Introduction to Program Analysis

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

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

July 2017
Part 1

About These Slides
Copyright

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

  (Indian edition published by Ane Books in 2013)

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


*These slides are being made available under GNU FDL v1.2 or later purely for academic or research use.*
Motivating the Need of Program Analysis

- Some representative examples
  - Classical optimizations performed by compilers
  - Optimizing heap memory usage
- Course details, schedule, assessment policies etc.
- Program Model
- Soundness and Precision
Part 2

Classical Optimizations
Examples of Optimising Transformations (ALSU, 2006)

A C program and its optimizations

```c
void quicksort(int m, int n)
{
    int i, j, v, x;
    if (n <= m) return;
    i = m-1; j = n; v = a[n];
    while(1)
    {
        do i = i + 1; while (a[i] < v);
        do j = j - 1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;
    quicksort(m,i); quicksort(i+1,n);
}
```

July 2017
Intermediate Code

For the boxed source code

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

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

- Identify maximal sequences of linear control flow
  ⇒ Basic Blocks

- No transfer into or out of basic blocks except the first and last statements
  Control transfer into the block: only at the first statement.
  Control transfer out of the block: only at the last statement.
Intermediate Code with Basic Blocks

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

B1

\[ i = m - 1 \]
\[ j = n \]
\[ t1 = 4 \times n \]
\[ t6 = a[t1] \]
\[ v = t6 \]

B2

\[ i = i + 1 \]
\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ \text{if } t3 < v \text{ goto B2} \]

B3

\[ j = j - 1 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ \text{if } t5 > v \text{ goto B3} \]

B4

\[ \text{if } i \geq j \text{ goto B6} \]

B6

\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ x = t3 \]
\[ t2 = 4 \times i \]
\[ t1 = 4 \times n \]
\[ t6 = a[t1] \]
\[ a[t2] = t6 \]
\[ a[t2] = t6 \]
\[ a[t1] = x \]

B5

\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ x = t3 \]
\[ t2 = 4 \times i \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ a[t2] = t5 \]
\[ t4 = 4 \times j \]
\[ a[t4] = x \]

\[ \text{goto B2} \]
## Program Flow Graph: Observations

<table>
<thead>
<tr>
<th>Nesting Level</th>
<th>Basic Blocks</th>
<th>No. of Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B1, B6</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>B4, B5</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>B2, B3</td>
<td>8</td>
</tr>
</tbody>
</table>
Local Common Subexpression Elimination

B1

\[ i = m - 1 \]
\[ j = n \]
\[ t1 = 4 \times n \]
\[ t6 = a[t1] \]
\[ v = t6 \]

B2

\[ i = i + 1 \]
\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ \text{if } t3 < v \text{ goto B2} \]

B3

\[ j = j - 1 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ \text{if } t5 > v \text{ goto B3} \]

B4

\[ \text{if } i \geq j \text{ goto B6} \]

B5

\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ x = t3 \]
\[ t2 = 4 \times i \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ a[t2] = t5 \]
\[ t4 = 4 \times j \]
\[ a[t4] = x \]
\[ \text{goto B2} \]
Local Common Subexpression Elimination

B1

\[ i = m - 1 \]
\[ j = n \]
\[ t1 = 4 \times n \]
\[ t6 = a[t1] \]
\[ v = t6 \]

B2

\[ i = i + 1 \]
\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ \text{if } t3 < v \text{ goto B2} \]

B3

\[ j = j - 1 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ \text{if } t5 > v \text{ goto B3} \]

B4

\[ \text{if } i \geq j \text{ goto B6} \]

B6

\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ x = t3 \]
\[ t1 = 4 \times n \]
\[ t6 = a[t1] \]
\[ a[t2] = t6 \]
\[ a[t1] = x \]

B5

\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ x = t3 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ a[t2] = t5 \]
\[ a[t4] = x \]
\[ \text{goto B2} \]
Global Common Subexpression Elimination

```
B1
i = m - 1
j = n
i = i + 1
j = j - 1
i = i + 1
j = j - 1
if i >= j goto B6
B2
i = i + 1
t2 = 4 * i
t3 = a[t2]
if t3 < v goto B2
B3
j = j - 1
t4 = 4 * j
t5 = a[t4]
if t5 > v goto B3
B4
if i >= j goto B6
B5

B6
B2
B3
B4
B5
```

```
i = m - 1
j = n
t1 = 4 * n
t6 = a[t1]
v = t6
t2 = 4 * i
t3 = a[t2]
t2 = 4 * i
t3 = a[t2]
t4 = 4 * j
t5 = a[t4]
t2 = 4 * i
t3 = a[t2]
t4 = 4 * j
t5 = a[t4]
t2 = 4 * i
t3 = a[t2]
t4 = 4 * j
t5 = a[t4]
t2 = 4 * i
t3 = a[t2]
t4 = 4 * j
t5 = a[t4]
t2 = 4 * i
t3 = a[t2]
t4 = 4 * j
t5 = a[t4]
at2 = t6
x = t3
a[t2] = t6
a[t1] = x
a[t4] = x
a[t2] = t5
a[t4] = x
```
Global Common Subexpression Elimination

B1
\[
\begin{align*}
    i &= m - 1 \\
    j &= n \\
    t1 &= 4 \times n \\
    t6 &= a[t1] \\
    v &= t6
\end{align*}
\]

B2
\[
\begin{align*}
    i &= i + 1 \\
    t2 &= 4 \times i \\
    t3 &= a[t2] \\
    \text{if } t3 < v \text{ goto B2}
\end{align*}
\]

B3
\[
\begin{align*}
    j &= j - 1 \\
    t4 &= 4 \times j \\
    t5 &= a[t4] \\
    \text{if } t5 > v \text{ goto B3}
\end{align*}
\]

B4
\[
\begin{align*}
    \text{if } i >= j \text{ goto B6}
\end{align*}
\]

B5
\[
\begin{align*}
    t2 &= 4 \times i \\
    t3 &= a[t2] \\
    x &= t3 \\
    t4 &= 4 \times j \\
    t5 &= a[t4] \\
    a[t2] &= t5 \\
    a[t4] &= x \\
    \text{goto B2}
\end{align*}
\]

...
Global Common Subexpression Elimination

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

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

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

B4
if i >= j goto B6

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

t2 = 4 * i

B6
t2 = 4 * i
t3 = a[t2]
x = t3
t4 = 4 * j
t5 = a[t4]
a[t2] = t5
a[t4] = x
goto B2

...
Global Common Subexpression Elimination

B1

\[ i = m - 1 \]
\[ j = n \]
\[ t1 = 4 \times n \]
\[ t6 = a[t1] \]
\[ v = t6 \]

B2

\[ i = i + 1 \]
\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
if \( t3 < v \) goto B2

B3

\[ j = j - 1 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
if \( t5 > v \) goto B3

B4

if \( i \geq j \) goto B6

B5

\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ x = t3 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ a[t2] = t5 \]
\[ a[t4] = x \]
goto B2

B6

\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ x = t3 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ a[t2] = t6 \]
\[ a[t1] = x \]
Global Common Subexpression Elimination

B1

\[
i = m - 1 \\
j = n \\
t1 = 4 \cdot n \\
t6 = a[t1] \\
v = t6
\]

B2

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

B3

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

B4

\[
if i >= j goto B6 \\
a[t1] = x
\]

B5

\[
x = t3 \\
a[t2] = t5 \\
a[t4] = x
\]

B6

\[
 x = t3 \\
t6 = a[t1] \\
a[t2] = t6 \\
a[t1] = x
\]
Other Classical Optimizations

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

B1

\[ i = m - 1 \]
\[ j = n \]
\[ t1 = 4 \times n \]
\[ t6 = a[t1] \]
\[ v = t6 \]

B2

\[ i = i + 1 \]
\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ \text{if } t3 < v \text{ goto B2} \]

B3

\[ j = j - 1 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ \text{if } t5 > v \text{ goto B3} \]

B4

\[ \text{if } i \geq j \text{ goto B6} \]

B5

\[ x = t3 \]
\[ a[t2] = t5 \]
\[ a[t4] = x \]
\[ \text{goto B2} \]

B6

\[ x = t3 \]
\[ t6 = a[t1] \]
\[ a[t2] = t6 \]
\[ a[t1] = x \]
Copy Propagation and Dead Code Elimination

