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

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
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);
}
```

Intermediate Code

For the boxed source code

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

Intermediate Code: Observations

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

Yet undecipherable!
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 B2
10. \( j = j - 1 \)
11. \( t4 = 4 \times j \)
12. \( t5 = a[t4] \)
13. if \( t5 > v \) goto 10
14. if \( i >= j \) 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

- Basic Blocks: B1, B2, B3, B4, B5
- Observations:
  - Nesting Level: 0, 1, 2
  - Basic Blocks: B1, B6, B4, B5, B2, B3
  - No. of Statements: 14, 11, 8
### Local Common Subexpression Elimination

```
B1
i = m - 1
j = 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[t1] = x
goto B2
```

### Global Common Subexpression Elimination

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

---

### Local Common Subexpression Elimination

```
B1
i = m - 1
j = 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[t1] = x
goto B2
```

### Global Common Subexpression Elimination

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

---

### Local Common Subexpression Elimination

```
B1
i = m - 1
j = 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[t1] = x
goto B2
```

### Global Common Subexpression Elimination

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

---
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 * n
  v = t6
  t2 = 4 * i
  t3 = a[t2]
  if t3 < v goto B2

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

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

B4
  if i >= j 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
  t1 = 4 * n
  v = t6
  t2 = 4 * i
  t3 = a[t2]
  if t3 < v goto B2

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

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

B4
  if i >= j 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
  v = t6
  t2 = 4 * i
  t3 = a[t2]
  if t3 < v goto B2

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

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

B4
  if t2 >= t4 goto B6
  if i >= j goto B6

B5
  a[t2] = t5
  a[t4] = t3
  goto B2

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

July 2017 IIT Bombay
### Optimized Program Flow Graph

<table>
<thead>
<tr>
<th>Nesting Level</th>
<th>No. of Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>8</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>Transformations</th>
<th>Machine Dependent</th>
<th>Independent Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Motion</td>
<td>Machine Independent</td>
<td>Flow Analysis (Data + Control)</td>
</tr>
<tr>
<td>Redundancy Elimination</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control Flow Optimization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop Transformations</td>
<td>Machine Dependent</td>
<td>Dependence Analysis (Data + Control)</td>
</tr>
<tr>
<td>Instruction Scheduling</td>
<td>Machine Dependent</td>
<td>Several Independent Techniques</td>
</tr>
<tr>
<td>Register Allocation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peephole Optimization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vectorization</td>
<td>Machine Dependent</td>
<td>Dependence Analysis (Data + Control)</td>
</tr>
<tr>
<td>Parallelization</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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 (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

Part 3

Optimizing Heap Memory Usage

Standard Memory Architecture of Programs

Heap allocation provides the flexibility of

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

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

Managing Heap Memory

Decision 1: When to Allocate?

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

Decision 2: When to Deallocate?

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

All white nodes are unused and should be considered garbage

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

Comparison between different sets of objects:

Live ⊆ Reachable ⊆ Allocated

The objects that are not live must be reclaimed.

¬ Live ⊆ ¬ Reachable ⊆ ¬ 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.

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

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)

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?

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.
Liveness of Stack Data: An Informal Introduction

1. \( w = x \) \hspace{1em} // 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. \( z\.sum = x.data + y.data \)

Stack

Heap

Accessing the location and reading its contents

What is the meaning of the use of data?

Reading \( x \) (Stack 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. \( z = \text{New } \text{class of } z \)
6. \( y = y.lptr \)
7. \( z.sum = x.data + y.data \)

Accessing the location and reading its contents

Reading \( x.data \) (Heap data)

End of iteration #1

Live

Dead

End of iteration #2
Applying Cedar Mesa Folk Wisdom to Heap Data

Liveness Analysis of Heap Data

If the while loop is not executed even once.

1. \( w = x \)  
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. \( z.\text{sum} = x.\text{data} + y.\text{data} \)

Stack

Heap

The Moral of the Story

- Mappings between access expressions and l-values keep changing

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

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

What about heap data?

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

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

While loop is executed twice

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
   } x = x.rptr
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 y = y.lptr
   y.lptr = y.rptr = null
7 z.sum = x.data + y.data
   x = y = z = null
Some Observations

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

Node i is live but link a → i is nullified

- 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

Stack

Heap
Some Observations

New access expressions are created. Can they cause exceptions?

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

BTW, What is Static Analysis of Heap?

Abstract, Bounded, Single Instance

Concrete, Unbounded, Infinitely Many

The Main Theme of the Course

Constructing suitable abstractions for sound & precise modelling of runtime behaviour of programs efficiently
Sequence of Generalizations in the Course Modules

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

- Intraprocedural Level
- Interprocedural Level
- Bit Vector Frameworks
- General frameworks
- Theoretical abstractions

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>Component</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>Total</td>
<td>100%</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
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 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

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;
}
}
```

```
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
```
Part 6
Soundness and Precision

