Introduction to Program Analysis

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

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

Dec 2019
Part 1

About These Slides
Copyright

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

  (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
- 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]; // v is the pivot */
    while(1) // Move values smaller */
    {
        do i = i + 1; while (a[i] < v); // than v to the left of */
        do j = j - 1; while (a[j] > v); // the split point (sp) */
        if (i >= j) break; // and other values */
        x = a[i]; a[i] = a[j]; a[j] = x; // to the right of sp */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; // of the split point */
    quicksort(m, i); quicksort(i+1, n); // Move the pivot to sp */
    // sort the partitions to */
} // the left of sp and to the right of sp independently */
```

Dec 2019 IIT Bombay
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
  \[\Rightarrow\] 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

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( i = m - 1 )</td>
</tr>
<tr>
<td>2</td>
<td>( j = n )</td>
</tr>
<tr>
<td>3</td>
<td>( t_1 = 4 \times n )</td>
</tr>
<tr>
<td>4</td>
<td>( t_6 = a[t_1] )</td>
</tr>
<tr>
<td>5</td>
<td>( v = t_6 )</td>
</tr>
<tr>
<td>6</td>
<td>( i = i + 1 )</td>
</tr>
<tr>
<td>7</td>
<td>( t_2 = 4 \times i )</td>
</tr>
<tr>
<td>8</td>
<td>( t_3 = a[t_2] )</td>
</tr>
<tr>
<td>9</td>
<td>if ( t_3 &lt; v ) goto 6</td>
</tr>
<tr>
<td>10</td>
<td>( j = j - 1 )</td>
</tr>
<tr>
<td>11</td>
<td>( t_4 = 4 \times j )</td>
</tr>
<tr>
<td>12</td>
<td>( t_5 = a[t_4] )</td>
</tr>
<tr>
<td>13</td>
<td>if ( t_5 &gt; v ) goto 10</td>
</tr>
<tr>
<td>14</td>
<td>if ( i \geq j ) goto 25</td>
</tr>
<tr>
<td>15</td>
<td>( t_2 = 4 \times i )</td>
</tr>
<tr>
<td>16</td>
<td>( t_3 = a[t_2] )</td>
</tr>
<tr>
<td>17</td>
<td>( x = t_3 )</td>
</tr>
<tr>
<td>18</td>
<td>( t_2 = 4 \times i )</td>
</tr>
<tr>
<td>19</td>
<td>( t_4 = 4 \times j )</td>
</tr>
<tr>
<td>20</td>
<td>( t_5 = a[t_4] )</td>
</tr>
<tr>
<td>21</td>
<td>( a[t_2] = t_5 )</td>
</tr>
<tr>
<td>22</td>
<td>( t_4 = 4 \times j )</td>
</tr>
<tr>
<td>23</td>
<td>( a[t_4] = x )</td>
</tr>
<tr>
<td>24</td>
<td>goto 6</td>
</tr>
<tr>
<td>25</td>
<td>( t_2 = 4 \times i )</td>
</tr>
<tr>
<td>26</td>
<td>( t_3 = a[t_2] )</td>
</tr>
<tr>
<td>27</td>
<td>( x = t_3 )</td>
</tr>
<tr>
<td>28</td>
<td>( t_2 = 4 \times i )</td>
</tr>
<tr>
<td>29</td>
<td>( t_1 = 4 \times n )</td>
</tr>
<tr>
<td>30</td>
<td>( t_6 = a[t_1] )</td>
</tr>
<tr>
<td>31</td>
<td>( a[t_2] = t_6 )</td>
</tr>
<tr>
<td>32</td>
<td>( t_1 = 4 \times n )</td>
</tr>
<tr>
<td>33</td>
<td>( a[t_1] = x )</td>
</tr>
</tbody>
</table>
Program Flow Graph

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

\[
\begin{align*}
\text{if } i &\geq j \text{ goto } B6
\end{align*}
\]

B5

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

\[
i = m - 1 \\
j = n \\
t_1 = 4 \times n \\
t_6 = a[t_1] \\
v = t_6
\]

B2

\[
i = i + 1 \\
t_2 = 4 \times i \\
t_3 = a[t_2] \\
\text{if } t_3 < v \text{ goto B2}
\]

B3

\[
\begin{align*}
&j = j - 1 \\
t_4 = 4 \times j \\
t_5 = a[t_4] \\
\text{if } t_5 > v \text{ goto B3}
\end{align*}
\]

B4

\[
\begin{align*}
\text{if } i \geq j \text{ goto B6}
\end{align*}
\]

B5

\[
\begin{align*}
t_2 &= 4 \times i \\
t_3 &= a[t_2] \\
x &= t_3 \\
t_2 &= 4 \times i \\
t_4 &= 4 \times j \\
t_5 &= a[t_4] \\
a[t_2] &= t_5 \\
t_4 &= 4 \times j \\
a[t_4] &= x \\
\text{goto B2}
\end{align*}
\]
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

\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ x = t3 \]

B6

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

\[
\begin{align*}
    &\text{if } i \geq j \text{ goto B6}
\end{align*}
\]

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

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

B2

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

B3

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

B4

\[ if i \geq j \text{ goto B6} \]
\[ a[t1] = x \]
\[ a[t2] = t6 \]

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

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

B1
- $i = m - 1$
- $j = n$
- $t1 = 4 \times n$
- $v = t6$

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 $i >= j$ goto B6
  - $t2 = 4 \times i$
  - $t3 = a[t2]$
  - $x = t3$
  - $t4 = 4 \times j$
  - $t5 = a[t4]$
  - $a[t2] = t5$
  - $a[t4] = x$
  - goto B2

B5
- $t2 = 4 \times i$
Global Common Subexpression Elimination

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

B2
\[ i = i + 1 \]
\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ t6 = a[t1] \]
\[ v = t6 \]
\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ t6 = a[t1] \]
\[ v = t6 \]

B3
\[ j = j - 1 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ x = t3 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ x = t3 \]

B4
\[ if \ i \geq j \ goto \ B6 \]
\[ a[t2] = t6 \]
\[ a[t4] = x \]
\[ a[t2] = t5 \]
\[ a[t4] = x \]

B5
\[ t2 = 4 \times i \]
\[ t3 = a[t2] \]
\[ x = t3 \]
\[ t4 = 4 \times j \]
\[ t5 = a[t4] \]
\[ a[t2] = t5 \]
\[ a[t2] = t5 \]
Global Common Subexpression Elimination

