of Access Paths




$x = x.n$

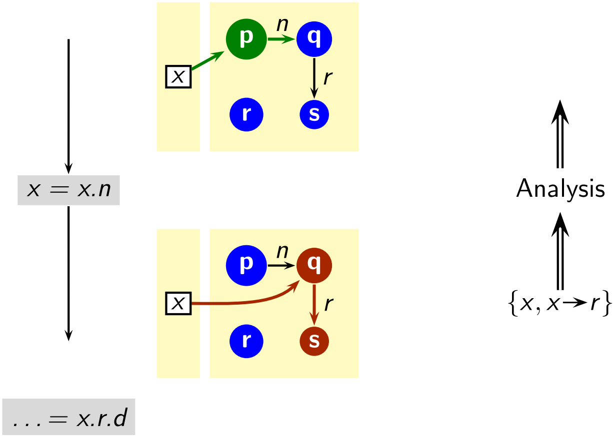



$\ldots = x.r.d$

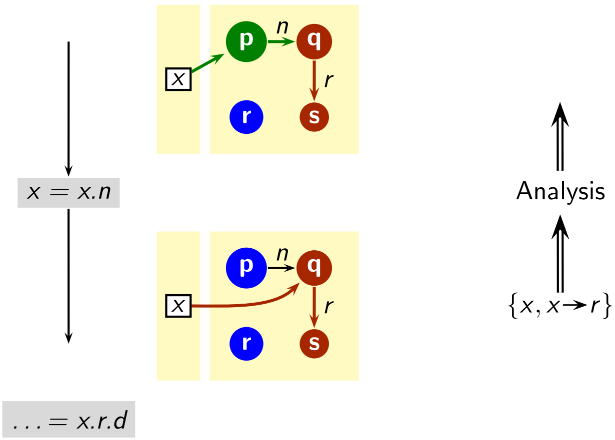
Analysis

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

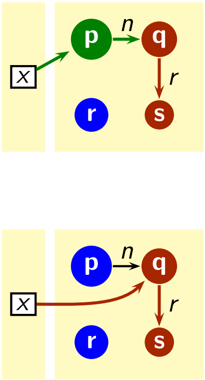
# Key Idea #2 : Transfer of Access Paths



# Key Idea #2 : Transfer of Access Paths



# Key Idea #2 : Transfer of Access Paths




$x = x.n$

$\ldots = x.r.d$

| | |
|---|---|
| **Generated** | |
| Constant | $\{x, x \rightarrow n\}$ |
| Dependent | $\{x \rightarrow n \rightarrow r\}$ |
| **Killed** | |
| Constant | $\{x, x \rightarrow r\}$ |
| Dependent | $\emptyset$ |

$\Uparrow$

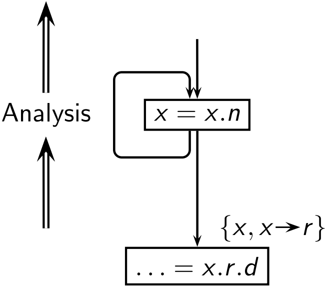
Analysis

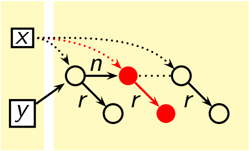
$\Uparrow$

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

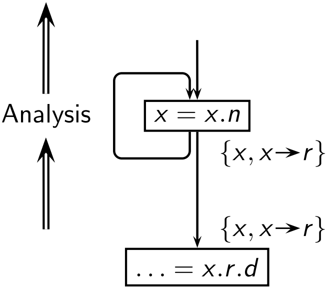
$x$ after the assinment is same as the $x \rightarrow n$ before the assignment

