Static Analysis

15-654:
Analysis of Software Artifacts

Jonathan Aldrich

Find the Bug!

Source: Engler et al., Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions, OSDI '00.

```c
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
    int b_size) {
    struct buffer_head *bh;
    unsigned long flags;
    save_flags(flags);
    cli();
    if ((bh = sh->buffer_pool) == NULL)
        return NULL;
    sh->buffer_pool = bh->b_next;
bh->b_size = b_size;
    restore_flags(flags);
    return bh;
}
```
Metal Interrupt Analysis

```c
#include "linux-includes.h"

sm_check_interrupts {
    // Variables
    decl (unsigned) flags;

    // Patterns
    // to specify enable/disable functions.
pat enable = { sti(); };
    pat disable = { cli(); };

    // States
    // The first state is the initial state.
    is_enabled: disable => is_disabled
    | enable => { err("double enable"); }
    ;
    is_disabled: enable => is_enabled
    | disable => { err("double disable"); }
    // Special pattern that matches when the SM
    // hits the end of any path in this state.
    | $end_of_path$ =>
    { err("exiting w/intr disabled!"); }
    ;
}
```

Applying the Analysis

```c
/* From Linux 2.3.99 drivers/block/raid5.c */

static struct buffer_head *
get_free_buffer(struct stripe_head *sh, ---------- initial state is_enabled
    int b_size) {
    struct buffer_head *bh;

    unsigned long flags;

    save_flags(flags);
    cli(); --------- transition to is-disabled
    if ((bh = sh->buffer_pool) == NULL)
        return NULL; --------- final state is_disabled: ERROR!
    sh->buffer_pool = bh->b_next;
    bh->b_size = b_size;
    restore_flags(flags); --------- transition to is_enabled
    return bh; --------- final state is_enabled is OK
}
```
Outline

- Why static analysis?
  - The limits of testing and inspection
- What is static analysis?
- Introduction to Dataflow Analysis
- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm

A problem has been detected and Windows has been shut down to prevent damage to your computer.

The problem seems to be caused by the following file: SPMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical Information:

*** STOP: 0x00000050 (0x08044C1,0x00000001,0xF8F76617,0x00000000)

*** SPMDCON.SYS - Address F8F76617 base at FBEF0000, datestamp 31d6dd67c
Static Analysis Finds “Mechanical” Errors

- Defects that result from inconsistently following simple, mechanical design rules
- Security vulnerabilities
  - Buffer overruns, unvalidated input…
- Memory errors
  - Null dereference, uninitialized data…
- Resource leaks
  - Memory, OS resources…
- Violations of API or framework rules
  - e.g. Windows device drivers; real time libraries; GUI frameworks
- Exceptions
  - Arithmetic/library/user-defined
- Encapsulation violations
  - Accessing internal data, calling private functions…
- Race conditions
  - Two threads access the same data without synchronization

Difficult to Find with Testing, Inspection

- Non-local, uncommon paths
  - Security vulnerabilities
  - Memory errors
  - Resource leaks
  - Violations of API or framework rules
  - Exceptions
  - Encapsulation violations

- Non-deterministic
  - Race conditions
Quality Assurance at Microsoft (Part 1)

- Original process: manual code inspection
  - Effective when system and team are small
  - Too many paths to consider as system grew
- Early 1990s: add massive system and unit testing
  - Tests took weeks to run
    - Diversity of platforms and configurations
    - Sheer volume of tests
    - Inefficient detection of common patterns, security holes
    - Non-local, intermittent, uncommon path bugs
    - Was treading water in Windows Vista development
- Early 2000s: add static analysis
  - More on this later

Process, Cost, and Quality

- Process intervention, testing, and inspection yield first-order software quality improvement
- Additional technology and tools are needed to close the gap
- Perfection (unattainable)
- Critical Systems Acceptability
- S&S, Agile, RUP, etc.: less rigorous ... more rigorous
Outline

- Why static analysis?
- What is static analysis?
  - Abstract state space exploration
- Introduction to Dataflow Analysis
- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm

Static Analysis Definition

- Static program analysis is the systematic examination of an abstraction of a program’s state space

- Metal interrupt analysis
  - Abstraction
    - 2 states: enabled and disabled
    - All program information—variable values, heap contents—is abstracted by these two states, plus the program counter
  - Systematic
    - Examines all paths through a function
    - What about loops? More later...
    - Each path explored for each reachable state
    - Assume interrupts initially enabled (Linux practice)
    - Since the two states abstract all program information, the exploration is exhaustive
Outline

