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.

```
/* 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;
}
```

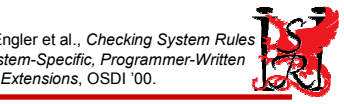
disable interrupts

ERROR: returning with interrupts disabled

re-enable interrupts

Metal Interrupt Analysis

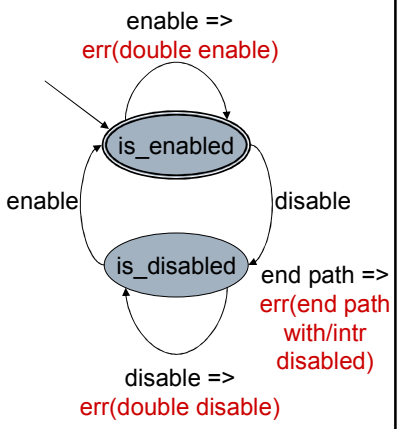
Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.



```
{ #include "linux-includes.h" }
sm check_interrupts {
  // Variables
  // used in patterns
  decl { unsigned } flags;

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

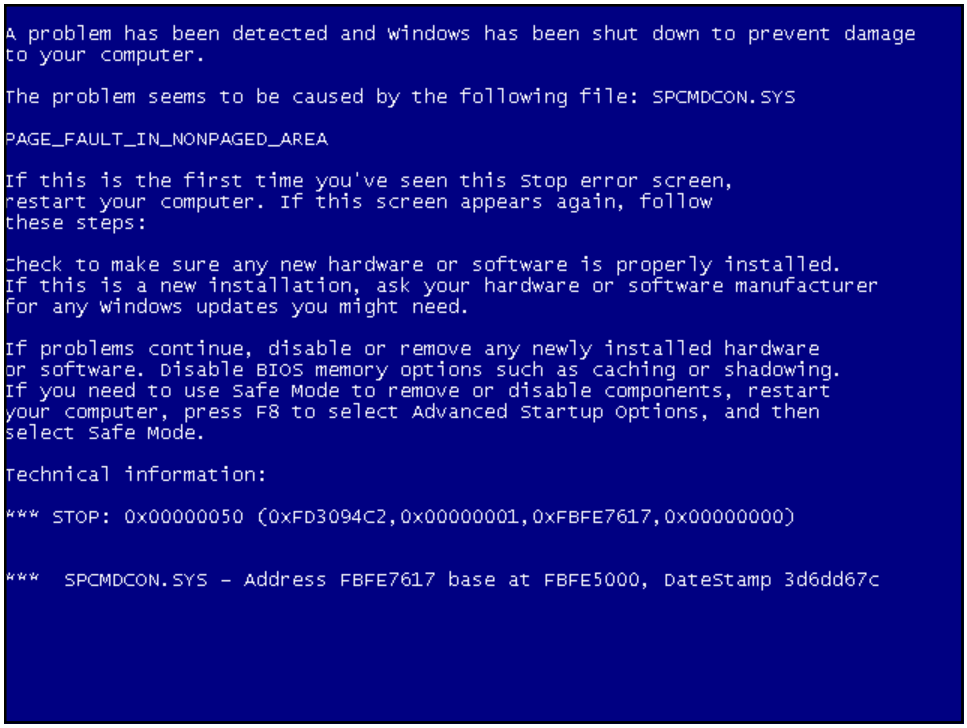
Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
/* 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



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

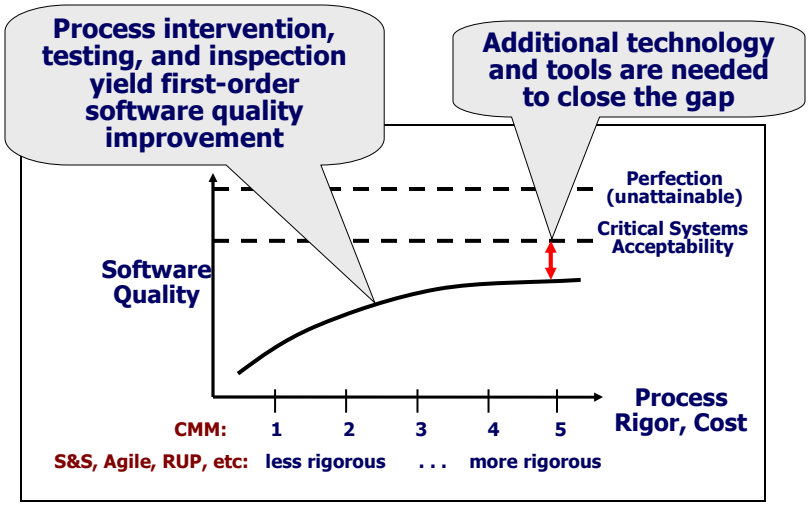
- Non-deterministic
  - Race conditions

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

Slide: William Scherlis



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
  - η maps every variable to an integer
- Semantic abstraction
  - σ maps every variable to non zero (NZ), zero(Z), or maybe zero (MZ)
  - Abstraction function for integers α_{ZI} :
    - $\alpha_{ZI}(0)$ = Z
    - $\alpha_{ZI}(n)$ = 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

σ =[]

x := 10;

σ =[x↦α_{ZI}(10)]

y := x;

z := 0;

while y > -1 do

    x := x / y;

    y := y-1;

    z := 5;

Zero Analysis Example

```
x := 10;
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-1;
