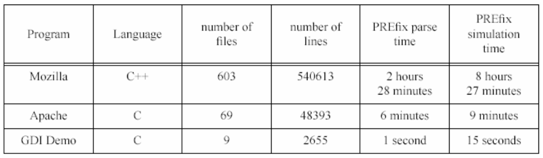
PREfix

(continued)

Reading: ***A Static Analyzer for Finding Dynamic Programming Errors***

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

PREfix Scaleability

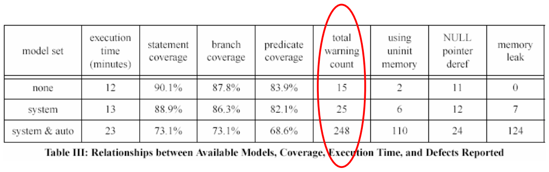


Program	Language	number of files	number of lines	PREfix parse time	PREfix simulation time
Mozilla	C++	603	540613	2 hours 28 minutes	8 hours 27 minutes
Apache	C	69	48393	6 minutes	9 minutes
GDI Demo	C	9	2655	1 second	15 seconds

Table I: Performance on Sample Public Domain Software

- Analysis cost = 2x-5x build cost
  - Scales linearly
    - Probably due to fixed cutoff on number of paths

Value of Interprocedural Analysis

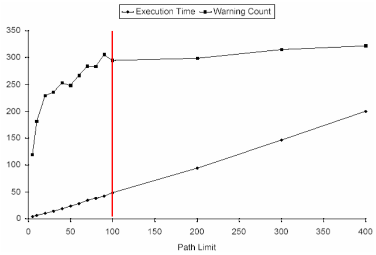


model set	execution time (minutes)	statement coverage	branch coverage	predicate coverage	total warning count	using uninit memory	NULL pointer deref	memory leak
none	12	90.1%	87.8%	83.9%	15	2	11	0
system	13	88.9%	86.3%	82.1%	25	6	12	7
system & auto	23	73.1%	73.1%	68.6%	248	110	24	124

Table III: Relationships between Available Models, Coverage, Execution Time, and Defects Reported

- 90% of errors require models (summaries)

You don't need every path



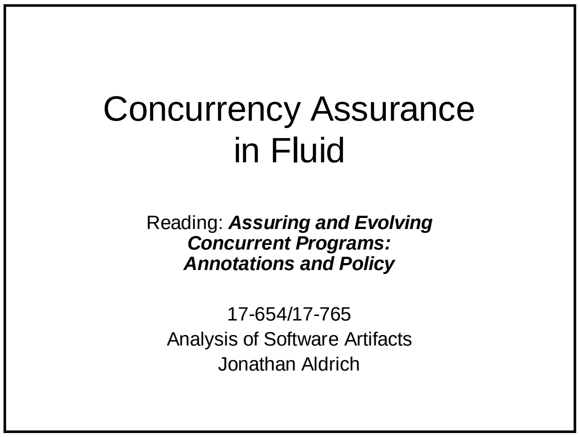
- Get most of the warnings with 100 paths

Empirical Observations

- PREfix finds errors off the main code paths
  - Main-path errors caught by careful coding and testing
- UI is essential
  - Text output is hard to read
  - Need tool to visualize paths, sort defect reports
- Noise warnings
  - Real errors that users don't care about
    - E.g., memory leaks during catastrophic shutdown

PREfix Summary

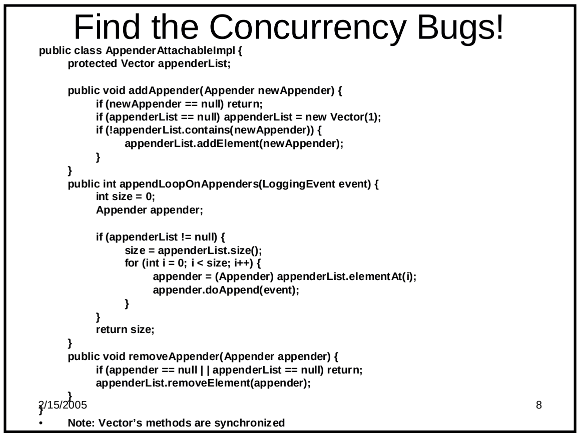
- Great tool to find errors
  - Can't guarantee that it finds them all
    - Role for other tools (e.g., Fluid)
  - Complements testing by analyzing uncommon paths
  - Focuses on low-level errors, not logic/functionality errors
    - Role for functional testing
- Huge impact
  - Used widely within Microsoft
  - Lightweight version will be part of next Visual Studio