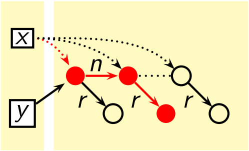
# Access Paths as Data Flow Values

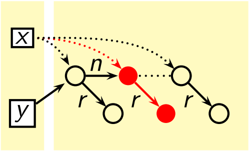


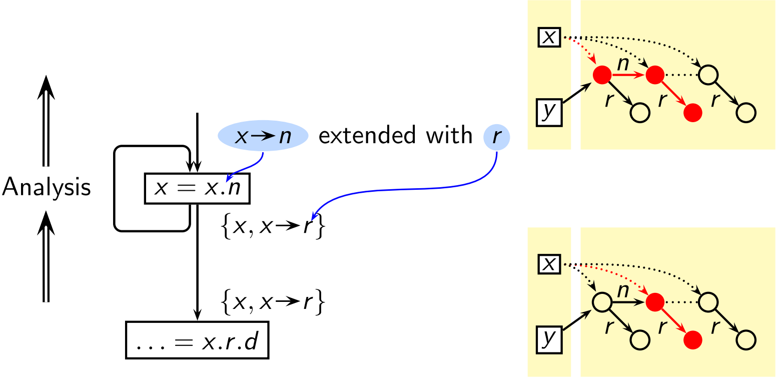
# Access Paths as Data Flow Values



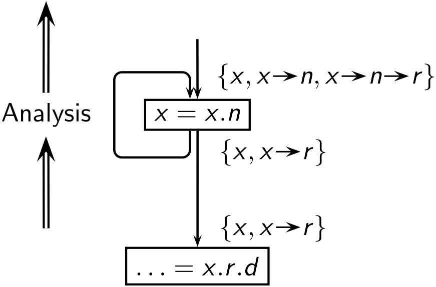



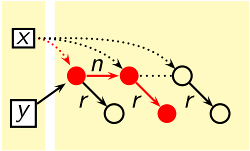



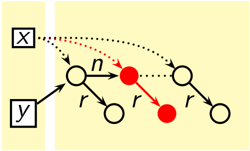
# Access Paths as Data Flow Values



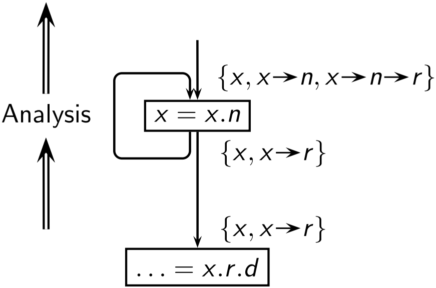
# Access Paths as Data Flow Values

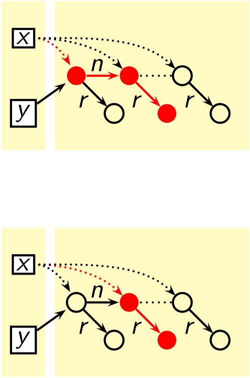


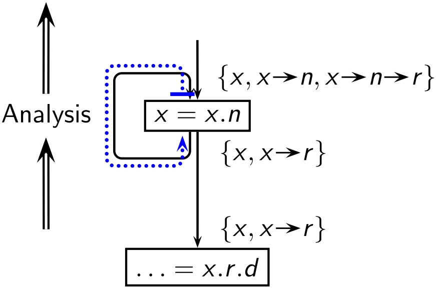
# Access Paths as Data Flow Values



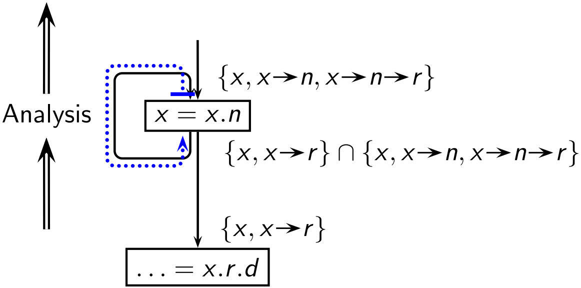
# Access Paths as Data Flow Values

Anticipability of Heap References: An *All Paths* problem



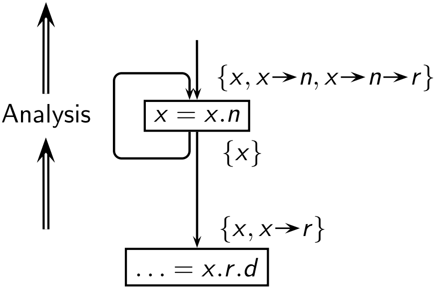
# Access Paths as Data Flow Values

Anticipability of Heap References: An *All Paths* problem



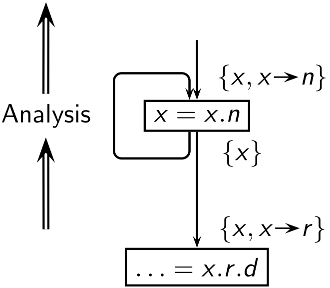
# Access Paths as Data Flow Values

Anticipability of Heap References: An *All Paths* problem



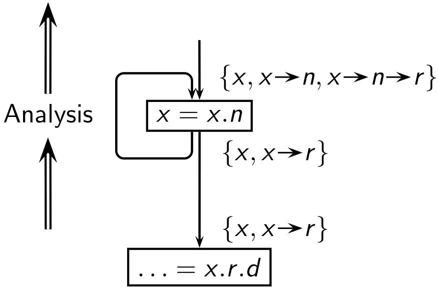
# Access Paths as Data Flow Values

Anticipability of Heap References: An *All Paths* problem



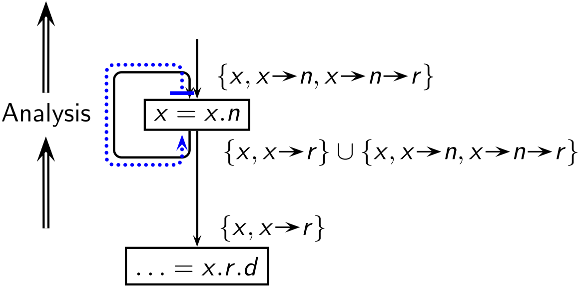
# Access Paths as Data Flow Values

Liveness of Heap References: An *Any Path* problem



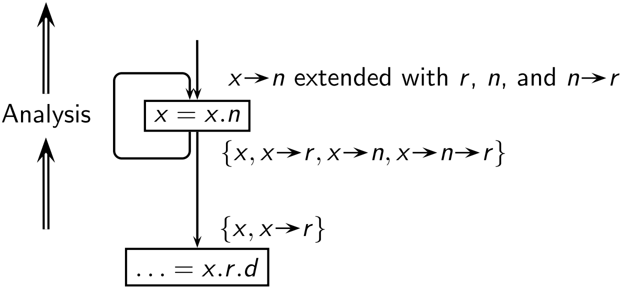
# Access Paths as Data Flow Values

Liveness of Heap References: An *Any Path* problem



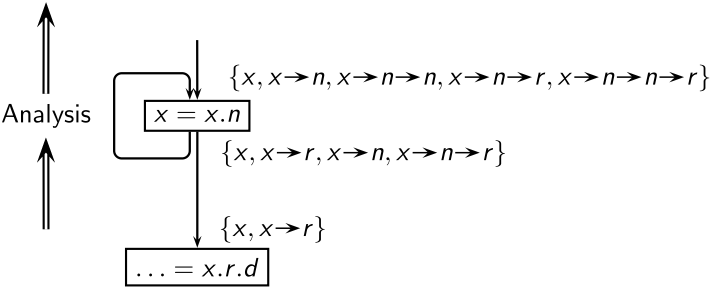
# Access Paths as Data Flow Values

Liveness of Heap References: An *Any Path* problem



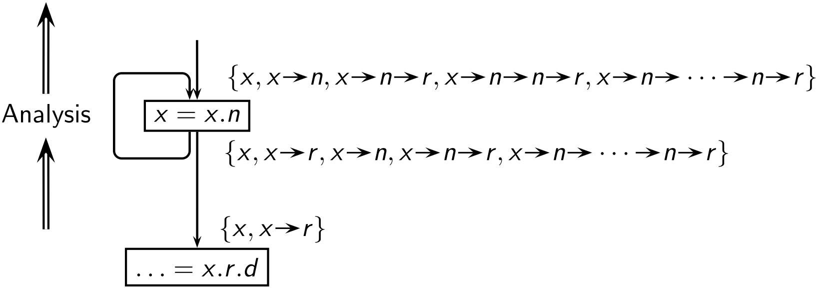
# Access Paths as Data Flow Values

Liveness of Heap References: An *Any Path* problem



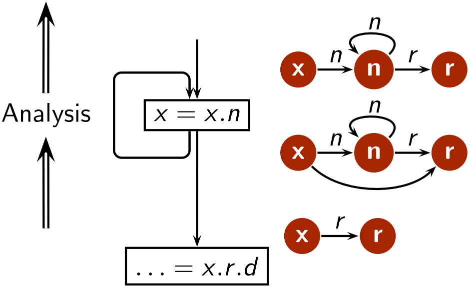
# Access Paths as Data Flow Values

Liveness of Heap References: An *Any Path* problem



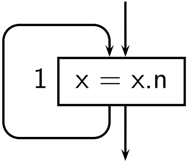

*Infinite Number of Unbounded Access Paths*

# Key Idea #3: Using Graphs as Data Flow Values



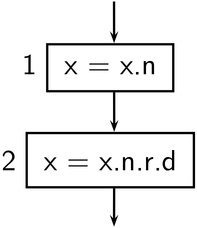

*Finite Number of Bounded Structures*

# Key Idea #4 : Include Program Point in Graphs




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

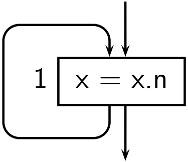
*Different occurrences of n's in an access path are*
**Indistinguishable**




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

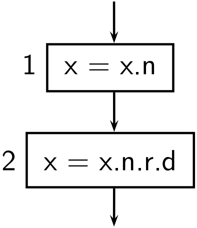
*Different occurrences of n's in an access path are*
**Distinct**

# Key Idea #4 : Include Program Point in Graphs



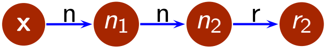

$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow n, \ldots\}$

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

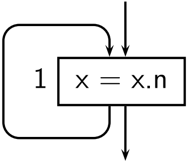



$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow r\}$

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

Access Graph : 

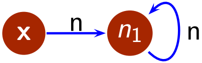
# Key Idea #4 : Include Program Point in Graphs

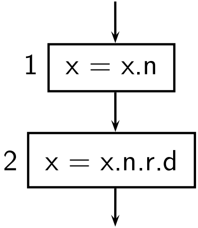



$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow n, \ldots\}$

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

Access Graph :

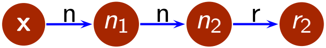


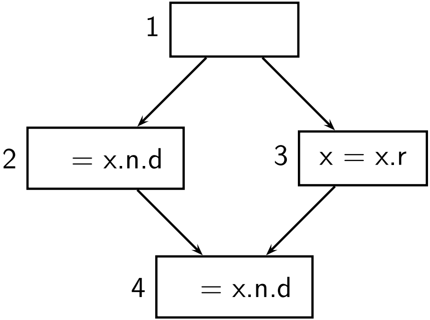

$\{x, x\rightarrow n, x\rightarrow n\rightarrow n, x\rightarrow n\rightarrow n\rightarrow r\}$