    z := 5;
```

σ =[]
σ =[x↦NZ]
σ =[x↦NZ,y↦σ(x)]

Zero Analysis Example

```
x := 10;
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-1;
    z := 5;
```

σ =[]
σ =[x↦NZ]
σ =[x↦NZ,y↦NZ]
σ =[x↦NZ,y↦NZ,z↦α_{ZI}(0)]

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦NZ,z↦Z]
x := x / y;	σ =[x↦NZ,y↦NZ,z↦Z]
y := y-1;	σ =[x↦NZ,y↦MZ,z↦Z]
z := 5;	σ =[x↦NZ,y↦MZ,z↦NZ]

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦MZ,z↦MZ]
x := x / y;	σ =[x↦NZ,y↦NZ,z↦Z]
y := y-1;	σ =[x↦NZ,y↦MZ,z↦Z]
z := 5;	σ =[x↦NZ,y↦MZ,z↦NZ]

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦MZ,z↦MZ]
x := x / y;	σ =[x↦NZ,y↦MZ,z↦MZ]
y := y-1;	σ =[x↦NZ,y↦MZ,z↦Z]
z := 5;	σ =[x↦NZ,y↦MZ,z↦NZ]

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦MZ,z↦MZ]
x := x / y;	σ =[x↦NZ,y↦MZ,z↦MZ]
y := y-1;	σ =[x↦NZ,y↦MZ,z↦MZ]
z := 5;	σ =[x↦NZ,y↦MZ,z↦NZ]
	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.

Program Statement	Analysis Info after that statement
0: <*beginning of program*>	
1: x := 0	
2: y := 1	
3: if (z == 0)	
4: x := x + y	
5: else y := y – 1	
6: w := y	

Outline

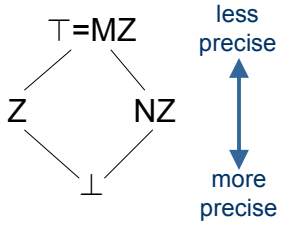
- Why static analysis?
- What is static analysis?
- Introduction to Dataflow Analysis
- Dataflow Analysis Frameworks
  - Lattices
  - Abstraction functions
  - Control flow graphs
  - Flow functions
  - Worklist algorithm

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

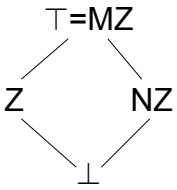
Lattice

- 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