Concurrency Assurance in Fluid

Reading: ***Assuring and Evolving Concurrent Programs: Annotations and Policy***

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

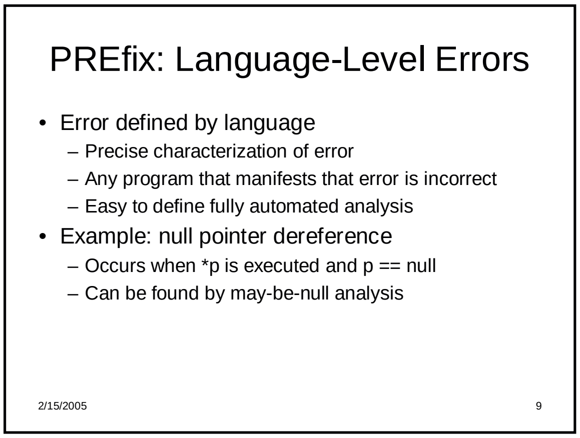


Find the Concurrency Bugs!

```
public class AppenderAttachableImpl {
        protected Vector appenderList;

        public void addAppender(Appender newAppender) {
                if (newAppender == null) return;
                if (appenderList == null) appenderList = new Vector(1);
                if (!appenderList.contains(newAppender)) {
                        appenderList.addElement(newAppender);
                }
        }
        public int appendLoopOnAppenders(LoggingEvent event) {
                int size = 0;
                Appender appender;

                if (appenderList != null) {
                        size = appenderList.size();
                        for (int i = 0; i < size; i++) {
                                appender = (Appender) appenderList.elementAt(i);
                                appender.doAppend(event);
                        }
                }
                return size;
        }
        public void removeAppender(Appender appender) {
                if (appender == null | | appenderList == null) return;
                appenderList.removeElement(appender);
        }
}
```

- Note: Vector's methods are synchronized

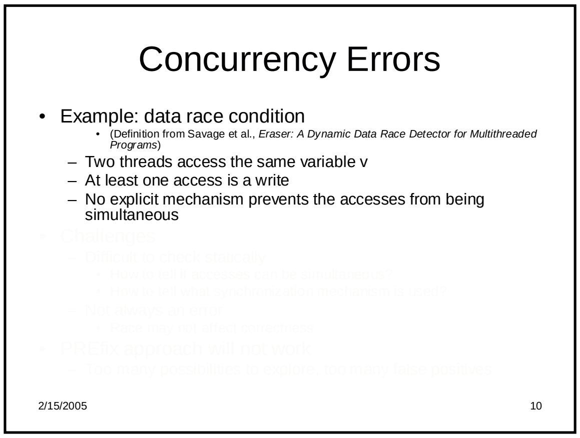
2/15/2005 8



PREfix: Language-Level Errors

- Error defined by language
  - Precise characterization of error
  - Any program that manifests that error is incorrect
  - Easy to define fully automated analysis
- Example: null pointer dereference
  - Occurs when *p is executed and p == null
  - Can be found by may-be-null analysis

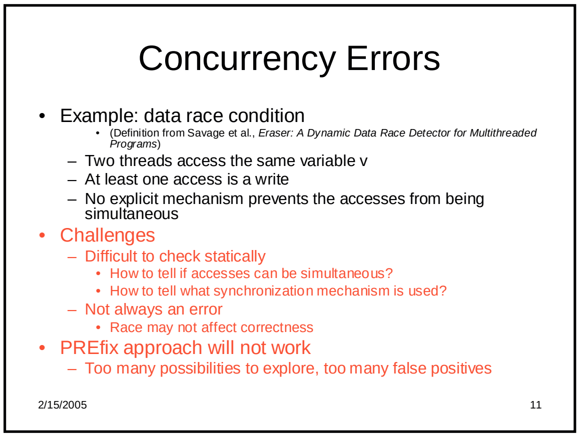
2/15/2005 9



Concurrency Errors

- Example: data race condition
    - (Definition from Savage et al., *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*)
  - Two threads access the same variable v
  - At least one access is a write
  - No explicit mechanism prevents the accesses from being simultaneous
- Challenges
  - Difficult to check statically
    - How to tell if accesses can be simultaneous?
    - How to tell what synchronization mechanism is used?
  - Not always an error
    - Race may not affect correctness
- PREfix approach will not work
  - Too many possibilities to explore, too many false positives