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

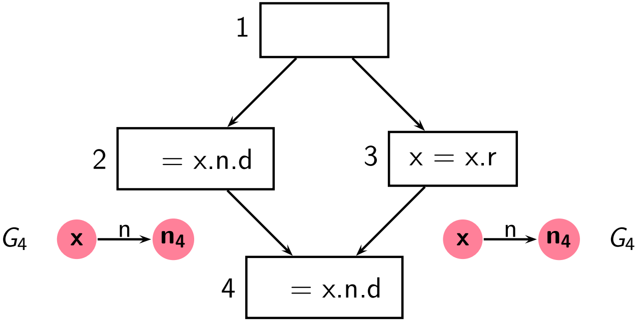
Access Graph :



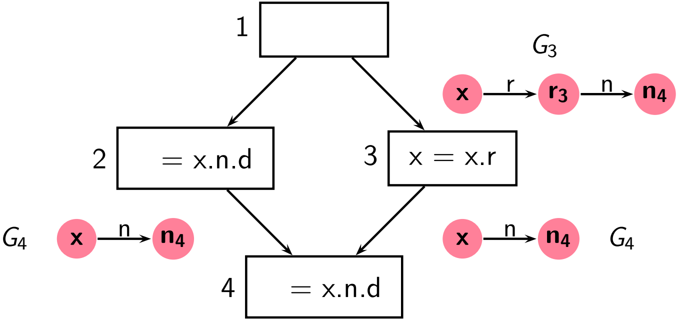
# Inclusion of Program Point Facilitates Summarization



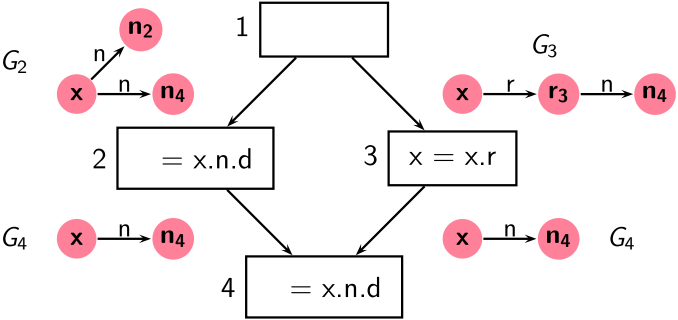
# Inclusion of Program Point Facilitates Summarization



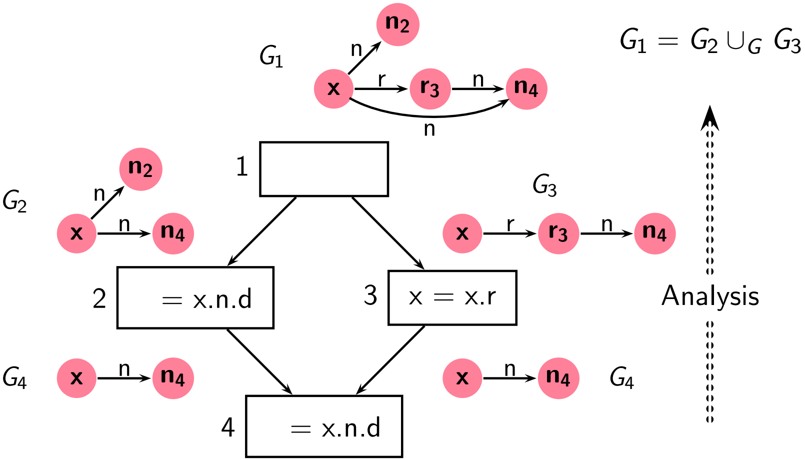
# Inclusion of Program Point Facilitates Summarization



# Inclusion of Program Point Facilitates Summarization

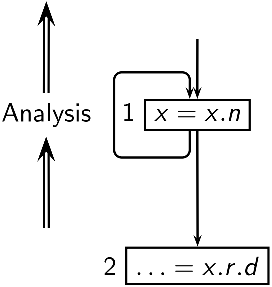


# Inclusion of Program Point Facilitates Summarization

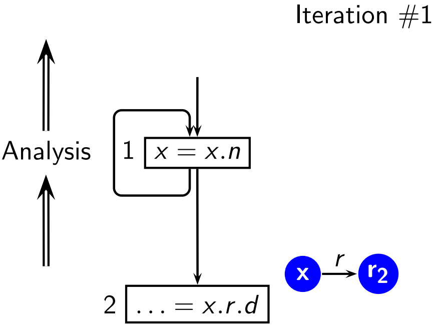


# Inclusion of Program Point Facilitates Summarization

Iteration #1

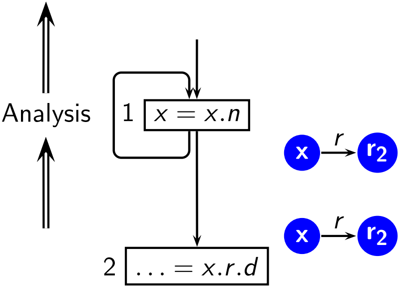


# Inclusion of Program Point Facilitates Summarization

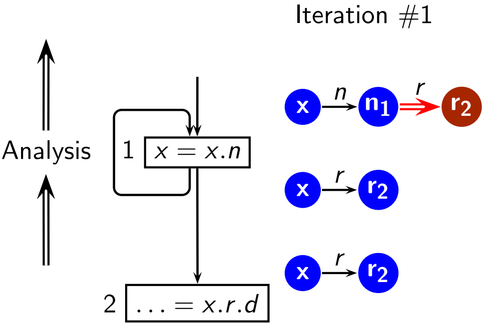


# Inclusion of Program Point Facilitates Summarization

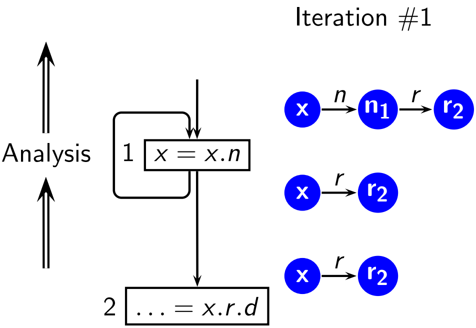
Iteration #1



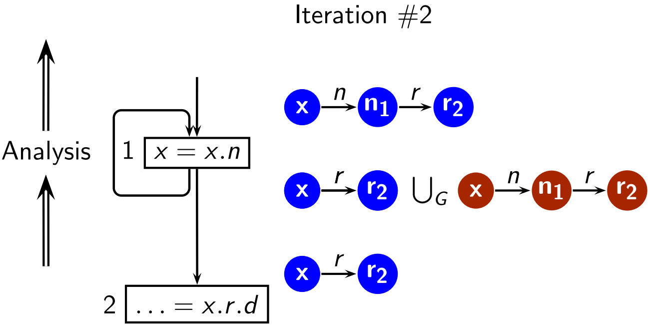
# Inclusion of Program Point Facilitates Summarization