B1

\[ i = m - 1 \]
\[ j = n \]
\[ t1 = 4 \times n \]
\[ t6 = a[t1] \]
\[ v = t6 \]

B2

\[ i = i + 1 \]
\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ \text{if } t3 < v \text{ goto B2} \]

B3

\[ j = j - 1 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ \text{if } t5 > v \text{ goto B3} \]

B4

\[ \text{if } i \geq j \text{ goto B6} \]

B5

\[ x = t3 \]
\[ a[t2] = t5 \]
\[ a[t4] = t3 \]
\[ \text{goto B2} \]

B6

\[ x = t3 \]
\[ t6 = a[t1] \]
\[ a[t2] = t6 \]
\[ a[t1] = t3 \]
Copy Propagation and Dead Code Elimination

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

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

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

if i >= j goto B6

if i >= j goto B6

a[t2] = t5
a[t4] = t3
goto B2
Strength Reduction and Induction Variable Elimination

B1

\[
i = m - 1
\]
\[
j = n
\]
\[
t1 = 4 \times n
\]
\[
t6 = a[t1]
\]
\[
v = t6
\]

B2

\[
i = i + 1
\]
\[
t2 = 4 \times i
\]
\[
t3 = a[t2]
\]
\[
\text{if } t3 < v \text{ goto B2}
\]

B3

\[
j = j - 1
\]
\[
t4 = 4 \times j
\]
\[
t5 = a[t4]
\]
\[
\text{if } t5 > v \text{ goto B3}
\]

B4

\[
\text{if } i \geq j \text{ goto B6}
\]

B5

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

B6

\[
t6 = a[t1]
\]
\[
a[t2] = t6
\]
\[
a[t1] = t3
\]
**Strength Reduction and Induction Variable Elimination**

**B1**
- \( i = m - 1 \)
- \( j = n \)
- \( t_1 = 4 \times n \)
- \( t_6 = a[t_1] \)
- \( v = t_6 \)
- \( t_2 = 4 \times i \)
- \( t_4 = 4 \times j \)

**B2**
- \( t_2 = t_2 + 4 \)
- \( t_3 = a[t_2] \)
- if \( t_3 < v \) goto B2

**B3**
- \( t_4 = t_4 - 4 \)
- \( t_5 = a[t_4] \)
- if \( t_5 > v \) goto B3

**B4**
- if \( t_2 \geq t_4 \) goto B6

**B5**
- \( a[t_2] = t_5 \)
- \( a[t_4] = t_3 \)
- goto B2

**B6**
- \( t_6 = a[t_1] \)
- \( a[t_2] = t_6 \)
- \( a[t_1] = t_3 \)
Final Intermediate Code

B1

\[
\begin{align*}
i &= m - 1 \\
j &= n \\
t1 &= 4 \times n \\
t6 &= a[t1] \\
v &= t6 \\
t2 &= 4 \times i \\
t4 &= 4 \times j
\end{align*}
\]

B2

\[
\begin{align*}
t2 &= t2 + 4 \\
t3 &= a[t2] \\
\text{if } t3 < v \text{ goto B2}
\end{align*}
\]

B3

\[
\begin{align*}
t4 &= t4 - 4 \\
t5 &= a[t4] \\
\text{if } t5 > v \text{ goto B3}
\end{align*}
\]

B4

\[
\begin{align*}
t1 &= 4 \times n \\
t6 &= a[t1] \\
a[t2] &= t6 \\
a[t1] &= t3 \\
t2 &= 4 \times i \\
t3 &= a[t2] \\
x &= t3 \\
t2 &= 4 \times i \\
t4 &= 4 \times j \\
a[t4] &= t3 \\
a[t2] &= t5 \\
\text{goto B2}
\end{align*}
\]

B5

\[
\begin{align*}
a[t2] &= t5 \\
a[t4] &= t3
\end{align*}
\]

B6
Optimized Program Flow Graph

<table>
<thead>
<tr>
<th>Nesting Level</th>
<th>No. of Statements</th>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>Code Motion Redundancy Elimination Control flow Optimization</th>
<th>Machine Independent</th>
<th>Flow Analysis (Data + Control)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Transformations</td>
<td>Machine Dependent</td>
<td>Dependence Analysis (Data + Control)</td>
</tr>
<tr>
<td>Instruction Scheduling Register Allocation Peephole Optimization</td>
<td>Machine Dependent</td>
<td>Several Independent Techniques</td>
</tr>
<tr>
<td>Vectorization Parallelization</td>
<td>Machine Dependent</td>
<td>Dependence Analysis (Data + Control)</td>
</tr>
</tbody>
</table>
What is Program Analysis?

Discovering information about a given program
What is Program Analysis?

Discovering information about a given program

- Representing the dynamic behaviour of the program
What is Program Analysis?

Discovering information about a given program

- Representing the dynamic behaviour of the program
- Most often obtained without executing the program
  - Static analysis Vs. Dynamic Analysis
  - Example of loop tiling for parallelization
Discovering information about a given program

- Representing the dynamic behaviour of the program
- Most often obtained without executing the program
  - Static analysis Vs. Dynamic Analysis
  - Example of loop tiling for parallelization
- Must represent all execution instances of the program
Why is it Useful?

- Code optimization
  - Improving time, space, energy, or power efficiency
  - Compilation for special architecture (eg. multi-core)
Why is it Useful?

- **Code optimization**
  - Improving time, space, energy, or power efficiency
  - Compilation for special architecture (e.g., multi-core)

- **Verification and validation**
  Giving guarantees such as: The program will
  - never divide a number by zero
  - never dereference a NULL pointer
  - close all opened files, all opened socket connections
  - not allow buffer overflow security violation
Why is it Useful?

- **Code optimization**
  - Improving time, space, energy, or power efficiency
  - Compilation for special architecture (e.g., multi-core)

- **Verification and validation**
  Giving guarantees such as: The program will
  - never divide a number by zero
  - never dereference a NULL pointer
  - close all opened files, all opened socket connections
  - not allow buffer overflow security violation

- **Software engineering**
  - Maintenance, bug fixes, enhancements, migration
  - Example: Y2K problem
Why is it Useful?

• Code optimization
  ▶ Improving time, space, energy, or power efficiency
  ▶ Compilation for special architecture (e.g. multi-core)

• Verification and validation
  Giving guarantees such as: The program will
  ▶ never divide a number by zero
  ▶ never dereference a NULL pointer
  ▶ close all opened files, all opened socket connections
  ▶ not allow buffer overflow security violation

• Software engineering
  ▶ Maintenance, bug fixes, enhancements, migration
  ▶ Example: Y2K problem

• Reverse engineering
  To understand the program
Part 3

Optimizing Heap Memory Usage
Heap allocation provides the flexibility of

- **Variable Sizes.** Data structures can grow or shrink as desired at runtime.
  
  (Not bound to the declarations in program.)

- **Variable Lifetimes.** Data structures can be created and destroyed as desired at runtime.
  
  (Not bound to the activations of procedures.)
Managing Heap Memory

Decision 1: When to Allocate?

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

Decision 1: When to Allocate?

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

Decision 2: When to Deallocate?

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

- Memory leaks
  10% to 20% of last development effort goes in plugging leaks

- Tool assisted manual plugging
  *Purify, Electric Fence, RootCause, GlowCode, yakTest, Leak Tracer, BDW Garbage Collector, mtrace, memwatch, dmalloc etc.*

- All leak detectors
  - are dynamic (and hence specific to execution instances)
  - generate massive reports to be perused by programmers
  - usually do not locate last use but only allocation escaping a call
  ⇒ At which program point should a leak be “plugged”? 
Garbage Collection \equiv \text{AutomaticDeallocation}

- Retain active data structure.
  Deallocate inactive data structure.

- What is an Active Data Structure?
Garbage Collection ≡ Automatic Deallocation

- Retain active data structure.
  Deallocate inactive data structure.

- What is an Active Data Structure?

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

- Retain active data structure.
  Deallocate inactive data structure.

- What is an Active Data Structure?

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

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

1. \( w = x \) // \( x \) points to \( m_a \)
2. \( \text{if} \ (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New class_of}_z \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

Stack

Heap

Garbage
What is Garbage?

1. \( w = x \quad // \quad \text{x points to } m_a \)
2. \( \text{if } (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New class_of_z} \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
What is Garbage?

