Lecture 08(b) – Typestate Verification

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav

SAFE

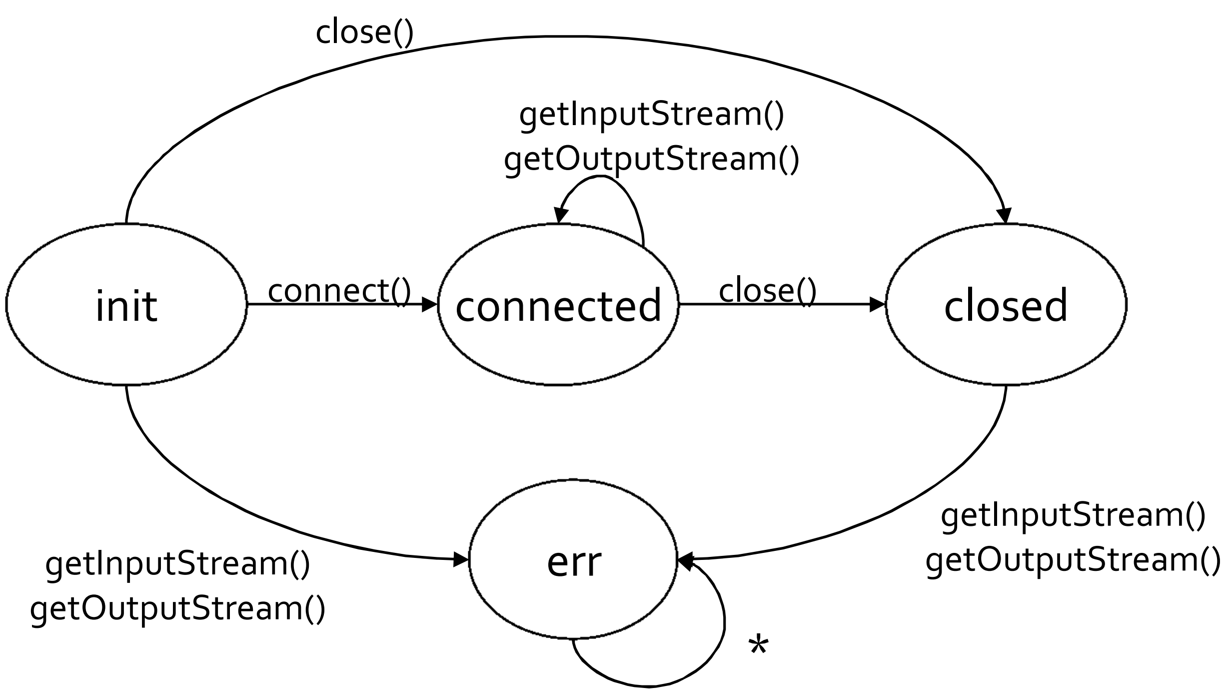
EFFECTIVE TYPESTATE VERIFICATION IN THE PRESENCE OF ALIASING

Joint work with Nurit Dor, Stephen Fink, Satish Chandra, Marco Pistoia, Ganesan Ramalingam, Sharon Shoham, Greta Yorsh

Motivation

- **Application Trend: Increasing number of libraries and APIs**
  - Non-trivial restrictions on permitted sequences of operations
- **Typestate:** Temporal safety properties
  - What sequence of operations are permitted on an object?
  - Encoded as DFA

  e.g. "Don't use a Socket unless it is connected"



Goal

- Typestate Verification: statically ensure that no execution of a program can transition to err
  - Sound[1] (excluding concurrency)
  - Precise enough[2] (reasonable number of false alarms)
  - Scalable[3] (handle programs of realistic size)