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)

Soundness and Precision of Static Analysis
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

Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets
{(5, 2, 7), (−5, −2, 8)}

Static Analysis Computes Abstractions of Traces (1)

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 (1)

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

Trace 2
\[
\begin{align*}
\text{Entry}_1, (−5, −2, 8) \\
\text{Entry}_2, (−5, −2, −1) \\
\text{Entry}_3, (−5, −1, −1) \\
\text{Entry}_4, (−5, −1, −1) \\
\text{Entry}_5, (−5, −1, −1) \\
\text{Entry}_6, (−5, −1, −1)
\end{align*}
\]

A possible static abstraction using sets
{(5, 2, 7), (−5, −2, 8)}

1. \(c = a/2\)
2. \(b = -\text{abs}(b)\)
3. \(b = b + 1\)
4. \(\text{if } (b > 0)\)
5. \(b = 0\)
6. \(\text{return } b\)
Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

Trace 1
\[ a = \{-5, 5\}, b = \{-2, 2\}, c = \{7, 8\} \]

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

1. \[ \text{if } (b < c) \]
2. \[ b = \{-2, -1\} \]
3. \[ b = b + 1 \]
4. \[ \text{if } (b > 0) \]
5. \[ b = 0 \]
6. \[ \text{return } b \]

We only show the values of \( b \)

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

At a program point $p$
- $a \rightarrow 1 \Rightarrow a$ is live at $p$
- $a \rightarrow 0 \Rightarrow a$ is not live at $p$

Static Analysis Computes Abstractions of Traces (2)

A possible static abstraction using sets

Trace 1

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

Trace 2

$\begin{align*}
&\text{Entry}_1, (5, -2, 8) \\
&\text{Entry}_2, (5, -2, 1) \\
&\text{Entry}_3, (5, -2, 1) \\
&\text{Entry}_4, (5, -2, 1) \\
&\text{Entry}_5, (5, 0, 1) \\
&\text{Entry}_6, (5, 0, 1)
\end{align*}$

A possible static abstraction using sets

Trace 1

$\{b, c\}$

Trace 2

$\{b\}$

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

At a program point $p$

- $a \rightarrow 1$ ⇒ $a$ is live at $p$
- $a \rightarrow 0$ ⇒ $a$ is not live at $p$

**Trace 1**

```
a b c
Entry1, (1, 1, 0)
Entry2, (0, 1, 1)
Entry3, (0, 1, 1)
Entry4, (0, 1, 1)
Entry5, (0, 1, 1)
Entry6, (0, 1, 0)
```

```
101 or \{a, b\}
```

**Trace 2**

```
a b c
Entry1, (1, 1, 0)
Entry2, (0, 1, 1)
Entry3, (0, 1, 1)
Entry4, (0, 1, 1)
Entry5, (0, 1, 1)
Entry6, (0, 1, 0)
```

```
011 or \{b, c\}
```

At a program point $p$

- $a \rightarrow 1$ ⇒ $a$ is live at $p$
- $a \rightarrow 0$ ⇒ $a$ is not live at $p$