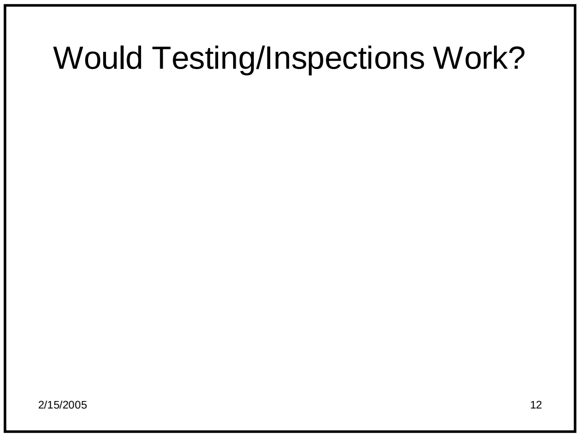
2/15/2005 10



Concurrency Errors

- Example: data race condition
    - (Definition from Savage et al., *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*)
  - Two threads access the same variable v
  - At least one access is a write
  - No explicit mechanism prevents the accesses from being simultaneous
- Challenges
  - Difficult to check statically
    - How to tell if accesses can be simultaneous?
    - How to tell what synchronization mechanism is used?
  - Not always an error
    - Race may not affect correctness
- PREfix approach will not work
  - Too many possibilities to explore, too many false positives

2/15/2005 11



Would Testing/Inspections Work?

2/15/2005 12

Would Testing/Inspections Work?

- Testing
  - Difficult because concurrency errors are non-deterministic
- Inspections
  - Concurrency errors are often non-local
    - Like errors that PREfix finds
  - Require knowledge of programmer intent

Concurrency Models

- Describe programmer’s intent
  - Data Y is protected by lock X
  - Data Z is only accessed by one thread
  - Data Y and Z must be updated together
    - To maintain some invariant
  - The race on variable V is harmless
- Can be checked against code
  - Using local static analysis

Challenge: Cost of Documenting Models

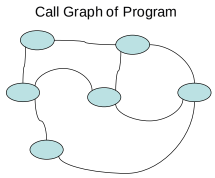
- Fluid’s approach?

Challenge: Cost of Documenting Models

- Fluid’s approach
  - Check consistency
    - No model ➔ No reported errors
  - Incrementality
    - Incremental benefit for each unit of cost
  - Usability
    - Investment in tools and usage scenarios

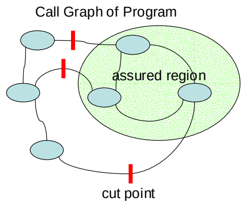
How Incrementality Works

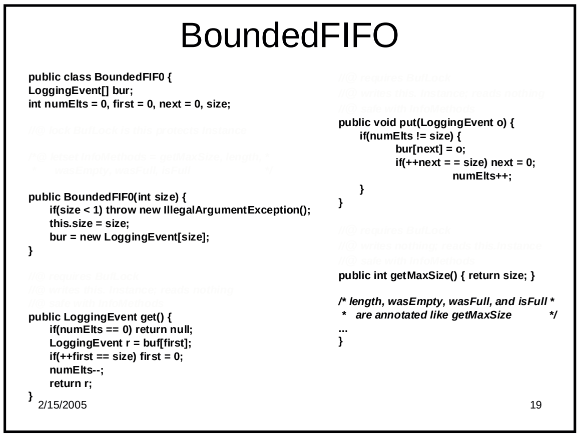
- How can one provide incremental benefit with mutual dependencies?
- Cut points
  - Method annotations partition call graph
  - Can assure property of a subgraph
  - Assurance is contingent on accuracy of trusted method annotations



How Incrementality Works

- How can one provide incremental benefit with mutual dependencies?
- Cut points
  - Method annotations partition call graph
  - Can assure property of a subgraph
  - Assurance is *contingent* on accuracy of trusted cut point method annotations