```
B1
i = m - 1
j = n

t1 = 4 * n

B2
i = i + 1

t2 = 4 * i

t3 = a[t1]
if t3 < v goto B2

B3
j = j - 1

t4 = 4 * j

t5 = a[t4]
if t5 > v goto B3

B4
if i >= j goto B6

B5
x = t3
a[t2] = t5
a[t4] = x

goto B2

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

Dec 2019
Other Classical Optimizations

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

\[
\begin{align*}
\text{B1} & \quad i &= m - 1 \\
& \quad j &= n \\
& \quad t1 &= 4 \times n \\
& \quad t6 &= a[t1] \\
& \quad v &= t6 \\
\text{B2} & \quad i &= i + 1 \\
& \quad t2 &= 4 \times i \\
& \quad t3 &= a[t2] \\
& \quad \text{if } t3 < v \text{ goto B2} \\
\text{B3} & \quad j &= j - 1 \\
& \quad t4 &= 4 \times j \\
& \quad t5 &= a[t4] \\
& \quad \text{if } t5 > v \text{ goto B3} \\
\text{B4} & \quad \text{if } i \geq j \text{ goto B6} \\
\text{B5} & \quad x &= t3 \\
& \quad a[t2] &= t5 \\
& \quad a[t4] &= x \\
& \quad \text{goto B2} \\
\end{align*}
\]
Copy Propagation and Dead Code Elimination

B1

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

B2

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

B2

\[
\text{if t3 < v goto B2}
\]

B3

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

B4

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

B5

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

B6

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

Dec 2019
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

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

Dec 2019
Strength Reduction and Induction Variable 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

\[
\begin{align*}
\text{if } i \geq j & \text{ goto B6}
\end{align*}
\]

B5

\[
\begin{align*}
a[t2] &= t5 \\
a[t4] &= t3 \\
goto B2
\end{align*}
\]

B6

\[
\begin{align*}
t6 &= a[t1] \\
a[t2] &= t6 \\
a[t1] &= t3
\end{align*}
\]
**Strength Reduction and Induction Variable Elimination**

B1

\[ i = m - 1 \]
\[ j = n \]
\[ t1 = 4 \times n \]
\[ t6 = a[t1] \]
\[ v = t6 \]
\[ t2 = 4 \times i \]
\[ t4 = 4 \times j \]