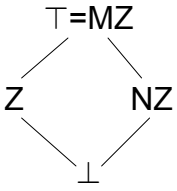
Zero Analysis Lattice

- Integer zero lattice
  - L_{ZI} = { ⊥, Z, NZ, MZ }
  - ⊥ ⊑ Z, ⊥ ⊑ NZ, NZ ⊑ MZ, Z ⊑ MZ
    - ⊥ ⊑ MZ holds by transitivity
  - ⊔ defined as join for ⊑
    - x ⊔ y = z iff
      - z is an upper bound of x and y
      - z is the least such bound
    - Obeys laws: $\bot \sqcup \mathcal{X} = \mathcal{X}$, $\top \sqcup \mathcal{X} = \top$, $\mathcal{X} \sqcup \mathcal{X} = \mathcal{X}$
    - Also Z ⊔ NZ = MZ
  - ⊥ = ⊥
    - $\forall \mathcal{X} . \bot \sqsubseteq \mathcal{X}$
  - ⊤ = MZ
    - $\forall \mathcal{X} . \mathcal{X} \sqsubseteq \top$



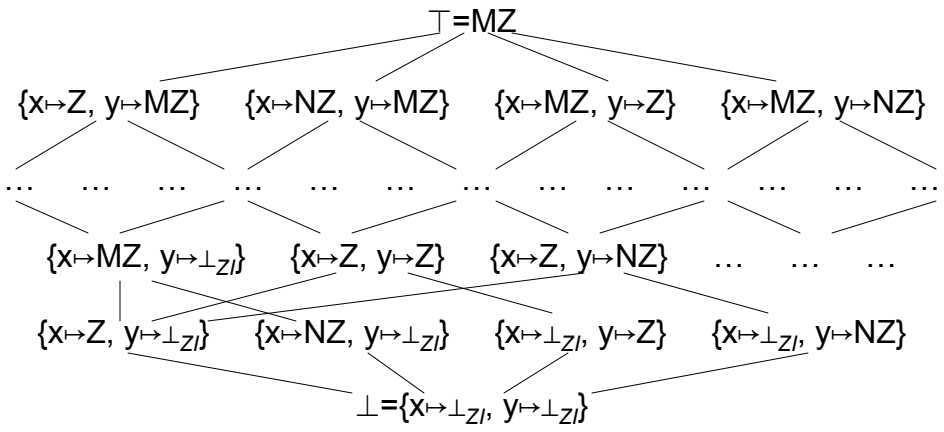
Zero Analysis Lattice

- Integer zero lattice
  - L_{ZI} = { ⊥, Z, NZ, MZ }
  - ⊥ ⊑ Z, ⊥ ⊑ NZ, NZ ⊑ MZ, Z ⊑ MZ
  - ⊔ defined as join for ⊑
  - ⊥ = ⊥
  - ⊤ = MZ
- Program lattice is a *tuple lattice*
  - L_Z is the set of all maps from **Var** to L_{ZI}
  - $\sigma_1 \sqsubseteq_Z \sigma_2$ iff $\forall x \in \mathbf{Var} . \sigma_1(x) \sqsubseteq_{ZI} \sigma_2(x)$
  - $\sigma_1 \sqcup_Z \sigma_2 = \{ x \mapsto \sigma_1(x) \sqcup_{ZI} \sigma_2(x) \mid x \in \mathbf{Var} \}$
  - $\bot = \{ x \mapsto \bot_{ZI} \mid x \in \mathbf{Var} \}$
  - $\top = \{ x \mapsto \top_{ZI} \mid x \in \mathbf{Var} \} = \{ x \mapsto MZ \mid x \in \mathbf{Var} \}$
  - Can produce a tuple lattice from *any* base lattice
    - Just define as above

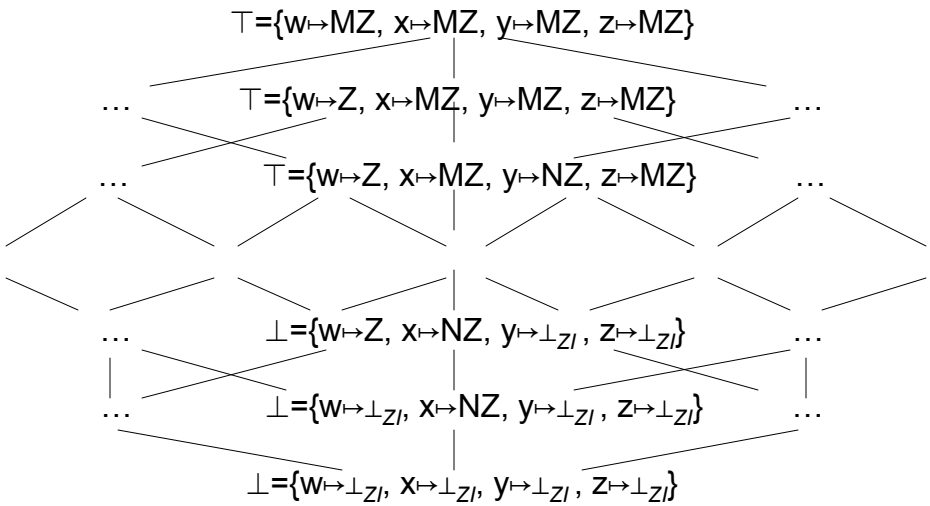


Tuple Lattices Visually

- For **Var** = { x,y }