- Why static analysis?
- What is static analysis?
- Introduction to Dataflow Analysis
- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm

An Analysis We’ve Seen

- Hoare logic
  - Useful for proving correctness
  - Requires a lot of work (even for ESC/Java)
  - Automated tool is unsound
    - So is manual proof, without a proof checker
Motivation: Dataflow Analysis

• Catch interesting errors
  • Non-local: x is null, x is written to y, y is dereferenced
• Optimize code
  • Reduce run time, memory usage…
• Soundness required
  • Safety-critical domain
    • Assure lack of certain errors
  • Cannot optimize unless it is proven safe
    • Correctness comes before performance
• Automation required
  • Dramatically decreases cost
  • Makes cost/benefit worthwhile for far more purposes

Dataflow analysis

• Tracks value flow through program
  • Can distinguish order of operations
    • Did you read the file after you closed it?
    • Does this null value flow to that dereference?
  • Differs from AST walker
    • Walker simply collects information or checks patterns
    • Tracking flow allows more interesting properties
• Abstracts values
  • Chooses abstraction particular to property
    • Is a variable null?
    • Is a file open or closed?
    • Could a variable be 0?
    • Where did this value come from?
  • More specialized than Hoare logic
    • Hoare logic allows any property to be expressed
    • Specialization allows automation and soundness
Zero Analysis

- Could variable x be 0?
  - Useful to know if you have an expression y/x
  - In C, useful for null pointer analysis
- Program semantics
  - $\eta$ maps every variable to an integer
- Semantic abstraction
  - $\sigma$ maps every variable to non zero (NZ), zero (Z), or maybe zero (MZ)
  - Abstraction function for integers $\alpha_{Z}$:
    - $\alpha_{Z}(0) = Z$
    - $\alpha_{Z}(n) = NZ$ for all $n \neq 0$
  - We may not know if a value is zero or not
    - Analysis is always an approximation
    - Need MZ option, too

Zero Analysis Example

\[
\sigma = [] \\
x := 10; \\
y := x; \\
z := 0; \\
while y > -1 do \\
  x := x / y; \\
y := y - 1; \\
z := 5; \\
\]

$\sigma = [x \mapsto \alpha_{Z}(10)]$
Zero Analysis Example

\[\sigma = []\]
\[x := 10;\]
\[y := x;\]
\[z := 0;\]
\[\text{while } y > -1 \text{ do}\]
\[\quad x := x / y;\]
\[\quad y := y - 1;\]
\[\quad z := 5;\]

\[\sigma = [x \mapsto \text{NZ}]\]
\[x := 10;\]
\[y := x;\]
\[z := 0;\]
\[\text{while } y > -1 \text{ do}\]
\[\quad x := x / y;\]
\[\quad y := y - 1;\]
\[\quad z := 5;\]
Zero Analysis Example

\[
\begin{align*}
\sigma &= [] \\
x &:= 10; & \sigma &= [x \mapsto \text{NZ}] \\
y &:= x; & \sigma &= [x \mapsto \text{NZ}, y \mapsto \text{NZ}] \\
z &:= 0; & \sigma &= [x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto \text{Z}] \\
\text{while } y > -1 \text{ do} & \sigma &= [x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto \text{Z}] \\
x &:= x / y; & \sigma &= [x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto \text{Z}] \\
y &:= y - 1; & \sigma &= [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{Z}] \\
z &:= 5; & \sigma &= [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{NZ}] 
\end{align*}
\]
Zero Analysis Example

\[\begin{align*}
\sigma &= [] \\
x &= 10; \quad \sigma = [x \mapsto \text{NZ}] \\
y &= x; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{NZ}] \\
z &= 0; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto Z] \\
\text{while } y > -1 \text{ do} \\
\quad x &= x / y; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto MZ] \\
\quad y &= y - 1; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto Z] \\
\quad z &= 5; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{NZ}]
\end{align*}\]

Nothing more happens!
Zero Analysis Termination

- The analysis values will not change, no matter how many times we execute the loop
  - Proof: our analysis is deterministic
  - We run through the loop with the current analysis values, none of them change
  - Therefore, no matter how many times we run the loop, the results will remain the same
  - Therefore, we have computed the dataflow analysis results for any number of loop iterations
- Why does this work
  - If we simulate the loop, the data values could (in principle) keep changing indefinitely
    - There are an infinite number of data values possible
    - Not true for 32-bit integers, but might as well be true
    - Counting to $2^{32}$ is slow, even on today’s processors
  - Dataflow analysis only tracks 2 possibilities!
    - So once we’ve explored them all, nothing more will change
    - This is the secret of abstraction
- We will make this argument more precise later

Using Zero Analysis

- Visit each division in the program
- Get the results of zero analysis for the divisor
- If the results are definitely zero, report an error
- If the results are possibly zero, report a warning

Quick Quiz

- Fill in the table to show how what information zero analysis will compute for the function given.