# Inclusion of Program Point Facilitates Summarization

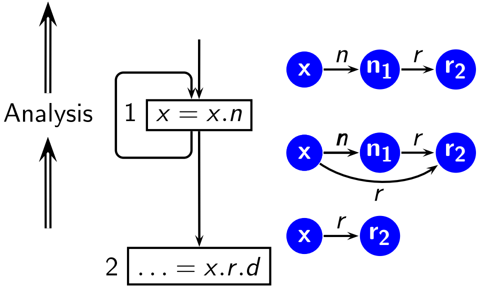


# Inclusion of Program Point Facilitates Summarization

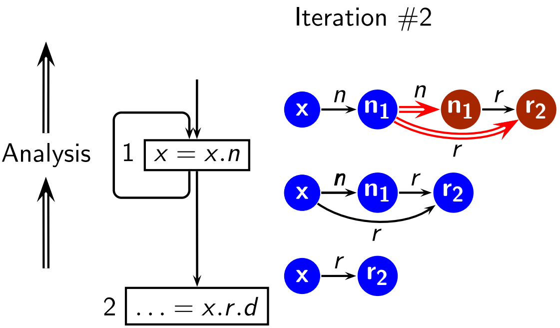


# Inclusion of Program Point Facilitates Summarization

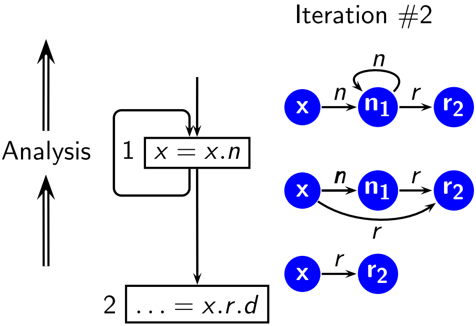
Iteration #2



# Inclusion of Program Point Facilitates Summarization

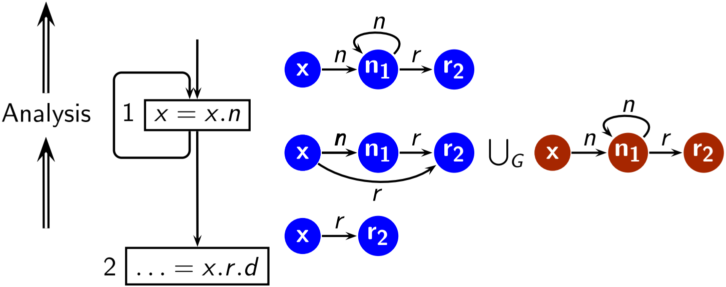


# Inclusion of Program Point Facilitates Summarization

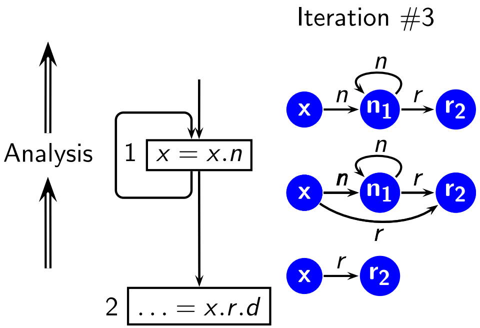


# Inclusion of Program Point Facilitates Summarization

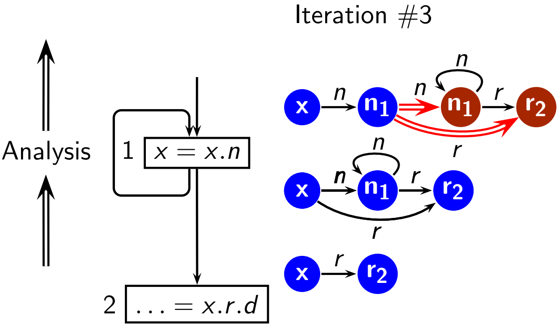
Iteration #3



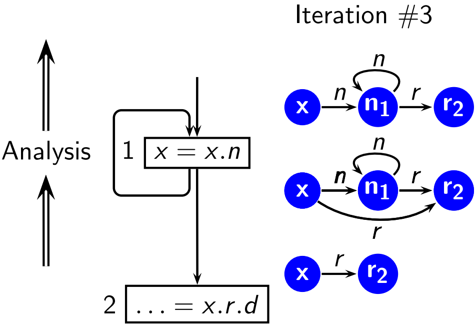
# Inclusion of Program Point Facilitates Summarization



# Inclusion of Program Point Facilitates Summarization



# Inclusion of Program Point Facilitates Summarization



# Lattice of Access Graphs

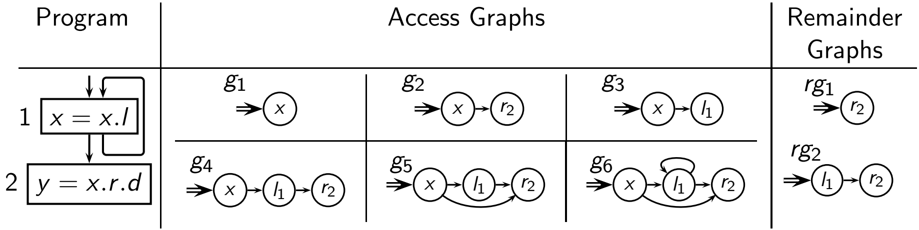
- Finite number of nodes in an access graph for a variable
- $\cup_G$ induces a partial order on access graphs
  - $\Rightarrow$ a finite (and hence complete) lattice
  - $\Rightarrow$ All standard results of classical data flow analysis can be extended to this analysis.

    *Termination and boundedness, convergence on MFP, complexity etc.*

# Access Graph Operations

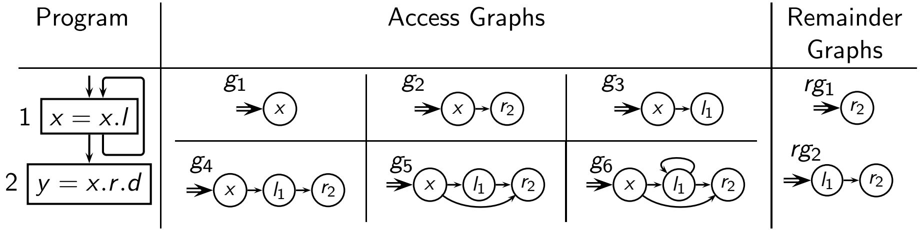
- *Union.* $G \cup_G G'$
- *Path Removal.*
  $G \ominus \rho$ removes those access paths in $G$ which have $\rho$ as a prefix.
- *Factorization* $(/)$.
- *Extension.*

# Access Graph Operations: Examples



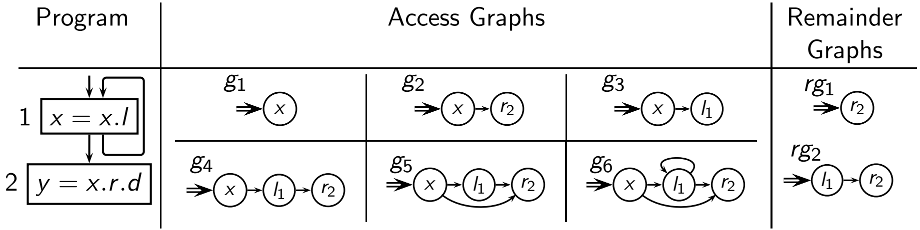

| Union | Path Removal | Factorisation | Extension |
|---|---|---|---|
| | | | |

# Access Graph Operations: Examples



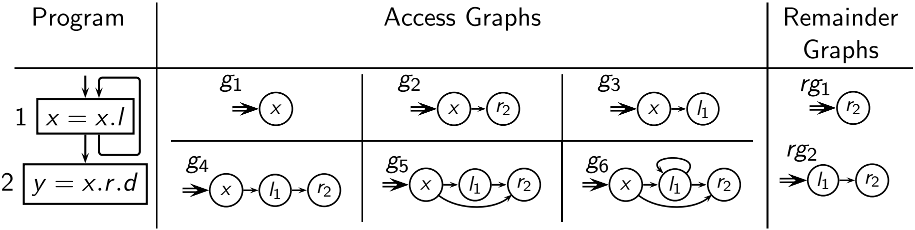

| Union | Path Removal | Factorisation | Extension |
|---|---|---|---|
| $g_3 \cup_G g_4 = g_4$<br>$g_2 \cup_G g_4 = g_5$<br>$g_5 \cup_G g_4 = g_5$<br>$g_5 \cup_G g_6 = g_6$ | | | |