1. \( w = x \)  
   
   // \( x \) points to \( m_a \)

2. if \( (x.data < \text{max}) \)

3. \( x = x.rptr \)

4. \( y = x.lptr \)

5. \( z = \text{New class of } z \)

6. \( y = y.lptr \)

7. \( z.sum = x.data + y.data \)
What is Garbage?

1. \( w = x \)  // \( x \) points to \( m_a \)
2. if \( (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New class of } z \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

All white nodes are unused and should be considered garbage.
Is Reachable Same as Live?

From www.memorymanagement.org/glossary

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

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

Similar terms: reachable. Opposites: dead. See also: undead.
Is Reachable Same as Live?

- Not really. Most of us know that.

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

- The state of art compilers, virtual machines, garbage collectors cannot distinguish between the two.
Comparison between different sets of objects:

Live  ?  Reachable  ?  Allocated
Reachability and Liveness

Comparison between different sets of objects:

\[
\text{Live } \subseteq \text{ Reachable } \subseteq \text{ Allocated}
\]
Reachability and Liveness

Comparison between different sets of objects:

\[ \text{Live} \subseteq \text{Reachable} \subseteq \text{Allocated} \]

The objects that are not live must be reclaimed.
Reachability and Liveness

Comparison between different sets of objects:

\[
\text{Live } \subseteq \text{ Reachable } \subseteq \text{ Allocated}
\]

The objects that are not live must be reclaimed.

\[
\neg \text{ Live } \quad ? \quad \neg \text{ Reachable } \quad ? \quad \neg \text{ Allocated}
\]
Reachability and Liveness

Comparison between different sets of objects:

\[ \text{Live} \subseteq \text{Reachable} \subseteq \text{Allocated} \]

The objects that are not live must be reclaimed.

\[ \neg \text{Live} \supseteq \neg \text{Reachable} \supseteq \neg \text{Allocated} \]
Reachability and Liveness

Comparison between different sets of objects:

\[ \text{Live} \subseteq \text{Reachable} \subseteq \text{Allocated} \]

The objects that are not live must be reclaimed.

\[ \neg \text{Live} \supseteq \neg \text{Reachable} \supseteq \neg \text{Allocated} \]

Garbage collectors collect these
Cedar Mesa Folk Wisdom

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

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

- Most promising, simplest to understand, yet the hardest to implement.

- Which references should be set to NULL?
  - Most approaches rely on feedback from profiling.
  - No systematic and clean solution.
Distinguishing Between Reachable and Live

The state of art

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

1. \( w = x \) // \( x \) points to \( m_a \)
2. \( \text{while } (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New class_of_z} \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

if changed to \texttt{while}
Liveness of Stack Data: An Informal Introduction

1. \( w = x \quad // \quad x \text{ points to } m_a \)
2. \( \text{while} \ (x.\text{data} < \max) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New class_of_z} \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

What is the meaning of the use of data?
Liveness of Stack Data: An Informal Introduction

1. \( w = x \quad // \text{x points to } m_a \)
2. \( \text{while } (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New \ class\_of\_z} \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

What is the meaning of the use of data?
Liveness of Stack Data: An Informal Introduction

1. \( w = x \)  // \( x \) points to \( m_a \)
2. while \((x.data < \text{max})\)
3. \( x = x.rptr \)
4. \( y = x.lptr \)
5. \( z = \text{New class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)

Accessing the location and reading its contents

What is the meaning of the use of data?
Liveness of Stack Data: An Informal Introduction

1. \( w = x \)  // \( x \) points to \( m_a \)
2. \( \text{while} \ (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New class}\_\text{of}_z \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

Accessing the location and reading its contents

Stack

Heap

Reading x (Stack data)
Liveness of Stack Data: An Informal Introduction

1. \( w = x \) // \( x \) points to \( m_a \)
2. \( \text{while} \ (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New class of } z \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
Liveness of Stack Data: An Informal Introduction

1. \( w = x \)  // \( x \) points to \( m_a \)
2. \( \text{while} \ (x\text{.data} < \text{max}) \)
3. \( x = x\text{.rptr} \)
4. \( y = x\text{.lptr} \)
5. \( z = \text{New class_of_z} \)
6. \( y = y\text{.lptr} \)
7. \( z\text{.sum} = x\text{.data} + y\text{.data} \)

Accessing the location and reading its contents.

Reading \( x\text{.rptr} \) (Heap data)

Heap

Stack
Liveness of Stack Data: An Informal Introduction

```
w = x

while (x.data < max)
  x = x.rptr

y = x.lptr

z = New class of z

y = y.lptr

z.sum = x.data + y.data
```

No variable is used beyond this program point
Liveness of Stack Data: An Informal Introduction

w = x

while (x.data < max)

x = x.rptr

y = x.lptr

z = New class of z

y = y.lptr

z.sum = x.data + y.data

Current values of x, y, and z are used beyond this program point
Liveness of Stack Data: An Informal Introduction

- Current values of x, y, and z are used beyond this program point
- The value of y is different before and after the assignment to y
Liveness of Stack Data: An Informal Introduction

\[
\begin{align*}
  &w = x \\
  &\text{while (}x.\text{data} < \text{max}) \\
  &\quad x = x.\text{rptr} \\
  &y = x.\text{lptr} \\
  &z = \text{New class of } z \\
  &\quad y = y.\text{lptr} \\
  &\quad z.\text{sum} = x.\text{data} + y.\text{data}
\end{align*}
\]

- The current values of x and y are used beyond this program point
- The current value of z is not used beyond this program point
Liveness of Stack Data: An Informal Introduction

- The current values of `x` is used beyond this program point
- Current values of `y` and `z` are not used beyond this program point
Liveness of Stack Data: An Informal Introduction

- Nothing is known as of now
- Some information will be available in the next iteration point
Liveness of Stack Data: An Informal Introduction

- Current value of x is used beyond this program point
- However its value is different before and after the assignment
Liveness of Stack Data: An Informal Introduction

Let's consider the following program:

```c
w = x

while (x.data < max)
    x = x.rptr
    y = x.lptr
    z = New class of z
    y = y.lptr

z.sum = x.data + y.data
```

- Current value of `x` is used beyond this program point.
- There are two control flow paths beyond this program point.
Liveness of Stack Data: An Informal Introduction

Current value of x is used beyond this program point

\( w = x \)

\( \text{while (} x.\text{data} < \text{max}) \)

\( x = x.\text{rptr} \)

\( y = x.\text{lptr} \)

\( z = \text{New class of } z \)

\( y = y.\text{lptr} \)

\( z.\text{sum} = x.\text{data} + y.\text{data} \)
Liveness of Stack Data: An Informal Introduction

```
while (x.data < max)
    x = x.rptr
    y = x.lptr
    z = New class of z
    y = y.lptr
    z.sum = x.data + y.data
```

Current value of x is used beyond this program point.
Liveness of Stack Data: An Informal Introduction

\[
\begin{align*}
&\text{while } (x.data < \text{max}) \\
&w = x \\
&x = x.rptr \\
&y = x.lptr \\
&z = \text{New class of } z \\
&y = y.lptr \\
&z.sum = x.data + y.data
\end{align*}
\]

End of iteration #1
Liveness of Stack Data: An Informal Introduction


drawings and diagrams

w = x

while (x.data < max)

x = x.rptr

y = x.lptr

z = New class of z

y = y.lptr

z.sum = x.data + y.data

End of iteration #2

Live

Dead

July 2017
Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data
If the while loop is not executed even once.

1. \( w = x \quad // x \text{ points to } m_a \)
2. \( \text{while } (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New class of } z \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data
If the while loop is executed once.

1. \( w = x \) // \( x \) points to \( m_a \)
2. \( \text{while} \ (x.\text{data} < \text{max}) \)
3. \( x = x.\text{rptr} \)
4. \( y = x.\text{lptr} \)
5. \( z = \text{New class_of_z} \)
6. \( y = y.\text{lptr} \)
7. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data
If the while loop is executed twice.

```
1 w = x  // x points to ma
2 while (x.data < max)
3   x = x.rptr
4 y = x.lptr
5 z = New class_of_z
6 y = y.lptr
7 z.sum = x.data + y.data
```
The Moral of the Story

- Mappings between access expressions and l-values keep changing

- This is a *rule* for heap data
  For stack and static data, it is an *exception*!

- Static analysis of programs has made significant progress for stack and static data.

What about heap data?