<table>
<thead>
<tr>
<th>Program Statement</th>
<th>Analysis Info after that statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: &lt;beginning of program&gt;</td>
<td></td>
</tr>
<tr>
<td>1: x := 0</td>
<td></td>
</tr>
<tr>
<td>2: y := 1</td>
<td></td>
</tr>
<tr>
<td>3: if (z == 0)</td>
<td></td>
</tr>
<tr>
<td>4: ( x := x + y )</td>
<td></td>
</tr>
<tr>
<td>5: else y := y – 1</td>
<td></td>
</tr>
<tr>
<td>6: w := y</td>
<td></td>
</tr>
</tbody>
</table>

Outline

- Why static analysis?
- What is static analysis?
- Introduction to Dataflow Analysis
- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm
Defining Dataflow Analyses

- **Lattice**
  - Describes program data abstractly
  - Abstract equivalent of environment
- **Abstraction function**
  - Maps concrete environment to lattice element
- **Flow functions**
  - Describes how abstract data changes
  - Abstract equivalent of expression semantics
- **Control flow graph**
  - Determines how abstract data propagates from statement to statement
  - Abstract equivalent of statement semantics

Lattice

- A lattice is a tuple \((L, \sqsubseteq, \sqcup, \bot, \top)\)
  - \(L\) is a set of abstract elements
  - \(\sqsubseteq\) is a partial order on \(L\)
    - Means at least as precise as
  - \(\sqcup\) is the least upper bound of two elements
    - Must exist for every two elements in \(L\)
    - Used to merge two abstract values
  - \(\bot\) (bottom) is the least element of \(L\)
    - Means we haven’t yet analyzed this yet
    - Will become clear later
  - \(\top\) (top) is the greatest element of \(L\)
    - Means we don’t know anything
  - \(L\) may be infinite
    - Typically should have finite height
      - All paths from \(\bot\) to \(\top\) should be finite
      - We’ll see why later
Zero Analysis Lattice

- **Integer zero lattice**
  - $L_{Z_l} = \{ \bot, Z, NZ, MZ \}$
  - $\bot \in Z, \bot \in NZ, NZ \subseteq MZ, Z \subseteq MZ$
    - $\bot \in MZ$ holds by transitivity
  - $\cup$ defined as join for $\subseteq$
    - $x \cup y = z$ iff
      - $z$ is an upper bound of $x$ and $y$
      - $z$ is the least such bound
    - Obeys laws: $\bot \cup \lambda' = \lambda', \top \cup \lambda' = \top, \lambda' \cup \lambda' = \lambda'$
    - Also $Z \cup NZ = MZ$
  - $\bot = \bot$
    - $\forall \lambda'. \bot \subseteq \lambda'$
  - $\top = MZ$
    - $\forall \lambda'. \lambda' \subseteq \top$

- **Program lattice is a tuple lattice**
  - $L_Z$ is the set of all maps from $\text{Var}$ to $L_{Z_l}$
  - $\sigma_1 \sqsubseteq_Z \sigma_2$ iff $\forall x \in \text{Var}. \sigma_1(x) \sqsubseteq_{Z_l} \sigma_2(x)$
  - $\sigma_1 \cup_Z \sigma_2 = \{ x \mapsto \sigma_1(x) \cup_{Z_l} \sigma_2(x) \mid x \in \text{Var} \}$
  - $\bot = \{ x \mapsto \bot_{Z_l} \mid x \in \text{Var} \}$
  - $\top = \{ x \mapsto \top_{Z_l} \mid x \in \text{Var} \} = \{ x \mapsto MZ \mid x \in \text{Var} \}$
  - Can produce a tuple lattice from any base lattice
    - Just define as above
One Path in a Tuple Lattice

\[
\top = \{ w \mapsto \text{MZ}, x \mapsto \text{MZ}, y \mapsto \text{MZ}, z \mapsto \text{MZ} \}
\]