1 In the real world, some other caveats apply

2 we'll get back to that

3 relatively speaking

Challenges

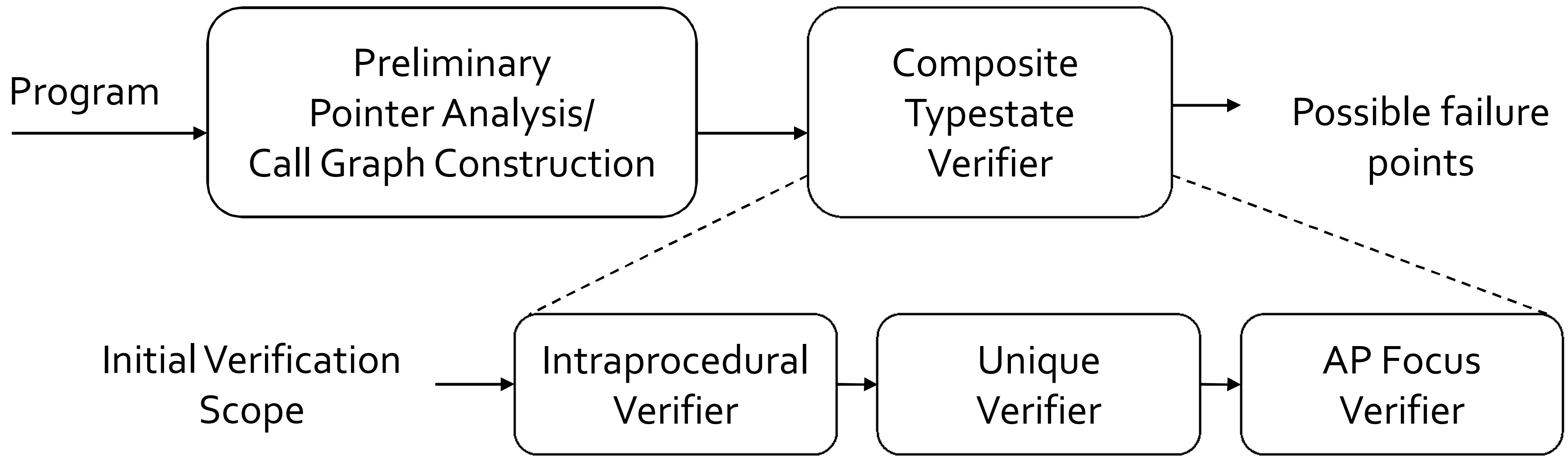
```
class SocketHolder {  Socket s;   }
Socket makeSocket() { return new Socket(); // A }
open(Socket l) {
   l.connect();
}
talk(Socket s) {
  s.getOutputStream()).write("hello");
}

main() {
  Set<SocketHolder> set = new HashSet<SocketHolder>();
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();
     set.add(h)
  }
  for (Iterator<SocketHolder> it = set.iterator(); …) {
     Socket g = it.next().s;
     open(g);
     talk(g);
   }
}
```

- Flow-Sensitivity
- Interprocedural flow
- Context-Sensitivity
- * Non-trivial Aliasing *
- Path Sensitivity
- Full Java Language
  - Exceptions, Reflection, …
- Big programs

Main Ideas

- Flow-sensitive, context-sensitive interprocedural analysis
  - **Abstract domains combine typestate and pointer information**
    - More precise than 2-stage approach
    - Focus expensive effort where it matters
  - ***Staging:* Sequence of abstractions of varying cost/precision**
    - Inexpensive early stages reduce work for later expensive stages
  - **Techniques for *inexpensive* strong updates (*Uniqueness*, *Focus*)**
    - Much cheaper than typical shape analysis
    - More precise than usual “scalable” analyses
- Results
  - Flow-sensitive functional IPA with sophisticated alias analysis on ~100KLOC in 10 mins. Scales up to ~500KLOC.
  - Verify ~92% of potential points of failure (PPF) as safe

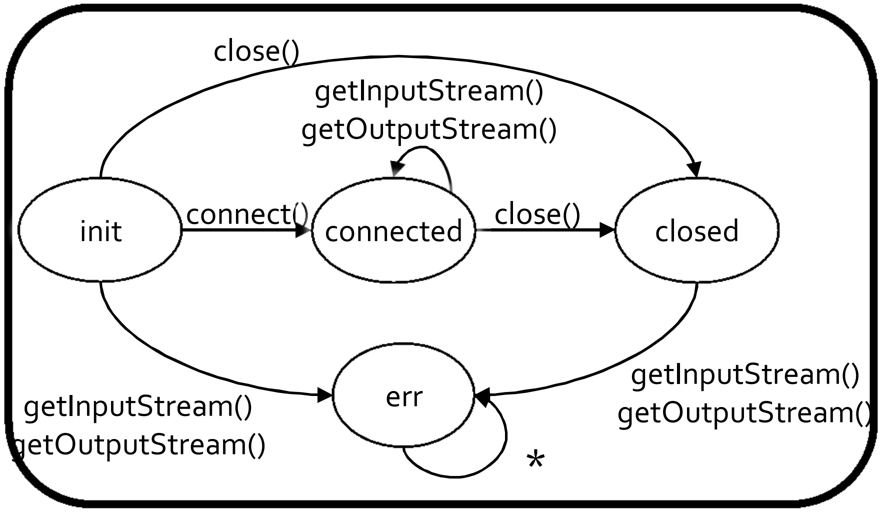
Analysis Overview



Dataflow Analysis

- Sound, abstract representation of program state
- Flow-sensitive propagation of abstract state
- Context-sensitive: Tabulation Solver [Reps-Horwitz-Sagiv 95]