- Given two access expressions at a program point, do they have the same l-value?
- Given the same access expression at two program points, does it have the same l-value?
Our Solution

```
1       w = x
        w = null
2   while (x.data < max)
      {      x.lptr = null
      x = x.rptr   }
3       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
   }
3 x.rptr = x.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

\[
y = z = \text{null}
\]

1. \( w = x \)
   \( w = \text{null} \)

2. \( \text{while} \ (x.\text{data} < \text{max}) \)
   \[
   \{
   \begin{align*}
   x.\text{lptr} &= \text{null} \\
   x &= x.\text{rptr}
   \end{align*}
   \}
   \]
   \( x.\text{rptr} = x.\text{lptr}.\text{rptr} = \text{null} \)
   \( x.\text{lptr}.\text{lptr}.\text{lptr} = \text{null} \)
   \( x.\text{lptr}.\text{lptr}.\text{rptr} = \text{null} \)

3. \( y = x.\text{lptr} \)
   \( x.\text{lptr} = y.\text{rptr} = \text{null} \)
   \( y.\text{lptr}.\text{lptr} = y.\text{lptr}.\text{rptr} = \text{null} \)

4. \( z = \text{New class of z} \)
   \( z.\text{lptr} = z.\text{rptr} = \text{null} \)

5. \( y = y.\text{lptr} \)
   \( y.\text{lptr} = y.\text{rptr} = \text{null} \)

6. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
   \( x = y = z = \text{null} \)

While loop is not executed even once

Stack

Heap

\( \text{HeapStack} \)

\( \begin{align*}
  & p \\
  & \quad a \\
  & \quad \quad b \\
  & \quad \quad \quad c \\
  & \quad \quad \quad \quad d \\
  & \quad i \\
  & \quad \quad j \\
  & \quad \quad \quad k \\
  & \quad \quad \quad \quad l \\
  & \quad \quad \quad \quad m \\
  & \quad \quad \quad \quad n \\
  & \quad \quad \quad \quad o \\
\end{align*} \)
**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
    }
3  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
6  y = y.lptr
    y.lptr = y.rptr = null
5  z.sum = x.data + y.data
4  y = y.lptr
    y.lptr = y.rptr = null
3  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
      x = x.rptr
   }  x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
3  y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
4  z = New class
   z.lptr = z.rptr = null
5  y = y.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
6  z.sum = x.data + y.data
6  x = y = z = null
```

While loop is not executed even once

Stack

Heap

July 2017
Our Solution

1. \( y = z = \text{null} \)
2. \( w = x \)
3. \( w = \text{null} \)
4. While loop is not executed even once
5. \( x.lptr = \text{null} \)
6. \( x = x.rptr \)
7. \( x.rptr = x.lptr.rptr = \text{null} \)
8. \( x.lptr.lptr.lptr = \text{null} \)
9. \( x.lptr.lptr.rptr = \text{null} \)
10. \( y = x.lptr \)
11. \( x.lptr = y.rptr = \text{null} \)
12. \( y.lptr.lptr = y.lptr.rptr = \text{null} \)
13. \( z = \text{null} \)
14. \( z.lptr = z.rptr = \text{null} \)
15. \( y = y.lptr \)
16. \( y.lptr = y.rptr = \text{null} \)
17. \( z \text{sum} = x.data + y.data \)
18. \( x = y = z = \text{null} \)
Our Solution

```
y = z = null
1 w = x
   w = null
2 while (x.data < max)
   { x.lptr = null
     x = x.rptr
   } x.rptr = x.lptr.rptr = null
   x.lptr.lptr.lptr = null
   x.lptr.lptr.rptr = null
3 y = x.lptr
   x.lptr = y.rptr = null
   y.lptr.lptr = y.lptr.rptr = null
4 z = New class
   z.lptr = z.rptr = null
5 z = New class_of_z
   z.lptr = z.rptr = null
6 y = y.lptr
   y.lptr = y.rptr = null
7 z.sum = x.data + y.data
   x = y = z = null
While loop is not executed even once
```
Our Solution

1. $y = z = \text{null}$
2. $w = x$
3. $w = \text{null}$
4. $\text{while } (x.\text{data} < \text{max})$
   1. $x.lptr = \text{null}$
   2. $x = x.rptr$
5. $x.rptr = x.lptr.rptr = \text{null}$
6. $x.lptr.lptr = x.lptr.rptr = \text{null}$
7. $y = x.lptr$
8. $y.lptr = y.rptr = \text{null}$
9. $y.lptr.lptr = y.lptr.rptr = \text{null}$
10. $z = \text{New class of } z$
11. $z.lptr = z.rptr = \text{null}$
12. $y = y.lptr$
13. $y.lptr = y.rptr = \text{null}$
14. $z.\text{sum} = x.\text{data} + y.\text{data}$
15. $x = y = z = \text{null}$

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

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

Node $i$ is live but link $a \rightarrow i$ is nullified
Some Observations

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

```plaintext
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.lptr = y.lptr.rptr = null
  z = New class of z
  5 z.lptr = z.rptr = null
  y = y.lptr
  6 y.lptr = y.rptr = null
  z.sum = x.data + y.data
  7 x = y = z = null
```

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

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

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

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

1. \( y = z = \text{null} \)
2. \( w = x \)
   \( w = \text{null} \)
3. \begin{align*}
   & \text{while} \ (x.\text{data} < \text{max}) \\
   & \{ \quad x.\text{lptr} = \text{null} \\
   & \quad x = x.\text{rptr} \}
\end{align*}
4. \( x.\text{rptr} = x.\text{lptr}.\text{rptr} = \text{null} \)
5. \( x.\text{lptr}.\text{lptr}.\text{lptr} = \text{null} \)
6. \( x.\text{lptr}.\text{lptr}.\text{rptr} = \text{null} \)
7. \( y = y.\text{lptr} \)
8. \( x.\text{lptr} = y.\text{rptr} = \text{null} \)
9. \( y.\text{lptr}.\text{lptr} = y.\text{lptr}.\text{rptr} = \text{null} \)
10. \( z = \text{New class of } z \)
11. \( z.\text{lptr} = z.\text{rptr} = \text{null} \)
12. \( y = y.\text{lptr} \)
13. \( y.\text{lptr} = y.\text{rptr} = \text{null} \)
14. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
15. \( x = y = z = \text{null} \)
Some Observations

- The memory address that \( x \) holds when the execution reaches a given program point is not an invariant of program execution.
- Whether we dereference \( \text{lptr} \) out of \( x \) or \( \text{rptr} \) out of \( x \) at a given program point is an invariant of program execution.
- A static analysis can discover only some invariants.
BTW, What is Static Analysis of Heap?

Static

Dynamic
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Dynamic

Program Execution
BTW, What is Static Analysis of Heap?

Static

Program Code

Dynamic

Program Execution

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Heap Memory

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Summary Heap Data

Dynamic

Program Execution

Heap Memory

Heap Memory

Heap Memory
BTW, What is Static Analysis of Heap?

- Abstract, Bounded, Single Instance
- Concrete, Unbounded, Infinitely Many

Static

Program Code

Summary Heap Data

Dynamic

Profiling

Program Execution

Heap Memory
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Static Analysis

Summary Heap Data

Dynamic

Program Execution

Heap Memory

?
Part 4

Course Details
The Main Theme of the Course

Constructing *suitable abstractions* for *sound & precise modelling* of *runtime behaviour* of programs efficiently
The Main Theme of the Course

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

---

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

- Program Code

Static Analysis

Summary Information

Dynamic

Program Execution

Memory

Memory

Memory
Sequence of Generalizations in the Course Modules

Bit Vector Frameworks
Sequence of Generalizations in the Course Modules

- Bit Vector Frameworks
- Theoretical abstractions
Sequence of Generalizations in the Course Modules

- Theoretical abstractions
  - General frameworks
    - Bit Vector Frameworks
      - Theoretical abstractions
Sequence of Generalizations in the Course Modules

Intraprocedural Level

Bit Vector Frameworks

General frameworks

Theoretical abstractions
Sequence of Generalizations in the Course Modules

Intraprocedural Level

General frameworks

Interprocedural Level

Bit Vector Frameworks

Theoretical abstractions

July 2017
Course Pedagogy

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

- Tentative plan

<table>
<thead>
<tr>
<th>Assessment</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mid Semester Examination</td>
<td>30%</td>
</tr>
<tr>
<td>End Semester Examination</td>
<td>40%</td>
</tr>
<tr>
<td>Two Quizzes</td>
<td>10%</td>
</tr>
<tr>
<td>Project</td>
<td>20%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>