\[
\vdots
\]

\[
\top = \{ w \mapsto \text{MZ}, x \mapsto \text{MZ}, y \mapsto \text{MZ}, z \mapsto \text{MZ} \}
\]

\[
\vdots
\]

\[
\top = \{ w \mapsto \text{MZ}, x \mapsto \text{MZ}, y \mapsto \text{MZ}, z \mapsto \text{MZ} \}
\]

\[
\vdots
\]

\[
\bot = \{ w \mapsto \text{NZ}, x \mapsto \text{NZ}, y \mapsto \bot_{Z I}, z \mapsto \bot_{Z I} \}
\]

\[
\vdots
\]

\[
\bot = \{ w \mapsto \bot_{Z I}, x \mapsto \bot_{Z I}, y \mapsto \bot_{Z I}, z \mapsto \bot_{Z I} \}
\]

\[
\bot = \{ w \mapsto \bot_{Z I}, x \mapsto \bot_{Z I}, y \mapsto \bot_{Z I}, z \mapsto \bot_{Z I} \}
\]

Analysis of Software Artifacts - Spring 2006
Outline

- Why static analysis?
- What is static analysis?
- Introduction to Dataflow Analysis
- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm

Abstraction Function

- Maps each concrete program state to a lattice element
  - For tuple lattices, the function can be defined for values and lifted to tuples
- Integer Zero abstraction function $\alpha_{ZI}$:
  - $\alpha_{ZI}(0) = Z$
  - $\alpha_{ZI}(n) = NZ$ for all $n \neq 0$
- Zero Analysis abstraction function $\alpha_{ZA}$:
  - $\alpha_{ZA}(\eta) = \{ x \mapsto \alpha_{ZI}(\eta(x)) | x \in \text{Var} \}$
  - This is just the tuple form of $\alpha_{ZI}(n)$
    - Can be done for any tuple lattice
Quick Quiz

- Consider the following two tuple lattice values: \([x \rightarrow Z, y \rightarrow MZ]\) and \([x \rightarrow MZ, y \rightarrow NZ]\)
  - How do the two compare in the lattice ordering for zero analysis?

- What is the join of these two tuple lattice values?

Outline

- Why static analysis?
- What is static analysis?
- Introduction to Dataflow Analysis
- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm
Control Flow Graph (CFG)

- Shows order of statement execution
- Determines where data flows
- Decomposes expressions into primitive operations
  - Typically one CFG node per “useful” AST node
    - constants, variables, binary operations, assignments, if, while...
- Loops are written out
  - Form a loop in the CFG
  - Benefit: analysis is defined one operation at a time

Intuition for Building a CFG

- Connect nodes in order of operation
  - Defined by language
- Java order of operation
  - Expressions, assignment, sequence
    - Evaluate subexpressions left to right
    - Evaluate node after children (postfix)
  - While, If
    - Evaluate condition first, then if/while
    - if branches to else and then
    - while branches to loop body and exit
Quick Quiz

• Draw a CFG for the following program:
  1: x := 0
  2: y := 1
  3: if (z == 0)
  4: x := x + y
  5: else y := y – 1
  6: w := y
Outline

• Why static analysis?
• What is static analysis?
• Introduction to Dataflow Analysis
• Dataflow Analysis Frameworks
  • Lattices
  • Abstraction functions
  • Control flow graphs
  • Flow functions
  • Worklist algorithm

Flow Functions

• Compute dataflow information after a statement from dataflow information before the statement
  • Formally, map a lattice element and a CFG node to a new lattice element
• Analysis performed on 3-address code
  • inspired by 3 addresses in assembly language: add x,y,z
• Convert complex expressions to 3-address code
  • Each subexpression represented by a temporary variable
  • $x+3*y \rightarrow t_1:=3; t_2:= t_1*y; t_3:=x+t_2$
While3Addr

- copy $x = y$
- binary op $x = y \text{ op } z$ (op $\in \{+, -, *, /, \ldots\}$)
- literal $x = n$
- unary op $x = \text{ op } y$ (op $\in \{!, ++, \ldots\}$)
- label label $lab$
- jump jump $lab$
- branch $\text{btrue } x \text{ } lab$