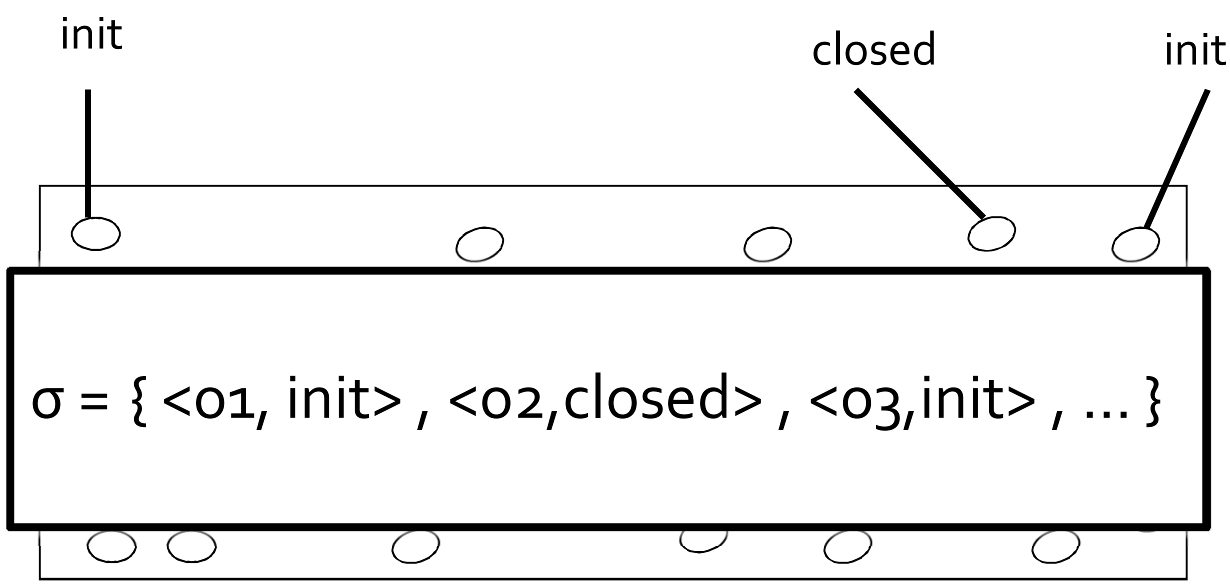
(Instrumented) Concrete Semantics



$[\![\text{x.connect()}]\!]$

$x \rightarrow o_1$

$\langle o_1, init \rangle \longrightarrow \langle o_1, connected \rangle$



Instrumented Concrete Semantics

$L^\natural \subseteq objects^\natural$
$v^\natural \in Val = objects^\natural \cup \{ null \}$
$\rho^\natural \in Env = VarId \rightarrow Val$
$h^\natural \in Heap = objects^\natural \times FieldId \rightarrow Val$

$state^\natural = <L^\natural, \rho^\natural, h^\natural> \in 2^{objects^\natural} \times Env \times Heap$