B2

- \[ t2 = t2 + 4 \]
- \[ t3 = a[t2] \]
- If \( t3 < v \) goto B2

B3

- \[ t4 = t4 - 4 \]
- \[ t5 = a[t4] \]
- If \( t5 > v \) goto B3

B4

- If \( t2 \geq t4 \) goto B6

B5

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

B6

- \[ t6 = a[t1] \]
- \[ a[t2] = t6 \]
- \[ a[t1] = t3 \]
Final Intermediate Code

```
B1
i = m - 1
j = n
t1 = 4 * n
t6 = a[t1]
v = t6
B2
B3
t2 = t2 + 4
t3 = a[t2]
t4 = t4 - 4
t5 = a[t4]
if t3 < v goto B2
if t5 > v goto B3
B4
if t2 >= t4 goto B6
B5
B6
t6 = a[t1]
a[t2] = t6
a[t1] = t3
a[t2] = t5
a[t4] = t3
goto B2
```

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>
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 Deallocation?

- **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 $\equiv$ 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?
We use Java style statements for convenience
Read “x.lptr” as “x→lptr

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

Dec 2019

IIT Bombay
What is Garbage?

The blue nodes will be used after statement 4.

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

The blue nodes will be used after statement 4

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

The blue nodes will be used after statement 4

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. \( z.\text{sum} = x.data + y.data \)
8. return \( z.\text{sum} \)

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

From www.memorymanagement.org/glossary

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

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

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

• Not really. Most of us know that.

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

• The state of art compilers, virtual machines, garbage collectors cannot distinguish between the two
Reachability and Liveness

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

For the heap memory on the right:

- **Allocated**: White + Blue + Brown nodes
- **Reachable**: White + Blue nodes
- **Live**: Blue nodes
Reachability and Liveness

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

For the heap memory on the right:

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

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

Hence, \(\neg\text{Live} \supseteq \neg\text{Reachable} \supseteq \neg\text{Allocated}\)
Cedar Mesa Folk Wisdom

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

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

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

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

The state of art

• Eliminating objects reachable from root variables which are not live.

• Implemented in current Sun JVMs.

• Uses liveness data flow analysis of root variables (stack data).

• What about liveness of heap data?
Liveness of Stack Data: An Informal Introduction (1)

We use Java style statements for convenience
Read “x.lptr” as “x→lptr

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

if changed to while
Liveness of Stack Data: An Informal Introduction (1)

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. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
8. \( \text{return } z.\text{sum} \)

What is the meaning of the use of data?
Liveness of Stack Data: An Informal Introduction (1)

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

What is the meaning of the use of data?

Dec 2019
Liveness of Stack Data: An Informal Introduction (1)

What is the meaning of the use of data?

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
8 return z.sum
Liveness of Stack Data: An Informal Introduction (1)

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

Accessing the location and reading its contents

Reading x (Stack data)

Heap

Stack
Liveness of Stack Data: An Informal Introduction (1)

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. \( z.sum = x.data + y.data \)
8. return \( z.sum \)

Accessing the location and reading its contents

Heap

Stack

Reading \( x.data \) (Heap data)
Liveness of Stack Data: An Informal Introduction (1)

1. `w = x`   // `x` points to `m_a`
2. `while (x.data < MAX)`
3. `x = x.rptr`
4. `y = x.lptr`
5. `z = New class_of_z`
6. `y = y.lptr`
7. `z.sum = x.data + y.data`
8. `return z.sum`
Liveness of Stack Data: An Informal Introduction (2)

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

No variable is used beyond this program point
Liveness of Stack Data: An Informal Introduction (2)

```
w = x

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

y = x.lptr

z = New class_of_z

y = y.lptr

z.sum = x.data + y.data

return z.sum
```

Current value of z is used beyond this program point

Live

Dead
w = x

while (x.data < MAX)

x = x.rptr

y = x.lptr

z = New class_of_z

y = y.lptr

z.sum = x.data + y.data

return z.sum

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

Live

Dead
Liveness of Stack Data: An Informal Introduction (2)

- Current values of x, y, and z are used beyond this program point
- The value of y is different before and after the assignment to y
Liveness of Stack Data: An Informal Introduction (2)