Zero Analysis Flow Functions

- $f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)] \sigma$
- $f_{ZA}(\sigma, [x := n]) = \text{if } n=0$
  - then $[x \mapsto Z] \sigma$
  - else $[x \mapsto NZ] \sigma$
- $f_{ZA}(\sigma, [x := \ldots]) = [x \mapsto MZ] \sigma$
- Could be more precise, e.g.
  - $f_{ZA}(\sigma, [x := y + z]) =$
  - if $\sigma[y]=Z$ & $\sigma[z]=Z$
  - then $[x \mapsto Z] \sigma$ else $[x \mapsto MZ] \sigma$
- $f_{ZA}(\sigma, /* \text{ any non-assignment */}) = \sigma$
Zero Analysis Example

\[ x := 0; \]
\[ \text{while } x > 3 \text{ do} \]
\[ x := x + 1 \]

Initial dataflow

\[ \sigma_i = \{ x \rightarrow \text{MZ} \mid x \in \text{Var} \} \]

Intuition:
We know nothing about initial variable values. We could use a precondition if we had one.
Zero Analysis Example

\[ \sigma_1 = \{ x \mapsto \text{MZ} \mid x \in \text{Var} \} \]

\[ \sigma_2 = f_{ZA}(\sigma_1, [t_2 := 0]) = [t_2 \mapsto \text{Z}] \sigma_1 \]

\[ f_{ZA}(\sigma, [x := n]) = \]

if \( n = 0 \)

then \([x \mapsto \text{Z}] \sigma \)

else \([x \mapsto \text{NZ}] \sigma \)
Zero Analysis Example

\[ \sigma_i = \{ x \rightarrow \text{MZ} \mid x \in \text{Var} \} \]
\[ \sigma_3 = [x \rightarrow Z, t_2 \rightarrow Z] \sigma_i \]

Input to [3]_5 comes from [:=]_3 and [:=]_12
Input should be \( \sigma_3 \cup \sigma_{12} \)

What is \( \sigma_{12} \)?
Solution: assume \( \bot \)
Benefit: \( \sigma_3 \cup \bot = \sigma_3 \)
Same result as ignoring back edge first time
Zero Analysis Example

\[ \sigma_1 = \{ x \mapsto \text{MZ} \mid x \in \text{Var} \} \]
\[ \sigma_3 = [x \mapsto \text{Z}, \ y \mapsto \text{Z}] \sigma_1 \]
\[ \sigma_{12} = \bot \]
\[ \sigma_5 = [y \mapsto \text{NZ}] \sigma_3 \]
\[ \sigma_6 = f_{ZA}(\sigma_5, [t_y := x < t_y]) \]
\[ = \sigma_5 \]
\[ = [y \mapsto \text{NZ}] \sigma_3 \]
\[ f_{ZA}(\sigma, /\forall \text{ any other } \forall) = \sigma \]

Zero Analysis Example

\[ \sigma_1 = \{ x \mapsto \text{MZ} \mid x \in \text{Var} \} \]
\[ \sigma_3 = [x \mapsto \text{Z}, \ y \mapsto \text{Z}] \sigma_1 \]
\[ \sigma_{12} = \bot \]
\[ \sigma_5 = [y \mapsto \text{NZ}] \sigma_3 \]

Skipping similar nodes...
Zero Analysis Example

\[ \sigma_1 = \{ x \rightarrow \text{MZ} \mid x \in \text{Var}\} \]
\[ \sigma_3 = [x \rightarrow Z, t_2 \rightarrow Z] \sigma_1 \]
\[ \sigma_{12} = \bot \]
\[ \sigma_{10} = [t_{10} \rightarrow \text{NZ}, \ldots] \sigma_3 \]
\[ \sigma_{11} = f_{ZA}(\sigma_{10}, [t_{11} := x + t_{10}]) \]
\[ = [t_{11} \rightarrow \text{MZ}] \sigma_{10} \]
\[ = f_{ZA}(\sigma, [x := y \text{ op } z]) = [x \rightarrow \text{MZ}] \sigma \]
\[ \sigma_{12} = f_{ZA}(\sigma_{11}, [x := t_{11}]) \]
\[ = [x \rightarrow \sigma_{11}(t_{11})] \sigma_{11} \]
\[ = [x \rightarrow \text{MZ}] \sigma_{11} \]
\[ = [x \rightarrow \text{MZ}, \ldots] \sigma_3 \]
\[ f_{ZA}(\sigma, [x := y]) = [x \rightarrow \sigma(y)] \sigma \]
Zero Analysis Example