# Access Graph Operations: Examples



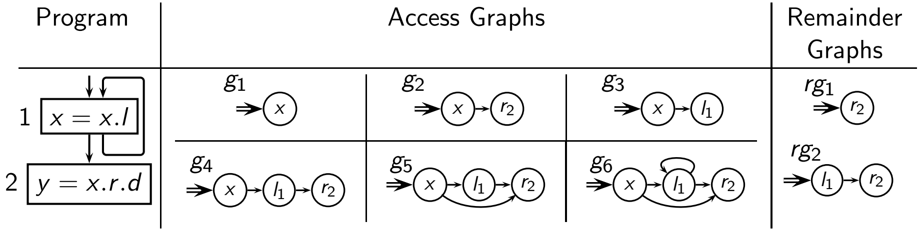

| Union | Path Removal | Factorisation | Extension |
|---|---|---|---|
| $g_3 \cup_G g_4 = g_4$ | $g_6 \ominus x \rightarrow l = g_2$ | | |
| $g_2 \cup_G g_4 = g_5$ | $g_5 \ominus x = \mathcal{E}_G$ | | |
| $g_5 \cup_G g_4 = g_5$ | $g_4 \ominus x \rightarrow r = g_4$ | | |
| $g_5 \cup_G g_6 = g_6$ |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

# Access Graph Operations: Examples



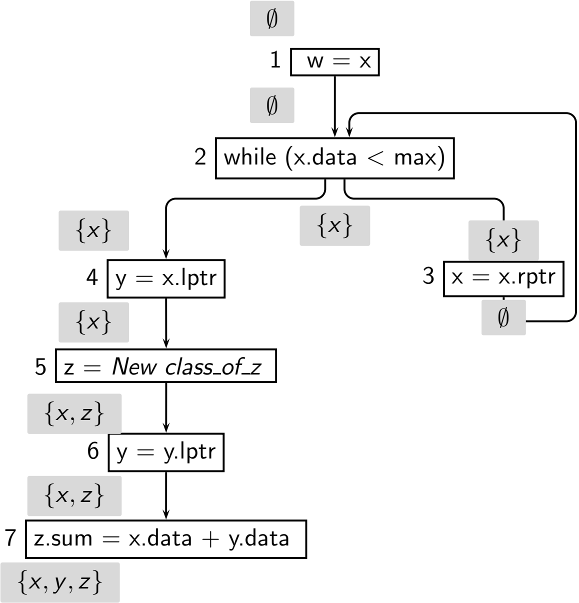

| Union | Path Removal | Factorisation | Extension |
|---|---|---|---|
| $g_3 \cup_G g_4 = g_4$<br>$g_2 \cup_G g_4 = g_5$<br>$g_5 \cup_G g_4 = g_5$<br>$g_5 \cup_G g_6 = g_6$ | $g_6 \ominus x \rightarrow l = g_2$<br>$g_5 \ominus x = \mathcal{E}_G$<br>$g_4 \ominus x \rightarrow r = g_4$<br>$g_4 \ominus x \rightarrow l = g_1$ | $g_2 / (g_1, \{x\}) = \{rg_1\}$<br>$g_5 / (g_1, \{x\}) = \{rg_1, rg_2\}$<br>$g_5 / (g_2, \{r_2\}) = \{\epsilon_{RG}\}$<br>$g_4 / (g_2, \{r_2\}) = \emptyset$ | |

# Access Graph Operations: Examples

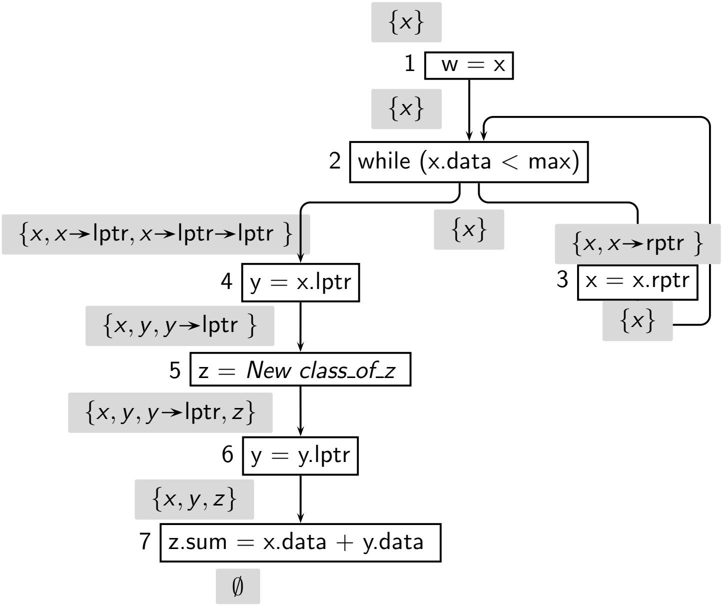



| Union | Path Removal | Factorisation | Extension |
|---|---|---|---|
| $g_3 \cup_G g_4 = g_4$ | $g_6 \ominus x \rightarrow l = g_2$ | $g_2 / (g_1, \{x\}) = \{rg_1\}$ | $(g_3, \{l_1\}) \# \{rg_1\} = g_4$ |
| $g_2 \cup_G g_4 = g_5$ | $g_5 \ominus x = \mathcal{E}_G$ | $g_5 / (g_1, \{x\}) = \{rg_1, rg_2\}$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

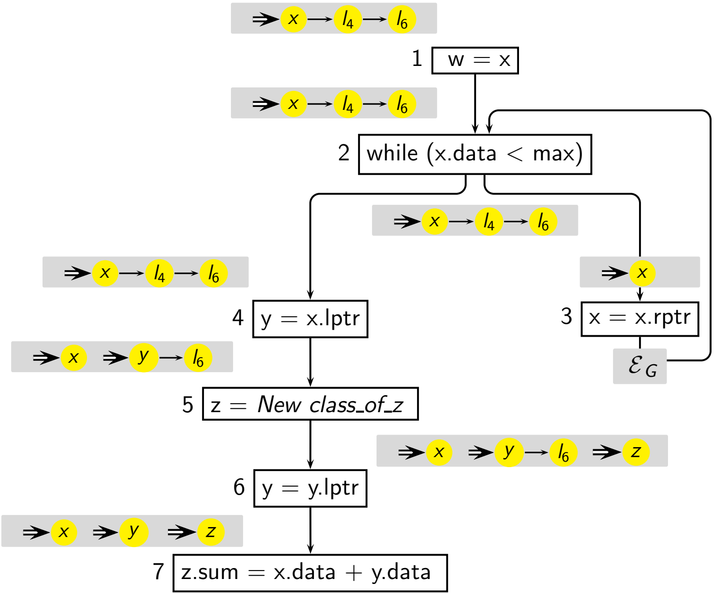
# Availability Analysis of Example Program