One Path in a Tuple Lattice



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 α_{ZI}:
  - $\alpha_{ZI}(0) = Z$
  - $\alpha_{ZI}(n) = NZ$ for all $n \neq 0$
- Zero Analysis abstraction function α_{ZA}:
  - $\alpha_{ZA}(\eta) = \{x \mapsto \alpha_{ZI}(\eta(x)) \mid x \in \mathbf{Var}\}$
  - 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↦Z, y↦MZ] and [x↦MZ, y↦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

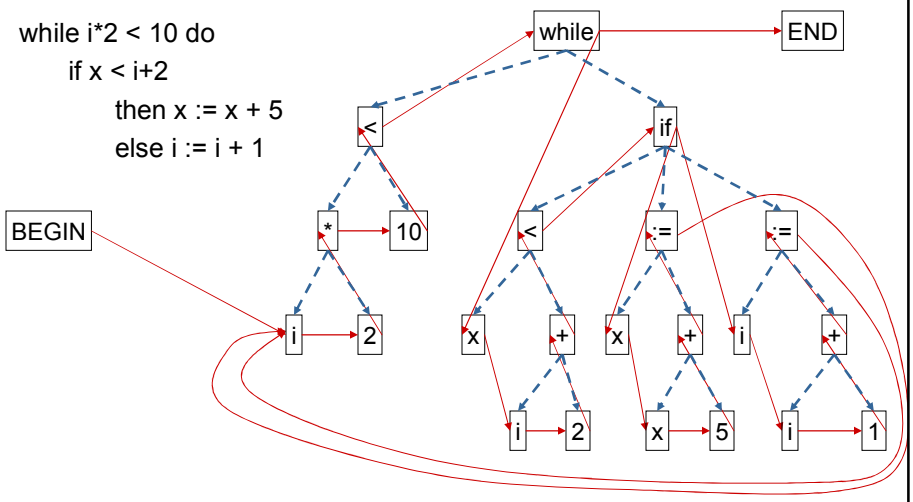
Control Flow Graph (CFG)

- 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

Control Flow Graph Example



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 ➔ t_1:=3; t_2:= t_1*y; t_3:=x+t_2

While3Addr

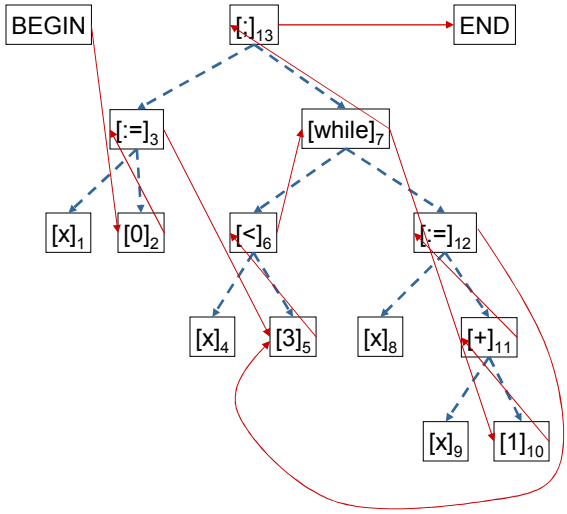
- copy — x = y
- binary op — x = y op z (op ∈ {+,-,*,/,…})
- literal — x = n
- unary op — x = op y (op ∈ {-,!,++,…})
- label — label *lab*
- jump — jump *lab*
- branch — btrue x *lab*

Zero Analysis Flow Functions

- $f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)]\ \sigma$
- $f_{ZA}(\sigma, [x := n])$ = 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 $[x \mapsto MZ]\sigma$
- $f_{ZA}(\sigma,$ */* any non-assignment */*$) = \sigma$

Zero Analysis Example

x := 0;
while x > 3 do
    x := x+1



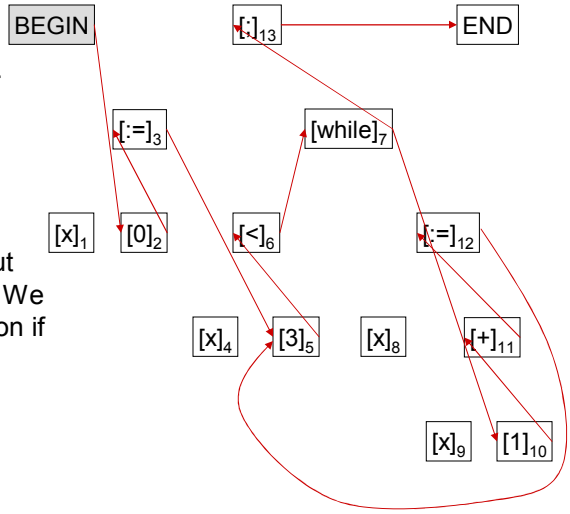
Zero Analysis Example

Initial dataflow

σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }

Intuition:

We know nothing about initial variable values. We could use a precondition if we had one.