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

```plaintext
w = x

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

y = x.lptr

z = New class of z

y = y.lptr

z.sum = x.data + y.data

return z.sum
```
Liveness of Stack Data: An Informal Introduction (2)

```plaintext
w = x

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

y = x.lptr

z = New class_of.z

y = y.lptr

z.sum = x.data + y.data

return z.sum
```

- The current values of `x` is used beyond this program point.
- Current values of `y` and `z` are not used beyond this program point.
Liveness of Stack Data: An Informal Introduction (2)

\[ w = x \]

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

\[ x = x.\text{rptr} \]

\[ y = x.\text{lptr} \]

\[ z = \text{New \ class\_of\_z} \]

\[ y = y.\text{lptr} \]

\[ \text{z.\text{sum}} = x.\text{data} + y.\text{data} \]

\[ \text{return } \text{z.\text{sum}} \]

- Nothing is known as of now
- Some information will be available in the next iteration point
Liveness of Stack Data: An Informal Introduction (2)

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

- Current value of x is used beyond this program point
- However its value is different before and after the assignment
Liveness of Stack Data: An Informal Introduction (2)

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

```
  w = x
  while (x.data < MAX)
    x = x.rptr
  y = x.lptr
  z = New class of z
  y = y.lptr
  z.sum = x.data + y.data
  return z.sum
```
Liveness of Stack Data: An Informal Introduction (2)

\[ w = x \]

while \( (x.data < MAX) \)

\[ x = x.rptr \]

\[ y = x.lptr \]

\[ z = \text{New class_of_z} \]

\[ y = y.lptr \]

\[ z.sum = x.data + y.data \]

return \( z.sum \)

Current value of \( x \) is used beyond this program point
Liveness of Stack Data: An Informal Introduction (2)

```plaintext
w = x

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

y = x.lptr

z = New class of z

y = y.lptr

z.sum = x.data + y.data

return z.sum
```

Current value of x is used beyond this program point
Liveness of Stack Data: An Informal Introduction (2)

\begin{align*}
  w &= x \\
  \text{while} \ (x.\text{data} < \text{MAX}) & \Rightarrow 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} \\
  \text{return } z.\text{sum}
\end{align*}

End of iteration #1
Liveness of Stack Data: An Informal Introduction (2)

\[ w = x \]

while \((x.data < MAX)\)

\[ x = x.rptr \]

\[ y = x.lptr \]

\[ z = \text{New class of } z \]

\[ y = y.lptr \]

\[ z.sum = x.data + y.data \]

return \(z.sum\)
Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the while loop is not executed even once.

```plaintext
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
8. return z.sum
```
Liveness Analysis of Heap Data
If the `while` loop is executed once.

1. \(w = x\) // \(x\) points to \(m_a\)
2. `while (x.data < MAX)`
3. \(x = x.rptr\)
4. \(y = x.lptr\)
5. \(z =\) New class of \(z\)
6. \(y = y.lptr\)
7. \(z.sum = x.data + y.data\)
8. return \(z.sum\)
Liveness Analysis of Heap Data
If the `while` loop is executed twice.

```plaintext
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
8  return z.sum
```

Diagram:
- **Stack**
  - `w`
  - `x`
  - `y`
- **Heap**
  - `p`
  - `q`
  - `r`
  - `s`
  - `t`
  - `u`
  - `v`
  - `w`
  - `x`
  - `y`
  - `z`
  - `a`
  - `b`
  - `c`
  - `d`
  - `e`
  - `f`
  - `g`
  - `h`
  - `i`
  - `j`
  - `k`
  - `l`
  - `m`
  - `n`
  - `o`
  - `p`
  - `q`
  - `r`
  - `s`
  - `t`
  - `u`
  - `v`
  - `w`
  - `x`
  - `y`
  - `z`
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)

1. $w = x$
   
2. while ($x.data < \text{MAX}$)
   
   \{
   \begin{align*}
   &x.lptr = null \\
   &x = x.rptr
   \end{align*}
   \}

3. \begin{align*}
   &x.rptr = x.lptr.rptr = null \\
   &x.lptr.lptr.lptr = x.lptr.lptr.rptr = null
   \end{align*}

4. $y = x.lptr$
   
5. \begin{align*}
   &x.lptr = y.rptr = null \\
   &y.lptr.lptr = y.lptr.rptr = null
   \end{align*}

6. $z = \text{New class of } z$
   
7. $z.sum = x.data + y.data$
   
8. return $z.sum$

Dec 2019 IIT Bombay
Our Solution (2)

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

While loop is not executed even once
Our Solution (2)