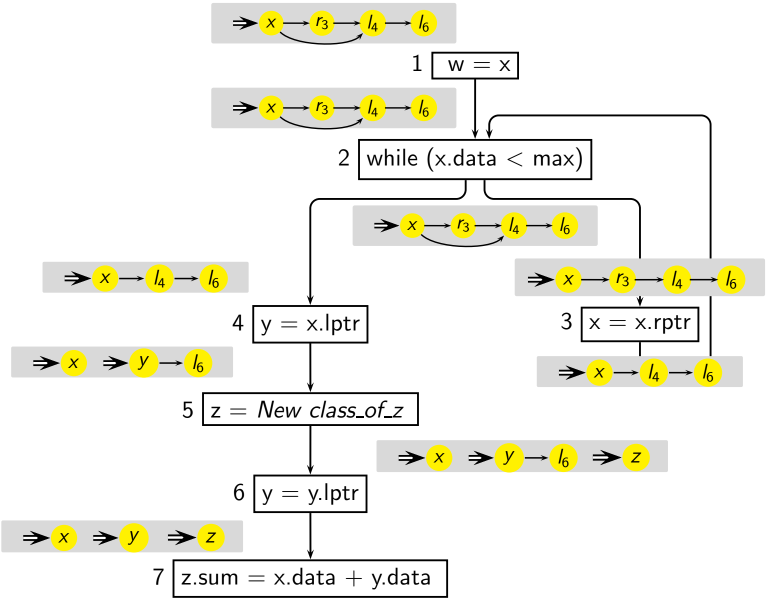
# Anticipability Analysis of Example Program



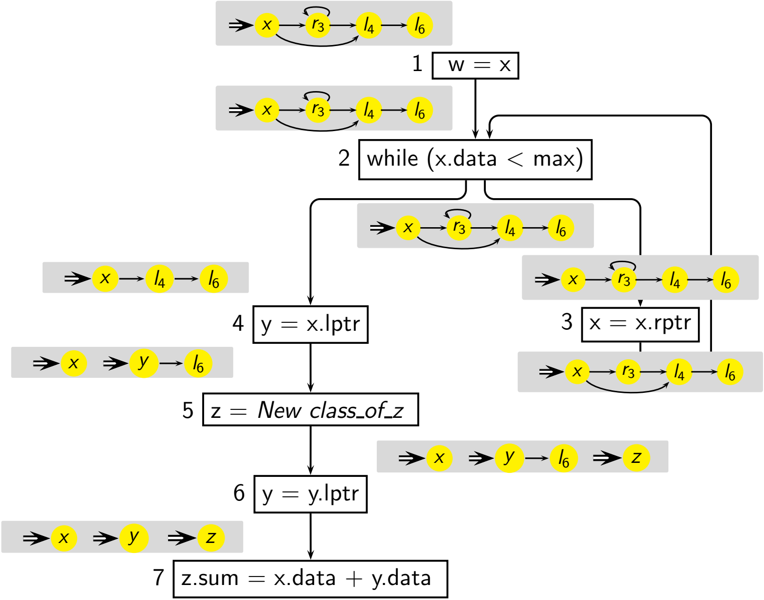
# Liveness Analysis of Example Program: Ist Iteration



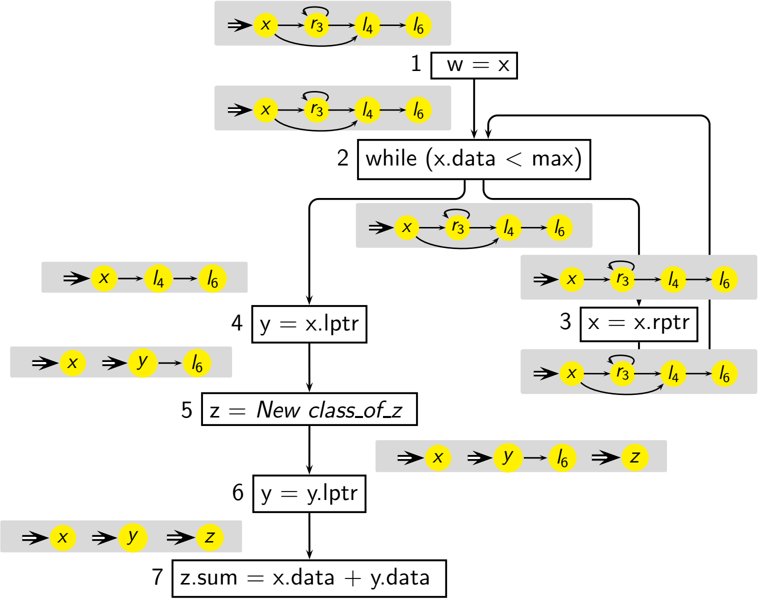
# Liveness Analysis of Example Program: 2nd Iteration



# Liveness Analysis of Example Program: 3rd Iteration



# Liveness Analysis of Example Program: 4th Iteration



# Which Access Paths Can be Nullified?

- Consider extensions of accessible paths for nullification.

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

- For simple access paths, $\rho$ 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 $\rho$ be accessible at $p$ (i.e. available or anticipable)
> **for** each reference field $f$ of the object pointed to by $\rho$
>     **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, $\rho$ is empty and $f$ is the root variable name.

# Which Access Paths Can be Nullified?

Can be safely dereferenced

Consider link aliases at $p$

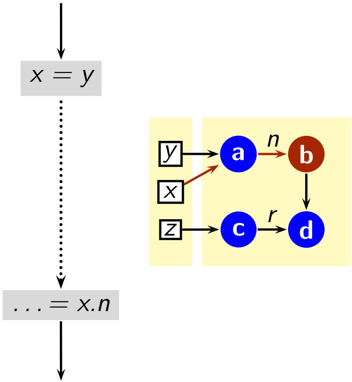
- Consider extensions of accessible paths for nullification.

> Let $\rho$ be accessible at $p$ (i.e. available or anticipable)
> **for** each reference field $f$ of the object pointed to by $\rho$
>   **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, $\rho$ is empty and $f$ is the root variable name.

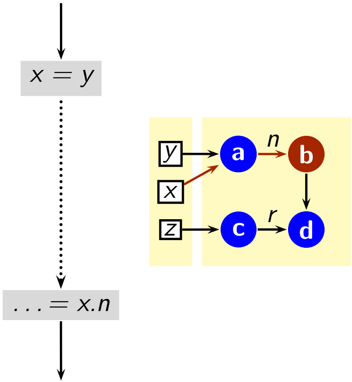
Cannot be hoisted and is not redefined at $p$

# Key Idea #5 : Liveness Closure Under Link Aliasing




$x$ and $y$ are node aliases

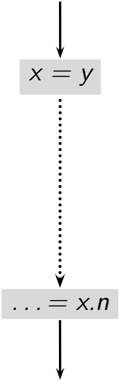
# Key Idea #5 : Liveness Closure Under Link Aliasing

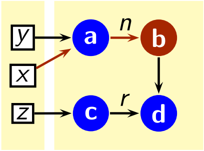



$x$ and $y$ are node aliases

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

# Key Idea #5 : Liveness Closure Under Link Aliasing



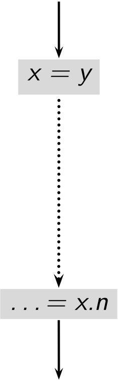



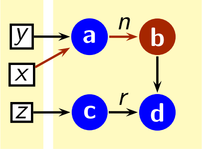

$x$ and $y$ are node aliases

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

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

# Key Idea #5 : 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 $x \rightarrow n$

# Issues in Using Access Graphs for Complete Liveness

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.

