ESC/Java Wrap-up

17-654/17-765
Analysis of Software Artifacts
Jonathan Aldrich

ESC/Java's Limitations

- Does not check for some errors
  - Infinite loops, arithmetic overflow
  - Functional properties not stated by user
  - Non-functional properties
- May miss some errors
  - Unsound: describes an analysis that can miss errors
  - Only checks one iteration of loops
  - @modifies is unchecked
  - Assumptions about invariants in referred-to objects
  - Several others as well!

Loops in ESC/Java

- The loop:

```
//@ loop_invariant E;
while (B) {
  S
}
```

- Is treated as:

```
//@ assert E;
if (B) {
  S
  //@ assert E;
  //@ assume !B;
}
```

- Can optionally increase # iterations with *-loop n*

ESC/Java's Limitations (con't)

- May report false positives
  - Often can be solved with an extra precondition or invariant
  - Spurious warnings can also be disabled

ESC/Java Tradeoffs

- Attempts to automate Hoare-logic style checking
- Benefits
  - Easier than manual proof
- Drawbacks
  - Unsound
  - Still quite labor-intensive
- Applicability
  - Checking of critical code
    - When it's worth the extra effort to get it right
    - When you can't do a complete Hoare-logic proof
  - Still must use other analysis techniques
    - ESC/Java is unsound
    - The spec must also be validated!