1. w = x
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

8. return z.sum

While loop is not executed even once
Our Solution (2)

1. \( w = x \)
   \( w = \text{null} \)
2. while \((x.\text{data} < \text{MAX})\) {
   \( x.\text{lptr} = \text{null} \)
   \( x = x.\text{rptr} \)
   \( x.\text{rptr} = x.\text{lptr}.\text{rptr} = \text{null} \)
   \( x.\text{lptr}.\text{lptr}.\text{lptr} = \text{null} \)
   \( x.\text{lptr}.\text{lptr}.\text{rptr} = \text{null} \)
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. \( z.\text{sum} = x.\text{data} + y.\text{data} \)
   \( x = y = \text{null} \)
6. return \( z.\text{sum} \)
   \( z = \text{null} \)

While loop is not executed even once
Our Solution (2)

```
y = z = null
1  w = x
w = null
2  while (x.data < MAX)
    {  x.lptr = null
        x = x.rptr
    }  x.rptr = x.lptr.rptr = null
x.lptr.lptr.lptr = null
x.lptr.lptr.rptr = null
3  y = x.lptr
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
y = y.lptr
y.lptr = y.rptr = null
6  y = y.lptr
z.sum = x.data + y.data
x = y = null
7  return z.sum
z = null
8```

While loop is not executed even once
Our Solution (2)

```
y = z = null
1  w = x
   w = null
2  while (x.data < MAX)
   {  x.lptr = null
       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
5  z.lptr = z.rptr = null
6  y = y.lptr
   y.lptr = y.rptr = null
7  z.sum = x.data + y.data
8  x = y = null
   return z.sum
   z = null
```

While loop is not executed even once
Our Solution (2)

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

While loop is not executed even once
Our Solution (2)

```
y = z = null
1  w = x
   w = null
2  while (x.data < MAX)
   {  x.lptr = null
      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 = null
8  return z.sum
   z = null
```

While loop is not executed even once
Our Solution (2)

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

While loop is executed once
Our Solution (2)

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
   x = y = null

7 return z.sum
   z = null

**While loop is executed twice**
Some Observations

```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.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.sum = x.data + y.data
6 x = y = null
7 return z.sum
8 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.
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

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. \( x = y = null \)
8. return \( z.sum \)
Some Observations

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

```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.lptr = null
    x.lptr.lptr.rptr = null

3 x = x.rptr

4 y = x.lptr
    x.lptr = y.rptr = null
    y.lptr.lptr = y.lptr.rptr = null

5 z = New class
    z.lptr = z.rptr = null

6 y = y.lptr
    y.lptr = y.rptr = null

7 z.sum = x.data + y.data
    x = y = null

8 return z.sum
    z = null
```
### Some Observations

```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.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
   y.lptr = y.rptr = null
6 z.sum = x.data + y.data
7 x = y = null
8 return z.sum
z = null
```

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

Static

Dynamic
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Dynamic

Program Execution
BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

Static

Program Code

Dynamic

Program Execution

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

Program Execution

Heap Memory

Summary Heap Data

Heap Memory
BTW, What is Static Analysis of Heap?

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

Static:
- Program Code
- Summary Heap Data

Dynamic:
- Program Execution
- Profiling

Heap Memory

Profiling

Dec 2019
IIT Bombay
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

Heap Memory

Heap Memory

Heap Memory

?
Part 4

What is Program Analysis?
What is Program Analysis?

Discovering information about a given program
What is Program Analysis?

Discovering information about a given program

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

Discovering information about a given program

- Representing the dynamic behaviour of the program
- Most often obtained without executing the program
  - Static analysis Vs. Dynamic Analysis
  - Example of loop tiling for parallelization
What is Program Analysis?

Discovering information about a given program

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

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

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

- **Software engineering**
  - Maintenance, bug fixes, enhancements, migration
  - Example: Y2K problem

- **Reverse engineering**
  To understand the program
Important Requirements of Static Analysis

- We discuss the following important requirements
  - Soundness
  - Precision
  - Efficiency
  - Scalability

- Soundness and precision are described more formally later in module 2
Inexactness of Static Analysis Results

- Static analysis predicts run time behaviour of programs
- Static analysis is undecidable
  
  there cannot exist an algorithm that can compute
  exact result for every program

- Possible reasons of undecidability
  - Values of variables not known
  - Branch outcomes not known
  - Infinitely many paths in the presence of loops or recursion
  - Infinitely many values