- Can be fine tuned based on the class feedback
Course Strength and Selection Criteria

• Unavailability of TAs forces restricting the strength
  Less than 30 is preferable, 40 is tolerable

• Course primarily aimed at M.Tech. 1 students
  Follow up course and MTPs

• If the number is large, selection will be based on a test
  ◦ Separate selection for M.Tech.1 and other students
  ◦ Preference to M.Tech.1 students
  ◦ May allow a reasonable number of audits
    ○ Attending all lectures is sufficient
    ○ No need to appear in examinations or do projects
  ◦ Need to finalize the logistics of the test

July 2017
Questions ??
Part 5

Program Model
Program Representation

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

- Control flow graph representation
  - Nodes represent maximal groups of statements devoid of any control transfer except fall through
  - Edges represent control transfers across basic blocks
  - A unique \textit{Start} node and a unique \textit{End} node
    Every node reachable from \textit{Start}, and \textit{End} reachable from every node

- Initially only intraprocedural programs
  Function calls brought in later
int main()
{
    int a, b, c, n;

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

    a = 4;
    b = 2;
    c = 3;
    n = c*2;

    while (a <= n)
    {
        a = a+1;
    }

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

1. a = 4
2. b = 2
3. c = 3
4. n = c*2
5. if (!(a\leq n)) goto 8
6. a = a + 1
7. goto 5
8. if (!(a<12)) goto 11
9. t1 = a+b
10. a = t1+c
11. return a
Part 6

Soundness and Precision
Example Program

```
int a;
int f(int b)
{
    int c;
    c = a%2;
    b = - abs(b);
    while (b < c)
    {
        b = b+1;
        if (b > 0)
        {
            b = 0;
            return b;
        }
    }
}
```
Soundness and Precision of Static Analysis

Example Program

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

Control Flow Graph
Soundness and Precision of Static Analysis

Example Program

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

Control Flow Graph

1. `c = a%2
2. b = -abs(b)
3. if (b < c)
4. if (b > 0)
5. b = 0
6. return b
Execution Traces for Concrete Semantics (1)

• States
  ▶ A data state: Variables → Values
  ▶ A program state: (Program Point, A data state)

• Execution traces (or traces, for short)
  ▶ Valid sequences of program states starting with a given initial state
Execution Traces for Concrete Semantics (2)

1. \[ c = a \% 2 \\ b = - \text{abs}(b) \]

2. if \( b < c \)

3. \[ b = b + 1 \]

4. if \( b > 0 \)

5. \[ b = 0 \]

6. return \( b \)
Execution Traces for Concrete Semantics (2)

1. \( c = a \mod 2 \)
   \( b = -\text{abs}(b) \)

2. if \( b < c \)

   - T
     - Entry 2, (5, \(-2\), 1)
     - Entry 3, (5, \(-2\), 1)
   - F
     - Entry 2, (5, 0, 1)
     - Entry 3, (5, 0, 1)

3. \( b = b + 1 \)

4. if \( b > 0 \)

5. \( b = 0 \)

6. return \( b \)

Trace 1

\[
\begin{array}{ccc}
 a & b & c \\
 Entry_1, (5, 2, 7) \\
 Entry_2, (5, -2, 1) \\
 Entry_3, (5, -2, 1) \\
 Entry_2, (5, -1, 1) \\
 Entry_3, (5, -1, 1) \\
 Entry_2, (5, 0, 1) \\
 Entry_3, (5, 0, 1) \\
 Entry_2, (5, 1, 1) \\
 Entry_4, (5, 1, 1) \\
 Entry_5, (5, 1, 1) \\
 Entry_6, (5, 0, 1) \\
\end{array}
\]
Execution Traces for Concrete Semantics (2)

1. \( c = a \% 2 \)
2. \( b = - \text{abs}(b) \)
3. \( b = b + 1 \)
4. \( \text{if } (b > 0) \)
5. \( b = 0 \)
6. \( \text{return } b \)

**Trace 1**

<table>
<thead>
<tr>
<th>Entry</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry_1</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Entry_2</td>
<td>5</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>Entry_3</td>
<td>5</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>Entry_2</td>
<td>5</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>Entry_3</td>
<td>5</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>Entry_2</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Entry_3</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Entry_2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Entry_4</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Entry_5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Entry_6</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Trace 2**

<table>
<thead>
<tr>
<th>Entry</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry_1</td>
<td>-5</td>
<td>-2</td>
<td>8</td>
</tr>
<tr>
<td>Entry_2</td>
<td>-5</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>Entry_3</td>
<td>-5</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>Entry_2</td>
<td>-5</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Entry_3</td>
<td>-5</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Entry_4</td>
<td>-5</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Entry_6</td>
<td>-5</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Execution Traces for Concrete Semantics (2)

- A separate trace for each combination of inputs
  - The number of traces is potentially infinite
- Program points may repeat in the traces
  - Traces may be very long
  - Non-terminating traces: Infinitely long

```
c = a % 2
b = -abs(b)
1
if (b < c)
2
b = b + 1
3
if (b > 0)
4
b = 0
5
return b
6
```

Trace 1
- Entry 1, (5, 2, 7)
- Entry 2, (5, -2, 1)
- Entry 3, (5, -2, 1)
- Entry 2, (5, -1, 1)
- Entry 3, (5, -1, 1)
- Entry 2, (5, 0, 1)
- Entry 3, (5, 0, 1)
- Entry 2, (5, 1, 1)
- Entry 4, (5, 1, 1)
- Entry 5, (5, 1, 1)
- Entry 6, (5, 0, 1)

Trace 2
- Entry 1, (-5, -2, 8)
- Entry 2, (-5, -2, -1)
- Entry 3, (-5, -2, -1)
- Entry 2, (-5, -1, -1)
- Entry 4, (-5, -1, -1)
- Entry 6, (-5, -1, -1)
Static Analysis Computes Abstractions of Traces (1)
Static Analysis Computes Abstractions of Traces (1)

Traces

An Abstraction of Traces

Execution Time
Static Analysis Computes Abstractions of Traces (1)
Static Analysis Computes Abstractions of Traces (1)

Traces

An Abstraction of Traces

For compile time modelling of possible runtime behaviours of a program

- compute a set of states that cover all traces
- associate the sets with appropriate program points

States may be defined in terms of properties derived from values of variables
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

**Trace 1**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry₁</td>
<td>(5, 2, 7)</td>
</tr>
<tr>
<td>Entry₂</td>
<td>(5, -2, 1)</td>
</tr>
<tr>
<td>Entry₃</td>
<td>(5, -2, 1)</td>
</tr>
<tr>
<td>Entry₄</td>
<td>(5, 0, 1)</td>
</tr>
<tr>
<td>Entry₅</td>
<td>(5, 1, 1)</td>
</tr>
<tr>
<td>Entry₆</td>
<td>(5, 0, 1)</td>
</tr>
</tbody>
</table>

**Trace 2**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry₁</td>
<td>(-5, -2, 8)</td>
</tr>
<tr>
<td>Entry₂</td>
<td>(-5, -2, -1)</td>
</tr>
<tr>
<td>Entry₃</td>
<td>(-5, -2, -1)</td>
</tr>
<tr>
<td>Entry₄</td>
<td>(-5, -1, -1)</td>
</tr>
<tr>
<td>Entry₅</td>
<td>(-5, -1, -1)</td>
</tr>
</tbody>
</table>

```python
c = a % 2
b = - abs(b)

if (b < c):
    b = b + 1

if (b > 0):
    b = 0

return b
```
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

{(5, 2, 7), (−5, −2, 8)}

Trace 1
a b c
Entry_1, (5, 2, 7)
Entry_2, (5, −2, 1)
Entry_3, (5, −2, 1)
Entry_2, (5, −1, 1)
Entry_3, (5, −1, 1)
Entry_2, (5, 0, 1)
Entry_3, (5, 0, 1)
Entry_2, (5, 1, 1)
Entry_4, (5, 1, 1)
Entry_5, (5, 1, 1)
Entry_6, (5, 0, 1)