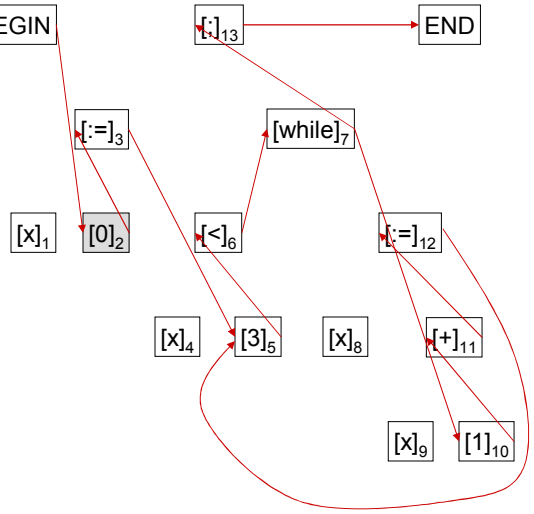


Zero Analysis Example

σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }

σ_2 = $f_{ZA}(\sigma_\iota, [t_2 := 0])$
= $[t_2 \mapsto Z]\ \sigma_\iota$

$f_{ZA}(\sigma, [x := n])$ =
if n==0
then $[x \mapsto Z]\sigma$
else $[x \mapsto NZ]\sigma$

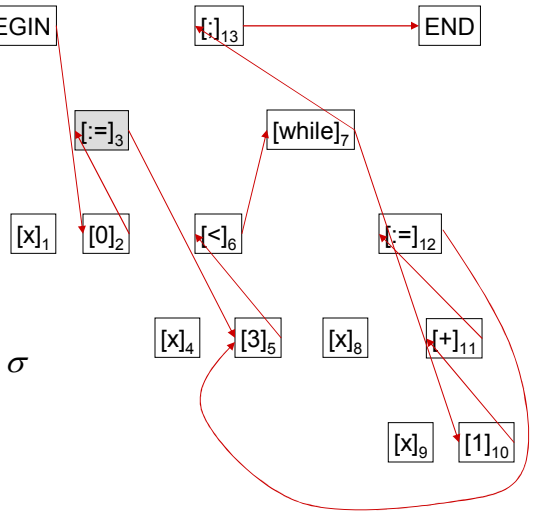


Zero Analysis Example

σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }
σ_2 = $[t_2 \mapsto Z]\ \sigma_\iota$

σ_3 = $f_{ZA}(\sigma_2, [x := t_2])$
= $[x \mapsto \sigma_2(t_2)]\ \sigma_2$
= $[x \mapsto Z]\ \sigma_2$
= $[x \mapsto Z, t_2 \mapsto Z]\ \sigma_\iota$

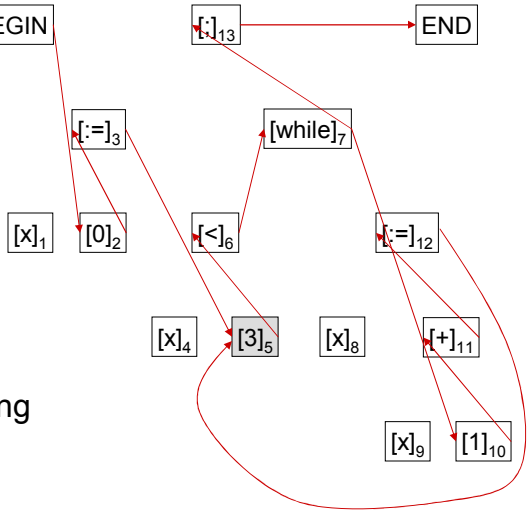
$f_{ZA}(\sigma, [x := y])$ = $[x \mapsto \sigma(y)]\ \sigma$



Zero Analysis Example

σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }
σ_3 = [x$\mapsto$Z, $t_2 \mapsto$Z] σ_ι

Input to $[3]_5$ comes from
$[:=]_3$ and $[:=]_{12}$
Input should be $\sigma_3 \sqcup \sigma_{12}$
What is σ_{12}?