- Static analysis predictions may not match the actual run time behaviour
Possible Errors in Static Analysis Predictions

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

  (Error ≡ Mismatch between run time behaviour and predicted behaviour)

• Some of these errors may be harmless whereas some may be harmful

• Some of these errors may be unavoidable (recall undecidability)

• How do we characterize, identify, and minimize, these errors?
Examples of Harmless and Harmful Errors in Predictions (1)

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

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

- Weather prediction during rainy season
  - A doubtful prediction of “heavy to very heavy rain” is harmless.
  - Not predicting “heavy to very heavy rain” could be harmful.
Examples of Harmless and Harmful Errors in Predictions (2)

- For medical diagnosis
  - Subjecting a person to further investigations may be erroneous but in most cases it is harmless
  - Avoiding further investigations even after some suspicions could be harmful

- For establishing justice in criminal courts
  - Starting with the assumption that an accused is innocent may be erroneous but is harmless
  - Starting with the assumption that an accused is guilty may be harmful
Harmless Errors and Harmful Errors in Static Analysis

• For a static analysis,
  o Harmless errors can be tolerated but should be minimized
  o Harmful errors MUST be avoided

• Some behaviours concluded by a static analysis are
  o uncertain and cannot be guaranteed to occur at run time,
    (This uncertainty is harmless and hence is conservative)
  o certain and can be guaranteed to occur at run time
    (The absence of this certainty for these behaviours may be harmful)
Examples of Conservative and Definite Information

- Liveness is uncertain (also called *conservative*)
  
  If a variable is declared live at a program point, it may or may not be used beyond that program point at run time
  
  (Why is it harmless if the variable is not actually used?)

- Deadness (i.e. absence of liveness) is certain (also called *definite*)
  
  If a variable is declared to be dead at a program point, it is guaranteed to be not used beyond that program point at run time
  
  (Why is it harmful if the variable is not actually dead?)
Efficiency and Scalability

- **Efficiency**
  - How well are resources used
  - Measured in terms of work done per unit resource
  - Resources: time, memory, power, energy, processors, network etc.
  - Example: Strike rate of a batter in cricket

- **Scalability**
  - How large inputs can be handled
  - Measured in terms of size of the input
  - Example: Total runs scored by a batter in cricket

- Efficiency and scalability are orthogonal
  - Efficiency does not necessarily imply scalability
  - Scalability does not necessarily imply efficiency
Efficiency and Scalability May be Unrelated

Examples of the combinations of efficiency and scalability from sorting algorithms

<table>
<thead>
<tr>
<th>Scalable</th>
<th>Efficient</th>
<th>Inefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td></td>
<td>Selection Sort</td>
</tr>
<tr>
<td>Non-scalable</td>
<td>Quicksort</td>
<td>Bubble Sort</td>
</tr>
</tbody>
</table>
Practical Static Analysis

- The goodness of a static analysis lies in minimizing imprecision without compromising on soundness

  Additional expectations: Efficiency and scalability

- Some applications (e.g. debugging) do not need to be sound

  Ex: Traffic police catching people for traffic violations

- Some features of a programming language may not be covered

  (e.g. “eval” in Javascript, aliasing of array indices, effect of libraries)

- Accept a “soundy” analysis [Livshits et. al. CACM 2015]

  OR

  Tolerate imprecision for complete soundness
The Goal of Program Analysis

Constructing suitable abstractions for sound & precise modelling of runtime behaviour of programs efficiently
The Goal of Program Analysis

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

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

**Static**

- Program Code
- **Static Analysis**
- Summary Information

**Dynamic**

- Program Execution
- Memory
- Memory
- Memory
- Memory
- Memory
- Memory
- Memory
- Memory
- Memory
Part 5

An Overview of the Tutorial
Topics Covered

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

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

Interleaving of interactive

- Lectures
- Demos
- Paper exercises
- Computer exercises using SPAN tool
Questions ??
Part 6

Program Model
Program Representation

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

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

- Initially only intraprocedural programs
  Function calls brought in later
int main()
{
    int a, b, c, n;
    a = 4;
    b = 2;
    c = 3;
    n = c*2;
    while (a <= n)
    {
        a = a+1;
    }
    if (a < 12)
        a = a+b+c;
    return a;
}
An Example Program

```c
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;
    }
    else
    {
        goto 8;
    }
}
```

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