# Issues in Using Access Graphs for Complete Liveness

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.
- Implicit Liveness at $p$
  Access paths that become live under link alias closure.

# Issues in Using Access Graphs for Complete Liveness

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.
- Implicit Liveness at $p$
  Access paths that become live under link alias closure.
  - The set of implicitly live access paths may not be prefix closed.

# Issues in Using Access Graphs for Complete Liveness

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.
- Implicit Liveness at $p$
  Access paths that become live under link alias closure.
  - The set of implicitly live access paths may not be prefix closed.
  - These *paths* are not accessed, their frontiers are accessed through some other access path

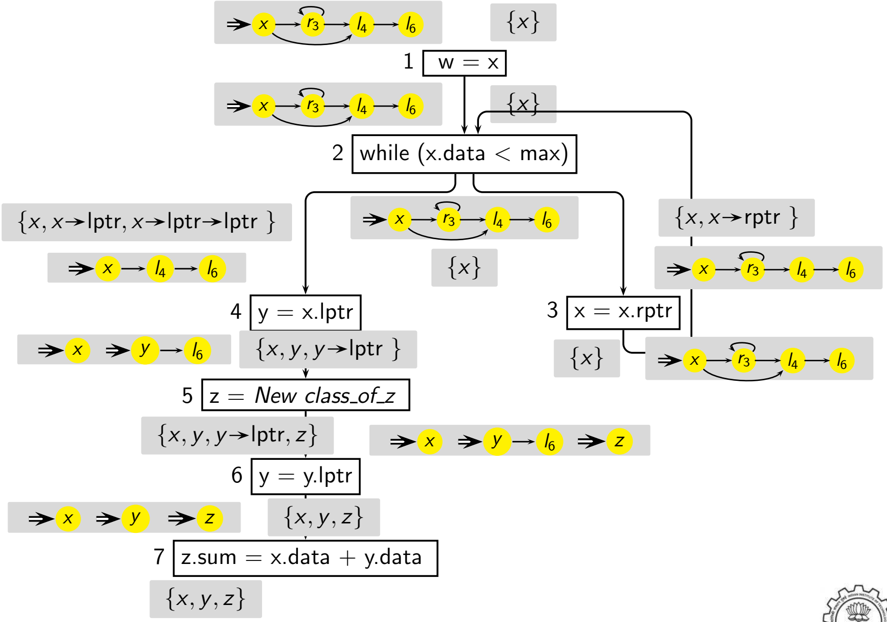
# Issues in Using Access Graphs for Complete Liveness

- Explicit Liveness at *p*
  Liveness purely due to the program beyond *p*.
  The effect of execution before *p* is not incorporated.
- Implicit Liveness at *p*
  Access paths that become live under link alias closure.
  - The set of implicitly live access paths may not be prefix closed.
  - These *paths* are not accessed, their frontiers are accessed through some other access path
- All paths in an access graph may not be access paths

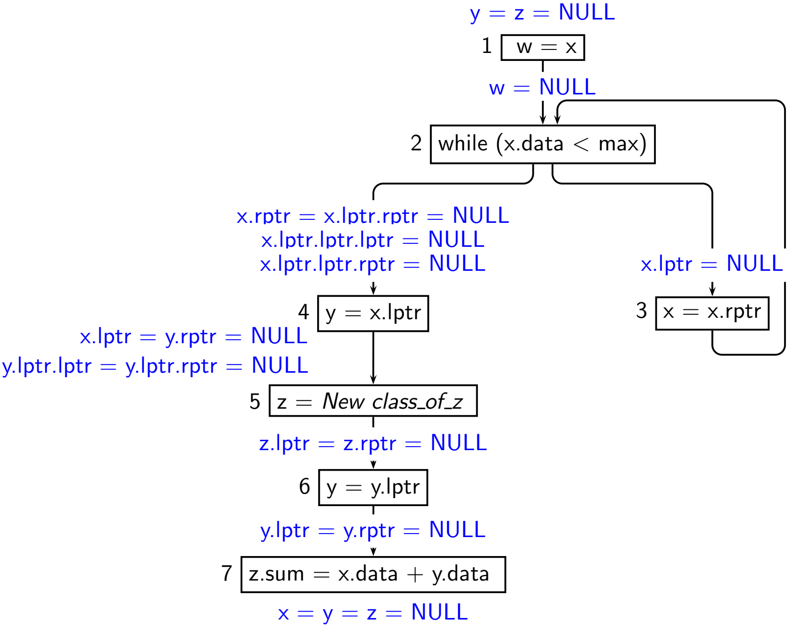
# Issues in Using Access Graphs for Complete Liveness

- Explicit Liveness at $p$
  Liveness purely due to the program beyond $p$.
  The effect of execution before $p$ is not incorporated.
- Implicit Liveness at $p$
  Access paths that become live under link alias closure.
  - The set of implicitly live access paths may not be prefix closed.
  - These *paths* are not accessed, their frontiers are accessed through some other access path
- All paths in an access graph may not be access paths
  - Define intermediate and final nodes in access graphs
  - Paths ending on final nodes are access paths through some other access path

# 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 |

# Issues Not Covered in This Presentation

- Precision of information
  - Implicit Vs. Explicit Liveness
  - May Vs. Must Alias Analysis
  - Cyclic Data Structure
  - Eliminating Redundant NULL Assignments
- Properties of Data Flow Analysis:
  Monotonicity, Distributivity, Boundedness, Complexity
- Interprocedural Analysis
- Extensions for C/C++

*Part 9*

# *Conclusions*

# Conclusions

- Data flow analysis is a very powerful program analysis technique
- Requires us to design appropriate
  - Set of values with reasonable approximations

    $\Rightarrow$ Acceptable partial order and merge operation
  - Monotonic functions which are closed under composition

# Conclusions

- Data flow analysis can be used for discovering complex semantics
- Unbounded information can summarized using interesting insights
  - Example: Heap Analysis

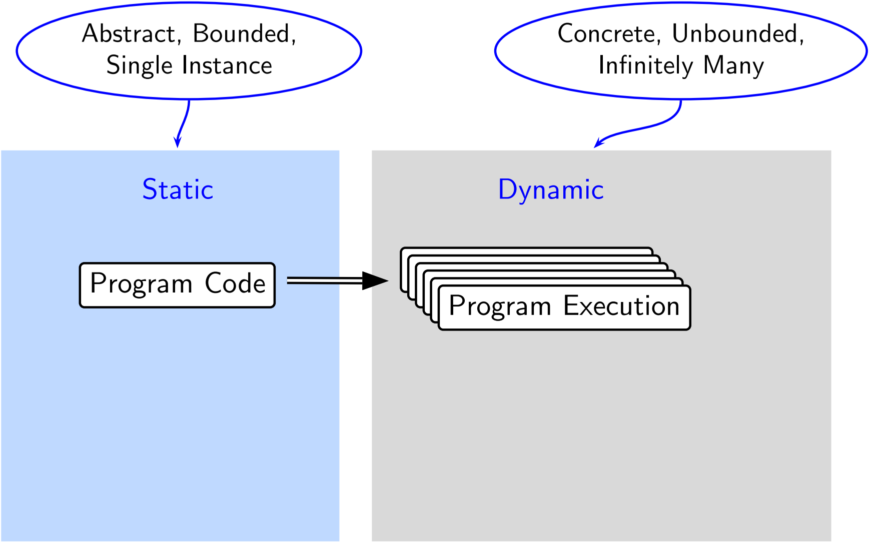
    *Heap manipulations consist of repeating patterns which bear a close resemblance to program structure*