Dataflow Analysis

17-654/17-765
Analysis of Software Artifacts
Jonathan Aldrich

Overview: Analyses We've Seen

- AST walker analyses
  - e.g. unread variable
  - Very approximate, very local
    - Misses case where var is written then never read again
- 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
- 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;
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-1;
    z := 5;
```

$\sigma=[x \mapsto \alpha_{ZI}(10)]$

Zero Analysis Example

σ=[]

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

$\sigma=[x \mapsto NZ]$

$\sigma=[x \mapsto NZ, y \mapsto \sigma(x)]$

Zero Analysis Example

Program	State
	σ=[]
x := 10;	σ=[x↦NZ]
y := x;	σ=[x↦NZ,y↦NZ]
z := 0;	σ=[x↦NZ,y↦NZ,z↦α_{ZI}(0)]
while y > -1 do	
x := x / y;	
y := y-1;	
z := 5;	

Zero Analysis Example

Program	State
	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

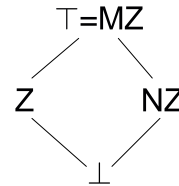
- Create an AST walker analysis
- DivByZeroVisitor
  - visit(InfixExpression b)
    - if (b.getOperator() == DIVIDE)
      - Element e = zeroAnalysis.getResultsBefore(b, b.getRightOperand())
      - if (e == Z)
        - report_error("divide by zero error", b)
      - if (e == MZ)
        - report_warning("possible divide by zero", b)

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

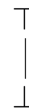
Lattice

- A lattice is a tuple (L, ⊑, ⊔, ⊥, ⊤)
  - L is a set of abstract elements
  - ⊑ is a partial order on L
    - Means *at least as precise as*
  - ⊔ is the least upper bound of two elements
    - Must exist for every two elements in L
    - Used to merge two abstract values
  - ⊥ (bottom) is the least element of L
    - Means we haven't yet analyzed this yet
    - Will become clear later
  - ⊤ (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 ⊥ to ⊤ should be finite
    - We'll see why later



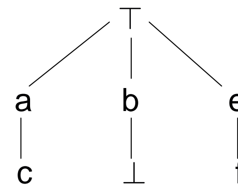
Is this a lattice?

- A lattice is a tuple (L, ⊑, ⊔, ⊥, ⊤)
  - L is a set of abstract elements
  - ⊑ is a partial order on L
  - ⊔ is the least upper bound of two elements
    - must exist for every two elements in L
  - ⊥ (bottom) is the least element of L
  - ⊤ (top) is the greatest element of L
- Yes!



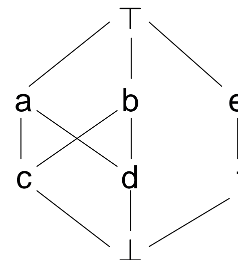
Is this a lattice?

- A lattice is a tuple (L, ⊑, ⊔, ⊥, ⊤)
  - L is a set of abstract elements
  - ⊑ is a partial order on L
  - ⊔ is the least upper bound of two elements
    - must exist for every two elements in L
  - ⊥ (bottom) is the least element of L
  - ⊤ (top) is the greatest element of L
- No!
  - No bottom element
    - ⊥ is not least in the lattice order
    - It is mis-named



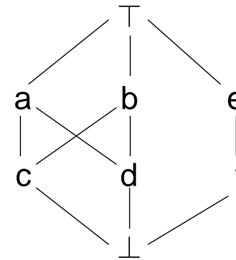
Is this a lattice?

- A lattice is a tuple (L, ⊑, ⊔, ⊥, ⊤)
  - L is a set of abstract elements
  - ⊑ is a partial order on L
  - ⊔ is the least upper bound of two elements
    - must exist for every two elements in L
  - ⊥ (bottom) is the least element of L
  - ⊤ (top) is the greatest element of L



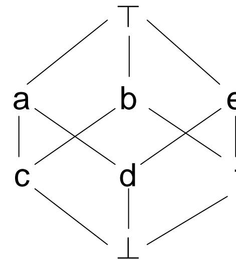
Definition: Least Upper Bounds

- x ⊔ y = z iff
  - z is an upper bound of x and y
    - x ⊑ z and y ⊑ z
  - z is the least such bound
    - ∀w∈ *L* such that x ⊑ w and y ⊑ w we have z ⊑ w
- Also called a join
- Not a lattice
  - What is c ⊔ d?
  - a, b, and ⊤ are upper bounds
    - Assume ⊑ is transitive
  - None is least upper bound



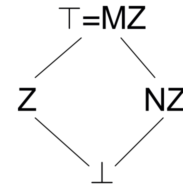
Is this a lattice?

- A lattice is a tuple (*L*, ⊑, ⊔, ⊥, ⊤)
  - *L* is a set of abstract elements
  - ⊑ is a partial order on *L*
  - ⊔ is the least upper bound of two elements
    - must exist for every two elements in *L*
  - ⊥ (bottom) is the least element of *L*
  - ⊤ (top) is the greatest element of *L*
- Yes!



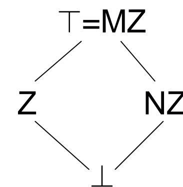
Zero Analysis Lattice

- 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
    - $\forall \mathcal{X} .\ \mathcal{X} \sqsubseteq \top$



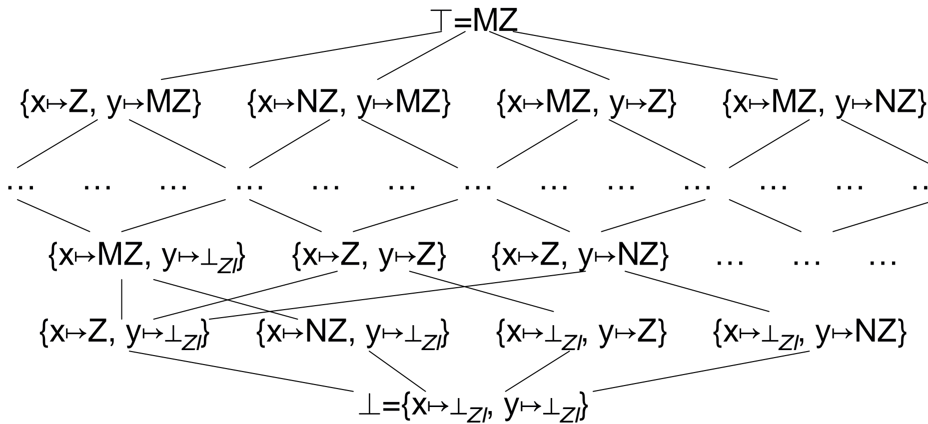
Zero Analysis Lattice

- 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 $\forall x \in$ **Var** . $\sigma_1(x) \sqsubseteq_{ZI} \sigma_2(x)$
  - $\sigma_1 \sqcup_Z \sigma_2 = \{ x \mapsto \sigma_1(x) \sqcup_{ZI} \sigma_2(x) \mid x \in$ **Var** }
  - $\bot = \{ x \mapsto \bot_{ZI} \mid x \in$ **Var** }
  - $\top = \{ x \mapsto \top_{ZI} \mid x \in$ **Var** } = { x ↦ MZ | x∈ **Var** }
  - Can produce a tuple lattice from *any* base lattice
    - Just define as above



Tuple Lattices Visually

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



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