**Trace 1**

```
a b c
Entry1, (1, 1, 0)
Entry2, (0, 1, 1)
Entry3, (0, 1, 1)
Entry4, (0, 1, 1)
Entry5, (0, 1, 1)
Entry6, (0, 1, 0)
```

```
101 or \{a, b\}
```

**Trace 2**

```
a b c
Entry1, (1, 1, 0)
Entry2, (0, 1, 1)
Entry3, (0, 1, 1)
Entry4, (0, 1, 1)
Entry5, (0, 1, 1)
Entry6, (0, 1, 0)
```

```
011 or \{b, c\}
```

Entry
Computing Static Abstraction for Liveness of Variables

At a program point \( p \)

\( a \rightarrow 1 \Rightarrow a \) is live at \( p \)

\( a \rightarrow 0 \Rightarrow a \) 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\(_4\), \((0, 1, 1)\)

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\(_4\), \((0, 1, 0)\)

Entry\(_5\), \((0, 1, 0)\)

Entry\(_6\), \((0, 1, 0)\)

**Soundness of Abstractions (1)**

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

**Soundness of Abstractions (2)**

- An unsound abstraction
- A sound abstraction using intervals

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, ...)  
- block 4 (e.g. 0)

\( b \) can be 1 because of the increment in basic block 1

\( c = a\%2 \)

\( b = -\text{abs}(b) \)
Soundness of Abstractions (2)

An unsound abstraction

```
a = {-5, 5}, b = {-2, 2}, c = {7, 8}
1 c = a%2
    b = - abs(b)
  b = {-2, -1, 0, 1}
if (b < c) 2
b = b + 1
  if (b > 0) 3
b = 0
  return b
4 b = {−∞, 1}
5 return b
6 b = {−1, 0}
```

A sound abstraction using intervals

```
a = [−∞, ∞], b = [−∞, ∞], c = [−∞, ∞]
1 c = a%2
    b = - abs(b)
  b = {−∞, 1}
if (b < c) 2
b = b + 1
  if (b > 0) 3
b = 0
  return b
4 b = [-1, 1]
5 return b
6 b = [-1, 0]
```

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

A sound abstraction using intervals

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

Soundness of Abstractions for Liveness Analysis

A sound abstraction

```
a = [−∞, ∞], b = [−∞, ∞], c = [−∞, ∞]
1 c = a%2
    b = - abs(b)
  b = {−∞, 1}
if (b < c) 2
b = b + 1
  if (b > 0) 3
b = 0
  return b
4 b = [−1, 1]
5 b = 0
6 return b
```

An unsound abstraction

```
a = [−∞, ∞], b = [−∞, ∞], c = [−∞, ∞]
1 c = a%2
    b = - abs(b)
  b = {−∞, 1}
if (b < c) 2
b = b + 1
  if (b > 0) 3
b = 0
  return b
4 b = [−1, 1]
5 b = 0
6 return b
```

Precision of Sound Abstractions (1)

Sound but imprecise Sound and more precise Sound and even more precise

Precision of Sound Abstractions (1)

Sound but imprecise Sound and more precise Sound and even more precise

- Precision is relative, soundness is absolute
- Qualifiers “more precise” and “less precise” are meaningful
- Qualifiers “more sound” and “less sound” are not meaningful
A precise abstraction using intervals:
\[
a = [-\infty, \infty], \quad b = [-\infty, \infty], \quad c = [-\infty, \infty]
\]
1. \( c = a \% 2 \)
2. \( b = -\text{abs}(b) \)
3. \( b = [b, c] \)
4. \( \text{if } (b > 0) \quad b = b + 1 \)
5. \( b = [b, 0] \)
6. \( \text{return } b \)

An imprecise abstraction using intervals:
\[
a = [-\infty, \infty], \quad b = [-\infty, \infty], \quad c = [-\infty, \infty]
\]
1. \( c = a \% 2 \)
2. \( b = -\text{abs}(b) \)
3. \( b = [b, c] \)
4. \( \text{if } (b > 0) \quad b = b + 1 \)
5. \( b = [b, 0] \)
6. \( \text{return } b \)

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