\[ \sigma_1 = \{ x \rightarrow \text{MZ} \mid x \in \text{Var} \} \]
\[ \sigma_3 = [x \rightarrow Z, t_2 \rightarrow Z] \sigma_1 \]
\[ \sigma_{12} = [x \rightarrow \text{MZ}, \ldots] \sigma_3 \]

\[ \sigma_5 = f_{ZA}(\sigma_3 \cup \sigma_{12}, [t_5 := 3]) \]
\[ = f_{ZA}([x \rightarrow \text{MZ}]\sigma_3, [t_5 := 3]) \]
\[ = [t_5 \rightarrow \text{NZ}] [x \rightarrow \text{MZ}, \ldots] \sigma_3 \]
\[ = [t_5 \rightarrow \text{NZ}, x \rightarrow \text{MZ}, \ldots] \sigma_3 \]

\[ f_{ZA}(\sigma, [v_i]) = [t_k \rightarrow \sigma(x)] \sigma \]

---

Zero Analysis Example

\[ \sigma_1 = \{ x \rightarrow \text{MZ} \mid x \in \text{Var} \} \]
\[ \sigma_3 = [x \rightarrow Z, t_2 \rightarrow Z] \sigma_1 \]
\[ \sigma_{12} = [x \rightarrow \text{MZ}, \ldots] \sigma_3 \]

Propagation of \( x \rightarrow \text{MZ} \) continues

\( \sigma_{12} \) does not change, so no need to iterate again

---

Analysis of Software Artifacts - Spring 2006

57

58
Quick Quiz

- Explain in detail how the dataflow lattice value for after the statement \( w := y \) is computed, using the CFG below as your point of reference.

\[
\text{begin} \rightarrow x := 0 \rightarrow y := 1 \rightarrow z := 0 \leftarrow x := x + y \leftarrow y := y - 1 \rightarrow w := y
\]

- Answer:

Outline

- Why static analysis?
- What is static analysis?
- Introduction to Dataflow Analysis
- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm
Worklist Dataflow Analysis Algorithm

```javascript
worklist = new Set();
for all node indexes i do
    results[i] = ⊥_A:
    results[entry] = i_A:
    worklist.add(all nodes);

while (!worklist.isEmpty()) do
    i = worklist.pop();
    before = \bigcup_{k \in \text{pred}(i)} \text{results}[k];
    after = f_A(before, node(i));
    if (!\text{after} \subseteq \text{results}[i])
        \text{results}[i] = after;
    for all k \in \text{succ}(i) do
        worklist.add(k);
```

Ok to just add entry node if flow functions cannot return ⊥_A (examples will assume this)

Pop removes the most recently added element from the set (performance optimization)

Example of Worklist

<table>
<thead>
<tr>
<th>[a := 0]_1</th>
<th>Position</th>
<th>Worklist</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>[b := 0]_2</td>
<td>1</td>
<td>2</td>
<td>Z</td>
<td>MZ</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>while [a &lt; 2]_3 do</td>
<td>3</td>
<td>4,6</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5,6</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>[b := a]_4;</td>
<td>5</td>
<td>3,6</td>
<td>MZ</td>
<td>Z</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4,6</td>
<td>MZ</td>
<td>Z</td>
</tr>
<tr>
<td>[a := a + 1]_5;</td>
<td>4</td>
<td>5,6</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>3,6</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>[a := 0]_6</td>
<td>3</td>
<td>4,6</td>
<td>MZ</td>
<td>MZ</td>
</tr>
</tbody>
</table>

Control Flow Graph

```
1 → 2 → 3 → 6
   \(\bigtriangleup\)

4 → 5
```

Analysis of Software Artifacts - Spring 2006
Quick Quiz

Show how the worklist algorithm given in class operates on the program given, by filling in the table below.

1: x := 0
2: y := 1
3: if (z == 0)
4:   x := x + y
5: else y := y – 1
6: w := y

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>x</th>
<th>y</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Worklist Algorithm Performance

- Performance
  - Visits node whenever input gets less precise
    - up to \( h \) = height of lattice
  - Propagates data along control flow edges
    - up to \( e \) = max outbound edges per node
  - Assume lattice operation cost is \( o \)
  - Overall, \( O(h*e*o) \)
    - Typically \( h, o, e \) bounded by \( n = \) number of statements in program
    - \( O(n^2) \) for many data flow analyses
    - \( O(n^2) \) if you assume a number of edges per node is small
  - Good enough to run on a function
    - Usually not run on an entire program at once, because \( n \) is too big