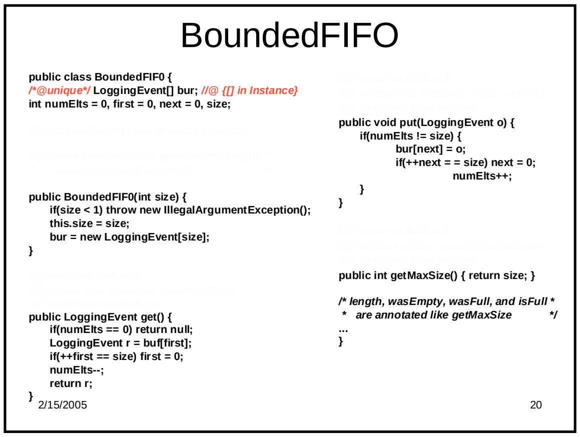


BoundedFIFO

```
public class BoundedFIF0 {
LoggingEvent[] bur;
int numElts = 0, first = 0, next = 0, size;

//@ lock BufLock is this protects Instance

/*@ letset InfoMethods = getMaxSize, length, *
 *      wasEmpty, wasFull, isFull                          */

public BoundedFIF0(int size) {
      if(size < 1) throw new IllegalArgumentException();
      this.size = size;
      bur = new LoggingEvent[size];
}

//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public LoggingEvent get() {
      if(numElts == 0) return null;
      LoggingEvent r = buf[first];
      if(++first == size) first = 0;
      numElts--;
      return r;
}
```

```
//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public void put(LoggingEvent o) {
      if(numElts != size) {
                bur[next] = o;
                if(++next = = size) next = 0;
                                numElts++;
      }
}

//@ requires BufLock
//@ writes nothing; reads this.Instance
//@ safe with InfoMethods
public int getMaxSize() { return size; }

/* length, wasEmpty, wasFull, and isFull *
 *   are annotated like getMaxSize           */
...
}
```

2/15/2005 19

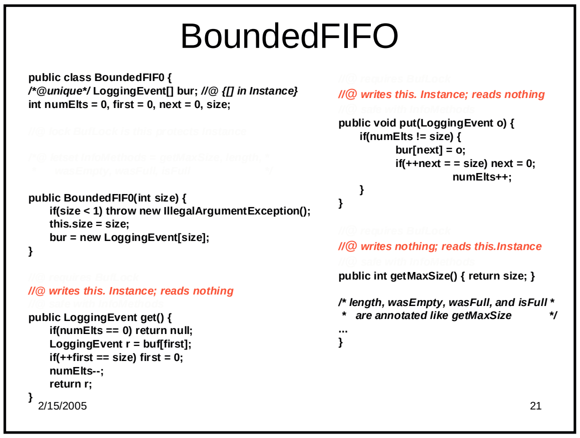


BoundedFIFO

```
public class BoundedFIF0 {
/*@unique*/ LoggingEvent[] bur; //@ {[] in Instance}
int numElts = 0, first = 0, next = 0, size;

//@ lock BufLock is this protects Instance

/*@ letset InfoMethods = getMaxSize, length, *
 *      wasEmpty, wasFull, isFull                          */

public BoundedFIF0(int size) {
      if(size < 1) throw new IllegalArgumentException();
      this.size = size;
      bur = new LoggingEvent[size];
}

//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public LoggingEvent get() {
      if(numElts == 0) return null;
      LoggingEvent r = buf[first];
      if(++first == size) first = 0;
      numElts--;
      return r;
}
```

```
//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public void put(LoggingEvent o) {
      if(numElts != size) {
                bur[next] = o;
                if(++next = = size) next = 0;
                                numElts++;
      }
}

//@ requires BufLock
//@ writes nothing; reads this.Instance
//@ safe with InfoMethods
public int getMaxSize() { return size; }

/* length, wasEmpty, wasFull, and isFull *
 *   are annotated like getMaxSize           */
...
}
```