Trace 2
a b c
Entry_1, (−5, −2, 8)
Entry_2, (−5, −2, −1)
Entry_3, (−5, −2, −1)
Entry_2, (−5, −1, −1)
Entry_2, (−5, −1, −1)
Entry_4, (−5, −1, −1)
Entry_6, (−5, −1, −1)

c = a%2
b = − abs(b)

if (b < c)

b = b+1

if (b > 0)

b = 0

return b
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

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

Trace 1
\[ a \ b \ c \]
Entry_1, (5, 2, 7)
Entry_2, (5, -2, 1)
Entry_3, (5, -2, 1)
Entry_2, (5, -1, 1)
Entry_3, (5, -1, 1)
Entry_2, (5, 0, 1)
Entry_3, (5, 0, 1)
Entry_4, (5, 1, 1)
Entry_5, (5, 1, 1)
Entry_6, (5, 0, 1)

Trace 2
\[ a \ b \ c \]
Entry_1, (-5, -2, 8)
Entry_2, (-5, -2, -1)
Entry_3, (-5, -2, -1)
Entry_2, (-5, -1, -1)
Entry_4, (-5, -1, -1)
Entry_6, (-5, -1, -1)

1. \[ c = a \% 2 \]
   \[ b = - \text{abs}(b) \]

2. if (b < c)

3. \[ b = b + 1 \]

4. if (b > 0)

5. \[ b = 0 \]

6. return b

A possible static abstraction using sets

\[ a = \{-5, 5\}, \ b = \{-2, 2\}, \ c = \{7, 8\} \]
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

\[ a = \{-5, 5\}, \quad b = \{-2, 2\}, \quad c = \{7, 8\} \]

We only show the values of \( b \)

Combine the values across all occurrences of a program point

\[ c = a \div 2 \]
\[ b = - \text{abs}(b) \]

1. \( c = a \div 2 \)
   \[ b = - \text{abs}(b) \]

2. \( \text{if } (b < c) \)

3. \( \text{if } (b > 0) \)
   \[ b = b + 1 \]

4. \( \text{if } (b > 0) \)

5. \( b = 0 \)

return \( b \)
A possible static abstraction using sets

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

\[ c = a \% 2 \]
\[ b = - \text{abs}(b) \]

1. \[ c = a \% 2 \]
\[ b = - \text{abs}(b) \]

2. \[ \text{if} \ (b < c) \]
\[ \text{else} \]

3. \[ b = b + 1 \]

4. \[ \text{if} \ (b > 0) \]
\[ \text{else} \]

5. \[ b = 0 \]

We only show the values of \( b \)

Trace 1

\( a \ b \ c \)

Entry\(_1\), (5, 2, 7)
Entry\(_2\), (5, -2, 1)
Entry\(_3\), (5, -2, 1)
Entry\(_2\), (5, -1, 1)
Entry\(_3\), (5, -1, 1)
Entry\(_2\), (5, 0, 1)
Entry\(_3\), (5, 0, 1)
Entry\(_2\), (5, 1, 1)
Entry\(_4\), (5, 1, 1)
Entry\(_5\), (5, 1, 1)
Entry\(_6\), (5, 0, 1)

Trace 2

\( a \ b \ c \)

Entry\(_1\), (-5, -2, 0)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Combine the values across all occurrences of a program point
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

\[ a = \{-5, 5\}, \quad b = \{-2, 2\}, \quad c = \{7, 8\} \]

We only show the values of \( b \)

Combine the values across all occurrences of a program point

Entry\(_1\), (5, 2, 7)
Entry\(_2\), (5, -2, 1)
Entry\(_3\), (5, -2, 1)
Entry\(_2\), (5, -1, 1)
Entry\(_3\), (5, -1, 1)
Entry\(_2\), (5, 0, 1)
Entry\(_3\), (5, 0, 1)
Entry\(_2\), (5, 1, 1)
Entry\(_4\), (5, 1, 1)
Entry\(_5\), (5, 1, 1)
Entry\(_6\), (5, 0, 1)

Entry\(_1\), (-5, -2, 8)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 7)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 8)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 1)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 8)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 7)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 1)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 8)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 7)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 1)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 8)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 7)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 1)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 8)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 7)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 1)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 8)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 7)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)

Entry\(_1\), (-5, -2, 1)
Entry\(_2\), (-5, -2, -1)
Entry\(_3\), (-5, -2, -1)
Entry\(_2\), (-5, -1, -1)
Entry\(_4\), (-5, -1, -1)
Entry\(_6\), (-5, -1, -1)
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

\[
\begin{align*}
 a &= \{-5, 5\}, \\
b &= \{-2, 2\}, \\
c &= \{7, 8\}
\end{align*}
\]

\[
\begin{align*}
\text{Entry}_1, (5, 2, 7) \\
\text{Entry}_2, (5, -2, 1) \\
\text{Entry}_3, (5, -2, 1) \\
\text{Entry}_2, (5, -1, 1) \\
\text{Entry}_3, (5, -1, 1) \\
\text{Entry}_2, (5, 0, 1) \\
\text{Entry}_3, (5, 0, 1) \\
\text{Entry}_2, (5, 1, 1) \\
\text{Entry}_4, (5, 1, 1) \\
\text{Entry}_5, (5, 1, 1) \\
\text{Entry}_6, (5, 0, 1)
\end{align*}
\]

We only show the values of b

Combine the values across all occurrences of a program point

\[
\begin{align*}
\text{Entry}_1, (-5, -2, 8) \\
\text{Entry}_2, (-5, -2, -1) \\
\text{Entry}_3, (-5, -2, -1) \\
\text{Entry}_2, (-5, -1, -1) \\
\text{Entry}_4, (-5, -1, -1) \\
\text{Entry}_6, (-5, -1, -1)
\end{align*}
\]
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

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

Trace 1

\[
\begin{array}{l}
Entry_1, (5, 2, 7) \\
Entry_2, (5, -2, 1) \\
Entry_3, (5, -2, 1) \\
Entry_2, (5, -1, 1) \\
Entry_3, (5, -1, 1) \\
Entry_2, (5, 0, 1) \\
Entry_3, (5, 0, 1) \\
Entry_2, (5, 1, 1) \\
Entry_4, (5, 1, 1) \\
Entry_5, (5, 1, 1) \\
Entry_6, (5, 0, 1)
\end{array}
\]

We only show the values of b

We only show the values of b

Trace 2

\[
\begin{array}{l}
Entry_1, (-5, -2, 8) \\
Entry_2, (-5, -2, -1) \\
Entry_3, (-5, -2, -1) \\
Entry_2, (-5, -1, -1) \\
Entry_4, (-5, -1, -1) \\
Entry_6, (-5, -1, -1)
\end{array}
\]

Combine the values across all occurrences of a program point

Combine the values across all occurrences of a program point
Computing Static Abstraction for Liveness of Variables

At a program point \( p \)
\( a \mapsto 1 \Rightarrow a \) is live at \( p \)
\( a \mapsto 0 \Rightarrow a \) is not live at \( p \)

Trace 1
\[
\begin{array}{llll}
\text{Entry}_1, & (1, 1, 0) \\
\text{Entry}_2, & (0, 1, 1) \\
\text{Entry}_3, & (0, 1, 1) \\
\text{Entry}_2, & (0, 1, 1) \\
\text{Entry}_3, & (0, 1, 1) \\
\text{Entry}_2, & (0, 1, 1) \\
\text{Entry}_3, & (0, 1, 1) \\
\text{Entry}_2, & (0, 1, 1) \\
\text{Entry}_4, & (0, 1, 0) \\
\text{Entry}_5, & (0, 0, 0) \\
\text{Entry}_6, & (0, 1, 0) \\
\end{array}
\]

Trace 2
\[
\begin{array}{llll}
\text{Entry}_1, & (1, 1, 0) \\
\text{Entry}_2, & (0, 1, 1) \\
\text{Entry}_3, & (0, 1, 1) \\
\text{Entry}_2, & (0, 1, 1) \\
\text{Entry}_3, & (0, 0, 1) \\
\text{Entry}_2, & (0, 1, 1) \\
\text{Entry}_4, & (0, 1, 0) \\
\text{Entry}_6, & (0, 1, 0) \\
\end{array}
\]

Program Code:
\[
c = a \% 2 \\
b = -\text{abs}(b)
\]

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

if \( b < c \)
3. \( b = b + 1 \)
4. \( b = b > 0 \)
5. \( b = 0 \)
6. return \( b \)
Computing Static Abstraction for Liveness of Variables