Solution: assume $\bot$
Benefit: $\sigma_3 \sqcup \bot = \sigma_3$
Same result as ignoring back edge first time

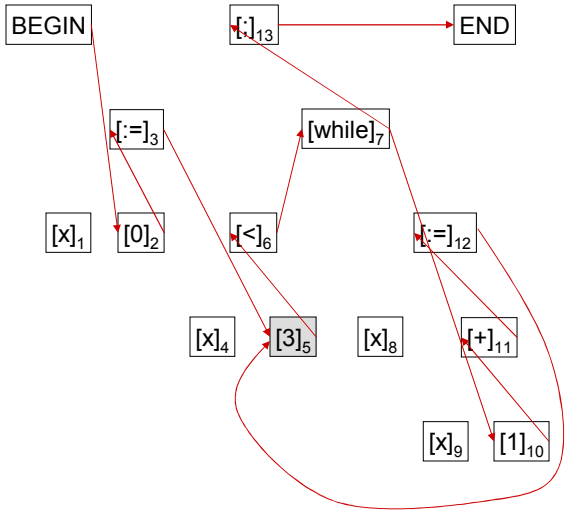


Zero Analysis Example

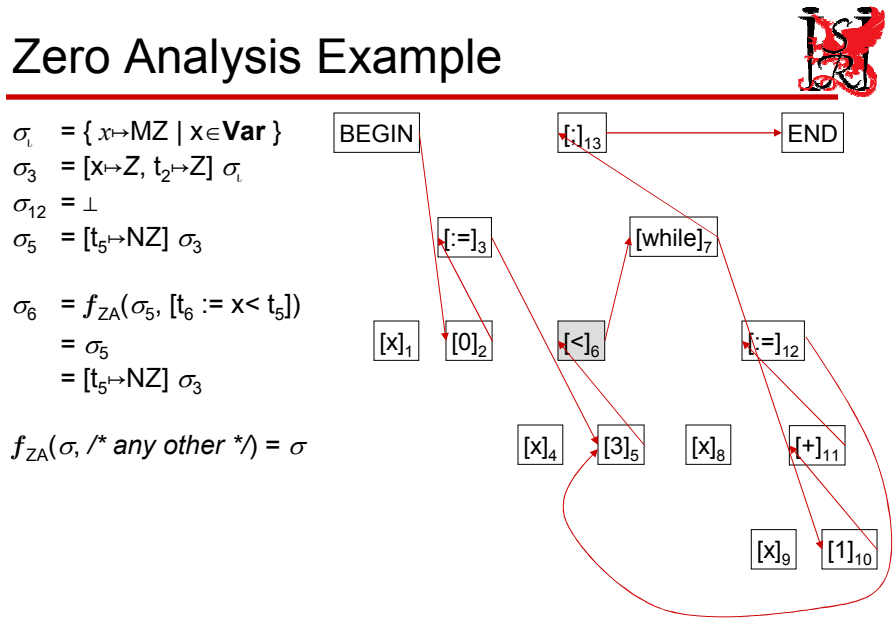
σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }
σ_3 = [x$\mapsto$Z, $t_2 \mapsto$Z] σ_ι
σ_{12} = $\bot$

σ_5 = $f_{ZA}(\sigma_3 \sqcup \sigma_{12}, [t_5 := 3])$
= $f_{ZA}(\sigma_3 \sqcup \bot, [t_5 := 3])$
= $f_{ZA}(\sigma_3, [t_5 := 3])$
= $[t_5 \mapsto NZ]\ \sigma_3$

$f_{ZA}(\sigma, [x := n])$ =
if n==0
then $[x \mapsto Z]\sigma$
else $[x \mapsto NZ]\sigma$



Zero Analysis Example

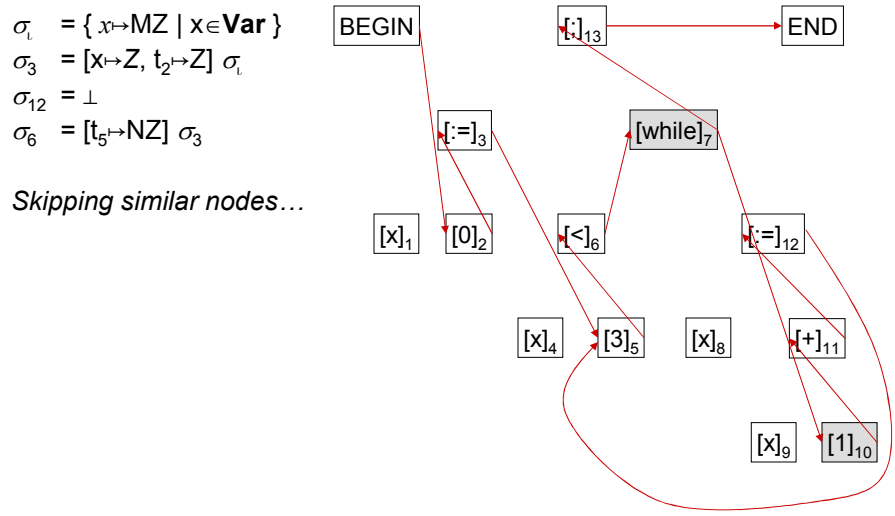


σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }
σ_3 = [x$\mapsto$Z, $t_2 \mapsto$Z] σ_ι
σ_{12} = $\perp$
σ_5 = [$t_5 \mapsto$NZ] σ_3

σ_6 = $f_{ZA}(\sigma_5, [t_6 := x < t_5])$
= σ_5
= [$t_5 \mapsto$NZ] σ_3

$f_{ZA}(\sigma$, /* *any other* */) = σ

Zero Analysis Example



σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }
σ_3 = [x$\mapsto$Z, $t_2 \mapsto$Z] σ_ι
σ_{12} = $\perp$
σ_6 = [$t_5 \mapsto$NZ] σ_3

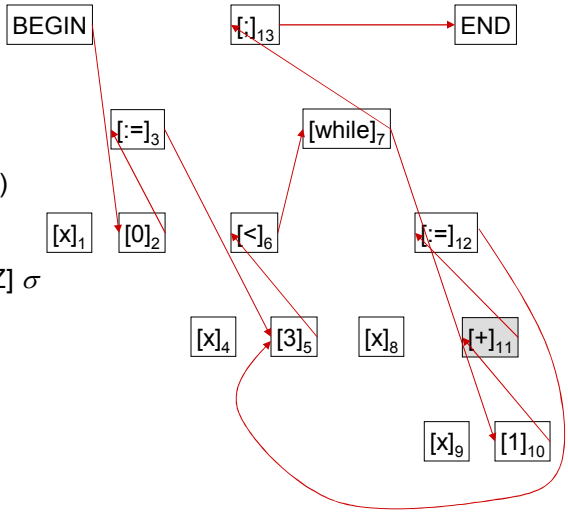
Skipping similar nodes…

Zero Analysis Example

σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }
σ_3 = [x$\mapsto$Z, $t_2 \mapsto$Z] σ_ι
σ_{12} = $\perp$