2/15/2005 20

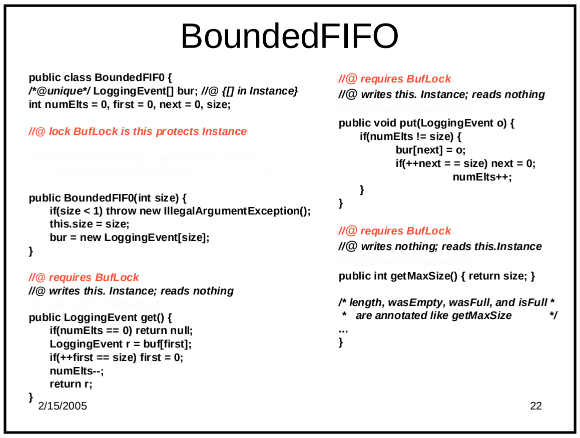


BoundedFIFO

```
public class BoundedFIF0 {
/*@unique*/ LoggingEvent[] bur; //@ {[] in Instance}
int numElts = 0, first = 0, next = 0, size;

//@ lock BufLock is this protects Instance

/*@ letset InfoMethods = getMaxSize, length, *
 *      wasEmpty, wasFull, isFull                          */

public BoundedFIF0(int size) {
      if(size < 1) throw new IllegalArgumentException();
      this.size = size;
      bur = new LoggingEvent[size];
}

//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public LoggingEvent get() {
      if(numElts == 0) return null;
      LoggingEvent r = buf[first];
      if(++first == size) first = 0;
      numElts--;
      return r;
}
```

```
//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public void put(LoggingEvent o) {
      if(numElts != size) {
                bur[next] = o;
                if(++next = = size) next = 0;
                                numElts++;
      }
}

//@ requires BufLock
//@ writes nothing; reads this.Instance
//@ safe with InfoMethods
public int getMaxSize() { return size; }

/* length, wasEmpty, wasFull, and isFull *
 *   are annotated like getMaxSize           */
...
}
```

2/15/2005 21

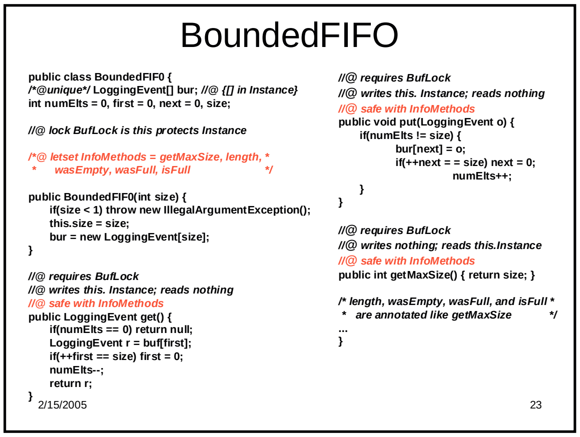


BoundedFIFO

```
public class BoundedFIF0 {
/*@unique*/ LoggingEvent[] bur; //@ {[] in Instance}
int numElts = 0, first = 0, next = 0, size;

//@ lock BufLock is this protects Instance

/*@ letset InfoMethods = getMaxSize, length, *
 *      wasEmpty, wasFull, isFull                          */

public BoundedFIF0(int size) {
      if(size < 1) throw new IllegalArgumentException();
      this.size = size;
      bur = new LoggingEvent[size];
}

//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public LoggingEvent get() {
      if(numElts == 0) return null;
      LoggingEvent r = buf[first];
      if(++first == size) first = 0;
      numElts--;
      return r;
}
```

```
//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public void put(LoggingEvent o) {
      if(numElts != size) {
                bur[next] = o;
                if(++next = = size) next = 0;
                                numElts++;
      }
}

//@ requires BufLock
//@ writes nothing; reads this.Instance
//@ safe with InfoMethods
public int getMaxSize() { return size; }

/* length, wasEmpty, wasFull, and isFull *
 *   are annotated like getMaxSize           */
...
}
```