At a program point $p$

$a \mapsto 1 \Rightarrow a$ is live at $p$

$a \mapsto 0 \Rightarrow a$ is not live at $p$

**Trace 1**

| Entry 1 | (1,1,0) |
| Entry 2 | (0,1,1) |
| Entry 3 | (0,1,1) |
| Entry 2 | (0,1,1) |
| Entry 3 | (0,0,1) |
| Entry 2 | (0,1,1) |
| Entry 4 | (0,1,0) |
| Entry 5 | (0,0,0) |
| Entry 6 | (0,1,0) |

**Trace 2**

| Entry 1 | (1,1,0) |
| Entry 2 | (0,1,1) |
| Entry 3 | (0,1,1) |
| Entry 2 | (0,1,1) |
| Entry 3 | (0,0,1) |
| Entry 2 | (0,1,1) |
| Entry 4 | (0,1,0) |
| Entry 6 | (0,1,0) |

$c = a \% 2$

$b = -\text{abs}(b)$

1. if ($b < c$)
   - 2. $b = b + 1$
   - 3. $b = 0$
   - 4. if ($b > 0$)
   - 5. return $b$
Computing Static Abstraction for Liveness of Variables

At a program point $p$

$a \mapsto 1 \Rightarrow a$ is live at $p$

$a \mapsto 0 \Rightarrow a$ is not live at $p$

**Trace 1**

```
<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry_1</td>
<td>(1, 1, 0)</td>
</tr>
<tr>
<td>Entry_2</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>Entry_3</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>Entry_4</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>Entry_5</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>Entry_6</td>
<td>(0, 1, 0)</td>
</tr>
</tbody>
</table>
```

**Trace 2**

```
<table>
<thead>
<tr>
<th>Entry</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry_1</td>
<td>(1, 1, 0)</td>
</tr>
<tr>
<td>Entry_2</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>Entry_3</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>Entry_4</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>Entry_5</td>
<td>(0, 1, 1)</td>
</tr>
<tr>
<td>Entry_6</td>
<td>(0, 1, 0)</td>
</tr>
</tbody>
</table>
```

110 or $\{a, b\}$

```
c = a\%2
b = -abs(b)
```

011 or $\{b, c\}$

```
if (b < c)
T F 110 or $\{a, b\}$
```

```
if (b > 0)
F T 011 or $\{b, c\}$
```

```
return b
```

1

```
c = a\%2
b = -abs(b)
```

2

```
if (b < c)
T F 110 or $\{a, b\}$
```

```
if (b > 0)
F T 011 or $\{b, c\}$
```

```
return b
```

3

```
if (b > 0)
F T 011 or $\{b, c\}$
```

```
if (b > 0)
F T 011 or $\{b, c\}$
```

```
return b
```

4

```
if (b > 0)
F T 011 or $\{b, c\}$
```

```
if (b > 0)
F T 011 or $\{b, c\}$
```

```
return b
```

5

```
if (b > 0)
F T 011 or $\{b, c\}$
```

```
if (b > 0)
F T 011 or $\{b, c\}$
```

```
return b
```

6

```
return b
```

July 2017
Computing Static Abstraction for Liveness of Variables

At a program point $p$

- $a \mapsto 1 \Rightarrow a$ is live at $p$
- $a \mapsto 0 \Rightarrow a$ is not live at $p$

**Trace 1**

- $a$  $b$  $c$
- Entry$_1$, (1, 1, 0)
- Entry$_2$, (0, 1, 1)
- Entry$_3$, (0, 1, 1)
- Entry$_2$, (0, 1, 1)
- Entry$_3$, (0, 1, 1)
- Entry$_2$, (0, 1, 1)
- Entry$_3$, (0, 1, 1)
- Entry$_2$, (0, 1, 1)
- Entry$_4$, (0, 1, 1)
- Entry$_6$, (0, 1, 0)
- Entry$_5$, (0, 0, 0)
- Entry$_6$, (0, 1, 0)

**Trace 2**

- $a$  $b$  $c$
- Entry$_1$, (1, 1, 0)
- Entry$_2$, (0, 1, 1)
- Entry$_3$, (0, 0, 1)
- Entry$_2$, (0, 1, 1)
- Entry$_4$, (0, 1, 0)
- Entry$_6$, (0, 1, 0)

**Code:**

1. $c = a \% 2$
2. $b = - \text{abs}(b)$
3. if $(b < c)$
   - 011 or $\{b, c\}$
4. if $(b > 0)$
   - 110 or $\{a, b\}$
5. $b = b + 1$
6. $b = 0$
7. return $b$

**Comment:**

- At a program point $p$
  - $a \mapsto 1 \Rightarrow a$ is live at $p$
  - $a \mapsto 0 \Rightarrow a$ is not live at $p$
Computing Static Abstraction for Liveness of Variables

At a program point $p$

- $a \mapsto 1 \Rightarrow a$ is live at $p$
- $a \mapsto 0 \Rightarrow a$ is not live at $p$

**Trace 1**

- $a \ b \ c$
- $Entry_1, (1, 1, 0)$
- $Entry_2, (0, 1, 1)$
- $Entry_3, (0, 1, 1)$
- $Entry_2, (0, 1, 1)$
- $Entry_3, (0, 1, 1)$
- $Entry_2, (0, 1, 1)$
- $Entry_3, (0, 1, 1)$
- $Entry_2, (0, 1, 1)$
- $Entry_4, (0, 1, 0)$
- $Entry_5, (0, 0, 0)$
- $Entry_6, (0, 1, 0)$

**Trace 2**

- $a \ b \ c$
- $Entry_1, (1, 1, 0)$
- $Entry_2, (0, 1, 1)$
- $Entry_3, (0, 0, 1)$
- $Entry_2, (0, 1, 1)$
- $Entry_3, (0, 1, 1)$
- $Entry_2, (0, 1, 0)$
- $Entry_6, (0, 1, 0)$

1. $c = a \% 2$
2. $b = -\text{abs}(b)$
3. 011 or $\{b, c\}$
4. 010 or $\{b\}$
5. $b = 0$
6. return $b$
Computing Static Abstraction for Liveness of Variables

At a program point $p$

$a \mapsto 1 \Rightarrow a$ is live at $p$

$a \mapsto 0 \Rightarrow a$ is not live at $p$

**Trace 1**

$a \ b \ c$

- **Entry**$_1$, (1, 1, 0)
- **Entry**$_2$, (0, 1, 1)
- **Entry**$_3$, (0, 1, 1)
- **Entry**$_2$, (0, 1, 1)
- **Entry**$_3$, (0, 1, 1)
- **Entry**$_2$, (0, 1, 1)
- **Entry**$_3$, (0, 1, 1)
- **Entry**$_2$, (0, 1, 1)
- **Entry**$_4$, (0, 1, 0)
- **Entry**$_5$, (0, 0, 0)
- **Entry**$_6$, (0, 1, 0)

**Trace 2**

- **Entry**$_1$, (1, 1, 0)
- **Entry**$_2$, (0, 1, 1)
- **Entry**$_3$, (0, 0, 1)
- **Entry**$_2$, (0, 1, 1)
- **Entry**$_2$, (0, 1, 0)
- **Entry**$_6$, (0, 1, 0)

$c = a \% 2$

$b = -\text{abs}(b)$

110 or $\{a, b\}$

011 or $\{b, c\}$

010 or $\{b\}$

011 or $\{b, c\}$

001 or $\{c\}$

000 or $\emptyset$

1. $c = a \% 2$

2. $b = -\text{abs}(b)$

3. if $(b < c)$

4. if $(b > 0)$

5. $b = 0$

6. return $b$
Computing Static Abstraction for Liveness of Variables

At a program point \( p \)
\( a \mapsto 1 \Rightarrow a \) is live at \( p \)
\( a \mapsto 0 \Rightarrow a \) is not live at \( p \)

**Trace 1**
\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
Entry_1, (1,1,0) & \\
Entry_2, (0,1,1) & \\
Entry_3, (0,1,1) & \\
Entry_2, (0,1,1) & \\
Entry_3, (0,1,1) & \\
Entry_2, (0,1,1) & \\
Entry_3, (0,1,1) & \\
Entry_2, (0,1,1) & \\
Entry_4, (0,1,0) & \\
Entry_5, (0,0,0) & \\
Entry_6, (0,1,0) & \\
\end{array}
\]