    Analysis of heap data is possible despite the fact that the mappings between access expressions and l-values keep changing
  - The basic theory can be applied by a careful design of represenations and operations

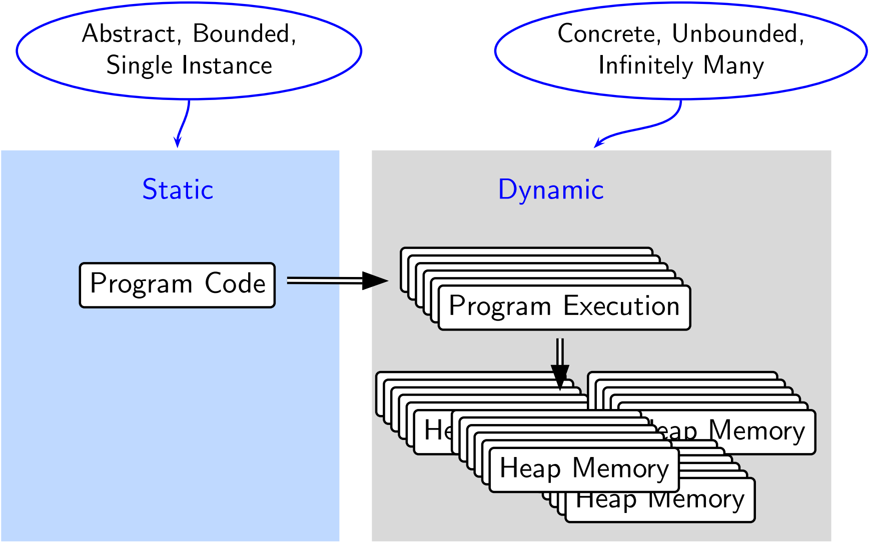
# BTW, What is Static Analysis of Heap?

| Static | Dynamic |
|---|---|

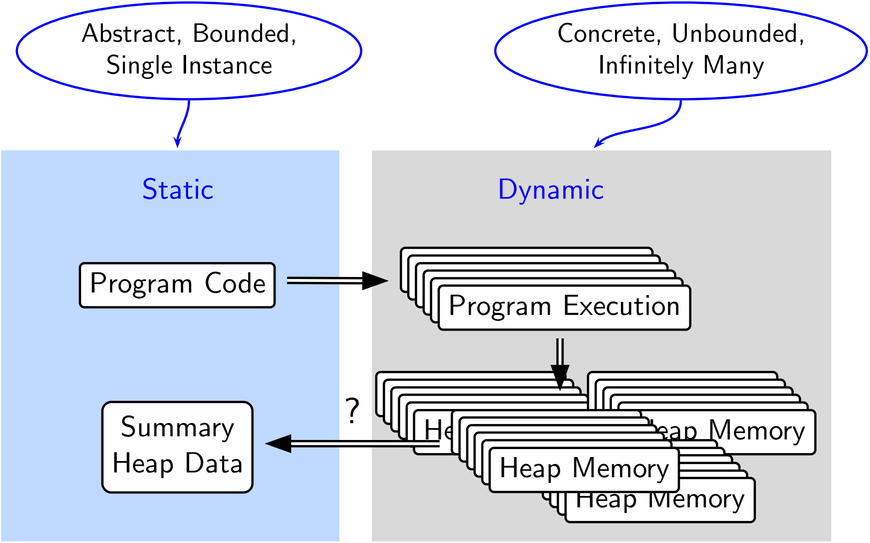
# BTW, What is Static Analysis of Heap?



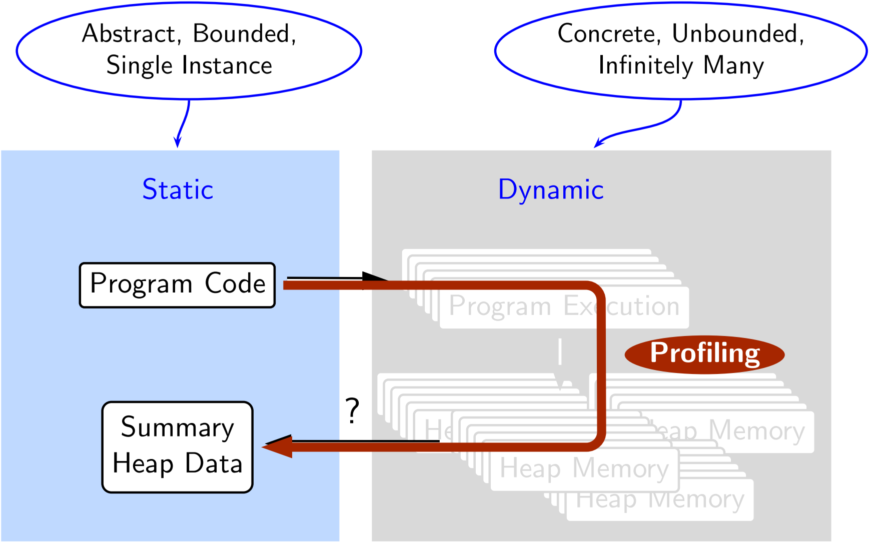
# BTW, What is Static Analysis of Heap?



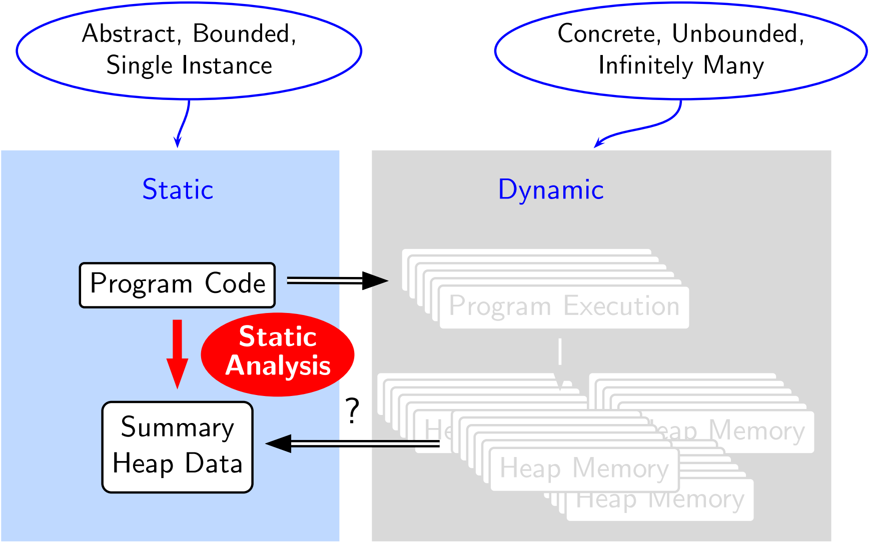
# BTW, What is Static Analysis of Heap?



# BTW, What is Static Analysis of Heap?



# BTW, What is Static Analysis of Heap?



# Last but not the least . . .

*Thank You!*