σ_{10} = [$t_{10} \mapsto$NZ,…] σ_3

σ_{11} = $f_{ZA}(\sigma_{10}, [t_{11} := x + t_{10}])$
= [$t_{11} \mapsto$MZ] σ_{10}

$f_{ZA}(\sigma, [x := y \; op \; z]) = [x \mapsto MZ]\ \sigma$

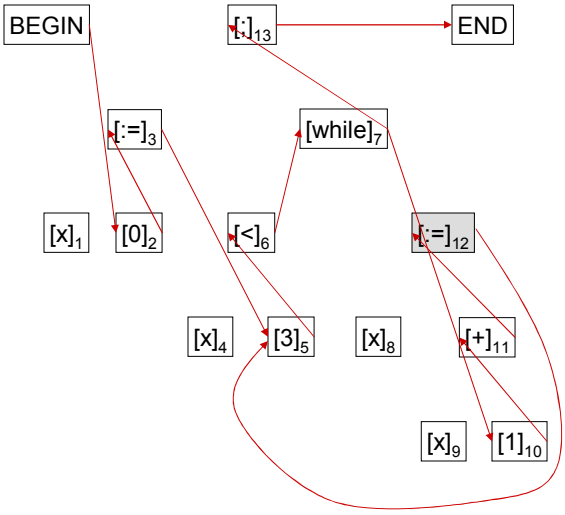


Zero Analysis Example

σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }
σ_3 = [x$\mapsto$Z, $t_2 \mapsto$Z] σ_ι
σ_{12} = $\perp$
σ_{11} = [$t_{10} \mapsto$NZ,$t_{11} \mapsto$MZ,…]σ_3

σ_{12} = $f_{ZA}(\sigma_{11}, [x := t_{11}])$
= [x$\mapsto \sigma_{11}(t_{11})$] σ_{11}
= [x$\mapsto$MZ] σ_{11}
= [x$\mapsto$MZ,…] σ_3

$f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)]\ \sigma$

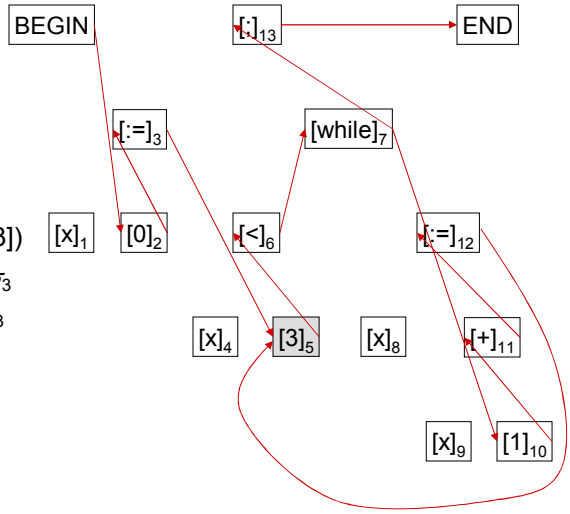


Zero Analysis Example

σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }
σ_3 = [x$\mapsto$Z, $t_2 \mapsto$Z] σ_ι
σ_{12} = [x$\mapsto$MZ, …] σ_3

σ_5 = $f_{ZA}(\sigma_3 \sqcup \sigma_{12}, [t_5 := 3])$
= $f_{ZA}([x \mapsto MZ]\sigma_3, [t_5 := 3])$
= $[t_5 \mapsto NZ]\ [x \mapsto MZ, \ldots]\sigma_3$
= $[t_5 \mapsto NZ, x \mapsto MZ, \ldots]\ \sigma_3$

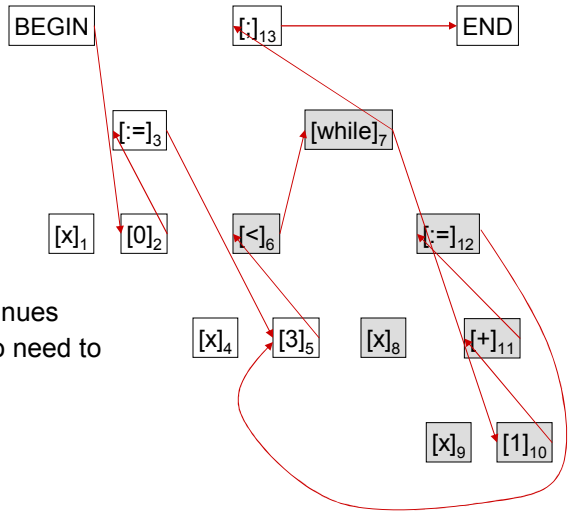
$f_{ZA}(\sigma, [x]_k) = [t_k \mapsto \sigma(x)]\ \sigma$



Zero Analysis Example

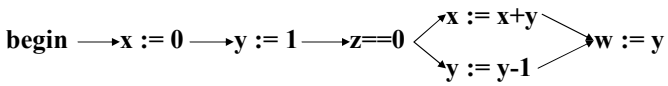
σ_ι = { $x \mapsto$ MZ | x$\in$**Var** }
σ_3 = [x$\mapsto$Z, $t_2 \mapsto$Z] σ_ι