2/15/2005 22

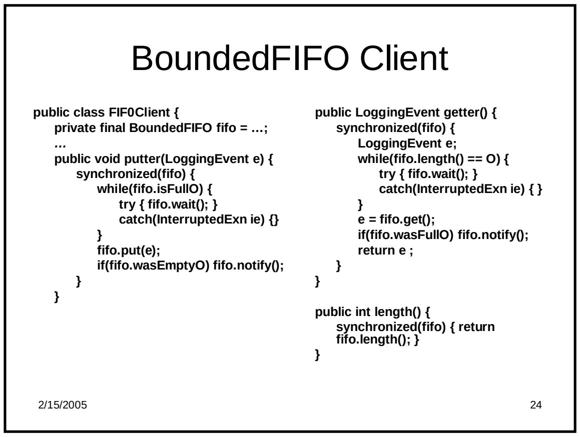


BoundedFIFO

```
public class BoundedFIF0 {
/*@unique*/ LoggingEvent[] bur; //@ {[] in Instance}
int numElts = 0, first = 0, next = 0, size;

//@ lock BufLock is this protects Instance

/*@ letset InfoMethods = getMaxSize, length, *
 *      wasEmpty, wasFull, isFull                          */

public BoundedFIF0(int size) {
      if(size < 1) throw new IllegalArgumentException();
      this.size = size;
      bur = new LoggingEvent[size];
}

//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public LoggingEvent get() {
      if(numElts == 0) return null;
      LoggingEvent r = buf[first];
      if(++first == size) first = 0;
      numElts--;
      return r;
}
```

```
//@ requires BufLock
//@ writes this. Instance; reads nothing
//@ safe with InfoMethods
public void put(LoggingEvent o) {
      if(numElts != size) {
                bur[next] = o;
                if(++next = = size) next = 0;
                                numElts++;
      }
}

//@ requires BufLock
//@ writes nothing; reads this.Instance
//@ safe with InfoMethods
public int getMaxSize() { return size; }

/* length, wasEmpty, wasFull, and isFull *
 *   are annotated like getMaxSize           */
...
}
```

2/15/2005 23



BoundedFIFO Client

```
public class FIF0Client {
     private final BoundedFIFO fifo = …;
     …
     public void putter(LoggingEvent e) {
          synchronized(fifo) {
               while(fifo.isFullO) {
                    try { fifo.wait(); }
                    catch(InterruptedExn ie) {}
               }
               fifo.put(e);
               if(fifo.wasEmptyO) fifo.notify();
          }
     }
```

```
public LoggingEvent getter() {
     synchronized(fifo) {
          LoggingEvent e;
          while(fifo.length() == O) {
               try { fifo.wait(); }
               catch(InterruptedExn ie) { }
          }
          e = fifo.get();
          if(fifo.wasFullO) fifo.notify();
          return e ;
     }
}

public int length() {
     synchronized(fifo) { return
     fifo.length(); }
}
```

2/15/2005 24

Lock Analysis, Fluid Style

Lattice

$\top$ = unknown

$\bot$ = locked

- Lattice is a tuple of custom lattices
  - One for each variable in the program
- Forward analysis
- Injected tuple ι = { $\bot$ for x if /* *@requires x */* annotation, $\top$ otherwise}
- Simple transfer functions *(σ is input data flow value)*
  - f^{LA}([synchronized(x) { S }],σ)= σ [x ↦ $\bot$] *// only for analysis of S*
    = σ *// for subsequent statements*
  - f^{LA}([x := f(e)],σ) = σ *// nothing special at method calls*
  - f^{LA}(S,σ) = σ *// for all other statements*
- Report errors
  - At $[y := f(e)]^{\ell}$, if /* *@requires x */* in *annotations(f)* and LA(ℓ,*x*) = $\top$
  - If y is used in ℓ, /* *@lock x protects y */* is in scope and LA (ℓ,*x*) = $\top$

Uniqueness Analysis

Lattice

$\top$ = unknown

$\bot$ = unique

- Lattice is a tuple of custom lattices
  - One for each variable in the program
- Forward analysis
- Injected tuple ι = { $\bot$ for x if /* *@unique x */* annotation, $\top$ otherwise}
- Example transfer functions *(σ is input data flow value)*
  - $f^{UA}([x := y]^{\ell},\sigma)$ = σ [x ↦ $\top$, y ↦ $\top$] *// if y ∈ LV(ℓ)*
    = σ [x ↦ σ[x]] *// if y ∉ LV(ℓ)*
  - $f^{UA}([x := f(y)]^{\ell},\sigma)$ = σ [x ↦ *annot(f)*, y ↦ $\top$] *// if y ∈ LV(ℓ)*
    // and annot(arg(f))≠borrowed
    = σ [x ↦ *annot(f)*] *// otherwise*
- Report errors
  - At $[x := f(y)]^{\ell}$, if /* *@unique arg */* in *annotations(f)* and UA(ℓ,*y*) = $\top$
  - If y is annotated /* *@unique */* but UA(ℓ,*x*) = $\top$ for some statement ℓ

Summary: PREfix vs. Fluid

- PREfix
  - Finds language-level errors
  - Fully automatic
  - Interprocedural
  - Goal: find bugs
- Fluid
  - Finds concurrency errors
  - Requires annotations
  - Intra-procedural with cut points
  - Goal: ensure absence of certain kinds of bugs