**Trace 2**
\[
\begin{array}{ccc}
\text{a} & \text{b} & \text{c} \\
Entry_1, (1,1,0) & \\
Entry_2, (0,1,1) & \\
Entry_3, (0,0,1) & \\
Entry_2, (0,1,1) & \\
Entry_4, (0,1,0) & \\
Entry_6, (0,1,0) & \\
\end{array}
\]

\[
\begin{align*}
1 & \quad c = a \% 2 \\
2 & \quad \text{if } (b < c) \\
3 & \quad b = b + 1 \\
4 & \quad \text{if } (b > 0) \\
5 & \quad b = 0 \\
6 & \quad \text{return } b
\end{align*}
\]

\[
\begin{align*}
010 & \quad \{a, b\} \\
000 & \quad \emptyset \\
011 & \quad \{b, c\} \\
010 & \quad \{b\} \\
110 & \quad \{a, b\} \\
\end{align*}
\]
Computing Static Abstraction for Liveness of Variables

At a program point $p$

- $a \mapsto 1 \Rightarrow a$ is live at $p$
- $a \mapsto 0 \Rightarrow a$ is not live at $p$

**Trace 1**

- $a \ b \ c$
- $Entry_1, (1, 1, 0)$
- $Entry_2, (0, 1, 1)$
- $Entry_3, (0, 1, 1)$
- $Entry_2, (0, 1, 1)$
- $Entry_3, (0, 1, 1)$
- $Entry_2, (0, 1, 1)$
- $Entry_3, (0, 1, 1)$
- $Entry_2, (0, 1, 1)$
- $Entry_4, (0, 1, 0)$
- $Entry_6, (0, 1, 0)$

**Trace 2**

- $a \ b \ c$
- $Entry_1, (1, 1, 0)$
- $Entry_2, (0, 1, 1)$
- $Entry_3, (0, 0, 1)$
- $Entry_2, (0, 1, 1)$
- $Entry_4, (0, 1, 0)$
- $Entry_6, (0, 1, 0)$

**Code**

```plaintext
110 or \{a, b\}
1
\[c = a \% 2\]
\[b = -\text{abs}(b)\]

011 or \{b, c\}
2
if (b < c)

010 or \{b\}
4
if (b > 0)

011 or \{b, c\}
3
b = b+1

000 or ∅
5
b = 0

010 or \{b\}
6
return b
```

Trace 2 does not add anything to the abstraction
Soundness of Abstractions (1)

- An over-approximation of traces is sound
Soundness of Abstractions (1)

Sound

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

Unsound
Soundness of Abstractions (2)

An unsound abstraction

\[ a = \{-5, 5\}, \quad b = \{-2, 2\}, \quad c = \{7, 8\} \]

1. \[ c = a \% 2 \]
   \[ b = -\text{abs}(b) \]

2. if \( b < c \)

3. \[ b = b + 1 \]

4. if \( b > 0 \)

5. \[ b = 0 \]

6. return \( b \)

All variables can have arbitrary values at the start.

\( b \) can have many more values at the entry of
- blocks 2 and 3 (e.g. -3, -8, \ldots)
- block 4 (e.g. 0)
Soundness of Abstractions (2)

An unsound abstraction

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

1. \[ c = a \% 2 \]
   \[ b = -\text{abs}(b) \]

2. if \( b < c \)
   \[ b = \{-2, -1, 0, 1\} \]

3. \[ b = b + 1 \]
   \[ b = \{-1, 1\} \]

4. if \( b > 0 \)
   \[ b = \{-1, 1\} \]

5. \[ b = 0 \]
   \[ b = \{-1, 0\} \]

6. return \( b \)

A sound abstraction using intervals

- Overapproximated range of values denoted by \([\text{low\_limit}, \text{high\_limit}]\)
- Inclusive limits with \(\text{low\_limit} \leq \text{high\_limit}\)
- One continuous range per variable with no "holes"
An unsound abstraction

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

1. \[ c = a \% 2 \]
   \[ b = -\text{abs}(b) \]

\[ b = \{-2, -1, 0, 1\} \]

2. if \( b < c \)
   \[ b = b + 1 \]

\[ b = \{-1, 1\} \]

4. if \( b > 0 \)
   \[ b = 0 \]

\[ b = \{-1, 0\} \]

6. return \( b \)

A sound abstraction using intervals

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

1. \[ c = a \% 2 \]
   \[ b = -\text{abs}(b) \]

\[ b = [-\infty, 1] \]

2. if \( b < c \)
   \[ b = b + 1 \]

\[ b = [-1, 1] \]

4. if \( b > 0 \)
   \[ b = 0 \]

\[ b = [-1, 0] \]

6. return \( b \)

July 2017
Soundness of Abstractions (2)

An unsound abstraction

\[ a = \{-5, 5\}, \quad b = \{-2, 2\}, \quad c = \{7, 8\} \]

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

1
\[
b = \{-2, -1, 0, 1\}
\]

2
\[
\text{if } (b < c)
\]

1
\[
b = \{-1, 1\}
\]

2
\[
\text{if } (b > 0)
\]

1
\[
\text{return } b
\]

A sound abstraction using intervals

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

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

1
\[
b = [-\infty, 1]
\]

2
\[
\text{if } (b < c)
\]

1
\[
b = [-1, 1]
\]

2
\[
\text{if } (b > 0)
\]

1
\[
\text{return } b
\]

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

\[ b = [-2, -1, 0, 1] \]

\[ b = [-2, -1, 0] \]

\[ b = [-1, 1] \]

\[ b = [-1, 0] \]
Soundness of Abstractions for Liveness Analysis

A sound abstraction

1. \( c = a \mod 2 \)
2. \( b = -\text{abs}(b) \)
3. \{b, c\}
4. if (b > 0)
5. \{b\}
6. return b

An unsound abstraction

1. \( c = a \mod 2 \)
2. if (b < c)
3. \{b, c\}
4. if (b > 0)
5. \{b\}
6. return b
Precision of Sound Abstractions (1)

Sound but imprecise
Precision of Sound Abstractions(1)

Sound but imprecise

Sound and more precise
Precision of Sound Abstractions (1)

- Sound but imprecise
- Sound and more precise
- Sound and even more precise
Precision of Sound Abstractions (1)

- Precision is relative, soundness is absolute
- Qualifiers “more precise” and “less precise” are meaningful
- Qualifiers “more sound” and “less sound” are not meaningful
Precision of Sound Abstractions (2)

A precise abstraction using intervals

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

1. \[ c = a \bmod 2 \]
   \[ b = -\text{abs}(b) \]

2. if (b < c)

3. \[ b = b + 1 \]

4. if (b > 0)

5. \[ b = 0 \]

6. return b

An imprecise abstraction using intervals

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

1. \[ c = a \bmod 2 \]
   \[ b = -\text{abs}(b) \]

2. if (b < c)

3. \[ b = b + 1 \]

4. if (b > 0)

5. \[ b = 0 \]

6. return b
Precision of Abstractions for Liveness Analysis

A precise abstraction

\[
\begin{align*}
{a, b} \\
c &= a \% 2 \\
b &= -\text{abs}(b) \\
\{b, c\} \\
\text{if (b < c)} \\
\{b\} \\
\text{if (b > 0)} \\
\emptyset \\
{a, b, c} \\
\text{return b}
\end{align*}
\]

An imprecise abstraction

\[
\begin{align*}
{a, b, c} \\
c &= a \% 2 \\
b &= -\text{abs}(b) \\
\{a, b, c\} \\
\text{if (b < c)} \\
\{a, b, c\} \\
\text{if (b > 0)} \\
\emptyset \\
{a, b, c} \\
\text{return b}
\end{align*}
\]
Limitations of Static Analysis

• In general, the computation of exact static abstraction is undecidable
  ▶ Possible reasons
    ○ Values of variables not known
    ○ Branch outcomes not known
    ○ Infinitely many paths in the presence of loops or recursion
    ○ Infinitely many values
  ▶ We have to settle for some imprecision
  ▶ How are data states compared to distinguish between a sound and unsound (or a precise or an imprecise result)?
    ○ We have introduced the concepts intuitively
    ○ Will define them formally in a later module

• Goodness of a static analysis lies in minimizing imprecision without compromising on soundness

  Additional expectations: Efficiency and scalability