σ_{12} = [x$\mapsto$MZ,…] σ_3

Propagation of x$\mapsto$MZ continues
σ_{12} does not change, so no need to iterate again



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.



- 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

```
worklist = new Set();
for all node indexes i do
    results[i] = ⊥A;
results[entry] = ιA;
worklist.add(all nodes);

while (!worklist.isEmpty()) do
    i = worklist.pop();
    before = ⊔k∈pred(i) results[k];
    after = fA(before, node(i));
    if (!(after ⊑ results[i]))
        results[i] = after;
        for all k∈succ(i) do
            worklist.add(k);
```

Ok to just add entry node if flow functions cannot return $\bot_A$ (examples will assume this)

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

Example of Worklist

$[a := 0]_1$

$[b := 0]_2$

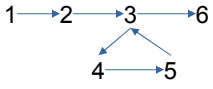
while $[a < 2]_3$ do

  $[b := a]_4$;

  $[a := a + 1]_5$;

$[a := 0]_6$

Control Flow Graph



Position	Worklist	a	b
0	1	MZ	MZ
1	2	Z	MZ
2	3	Z	Z
3	4,6	Z	Z
4	5,6	Z	Z
5	3,6	MZ	Z
3	4,6	MZ	Z
4	5,6	MZ	MZ
5	3,6	MZ	MZ
3	4,6	MZ	MZ
4	6	MZ	MZ
6		Z	MZ

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
```

Position	Worklist	x	y	w
0				

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
    - $O(n^3)$ 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