Typestate property DFA $<\Sigma, Q, \delta, init, Q \setminus \{err\}>$

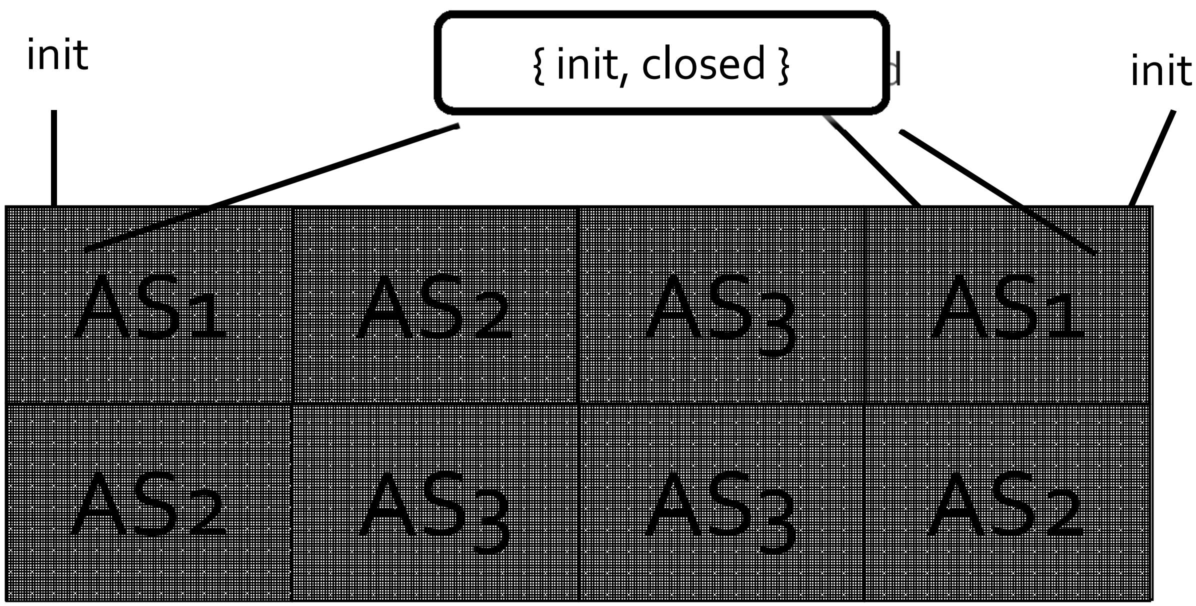
$typestate^\natural: L^\natural \rightarrow Q$

$istate^\natural = <L^\natural, \rho^\natural, h^\natural, typestate^\natural>$

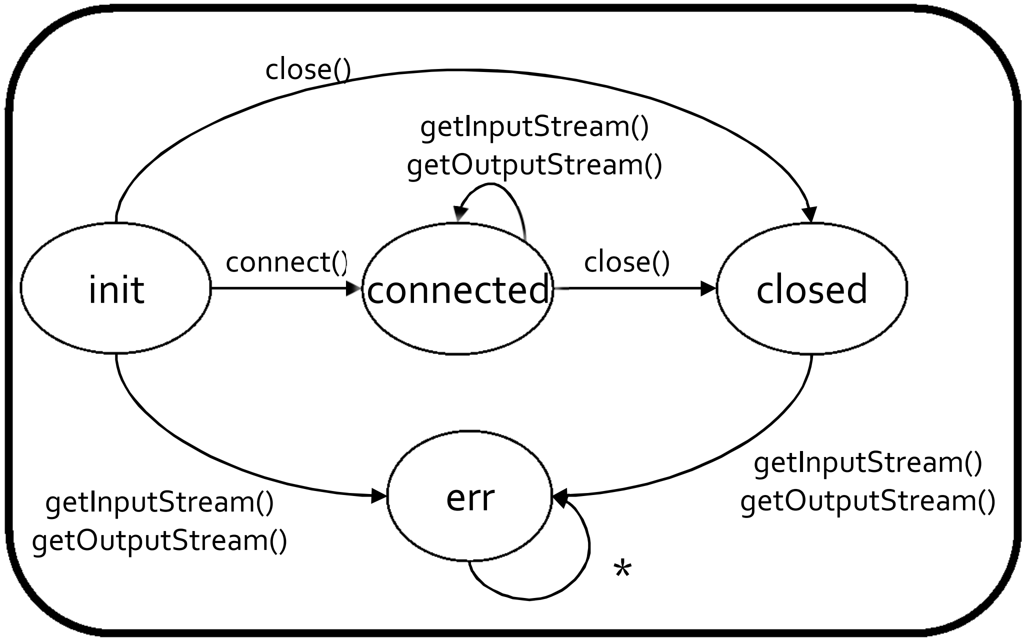
Abstract State

$\sigma = \{ \langle o1, init \rangle , \langle o2, closed \rangle , \langle o3, init \rangle , \ldots \}$

$\sigma^{\#} = \{ \langle AS1, init \rangle , \langle AS1, closed \rangle \}$



Base Abstraction




```
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
  Socket s = new Socket(); //S
  open(s);
  talk(s);
  dispose(s);
}
```

<S, init>

<S, init> , <S, connected>

<S, init> , <S, connected>, <S, err> ×

Base

Unique Abstraction

Abstract State := { < Abstract Object, TypeState, UniqueBit> }

- "UniqueBit" ≈ "∃ exactly one concrete instance of abstract object"
- Allows strong updates
- Works sometimes (80% in our experience)

```
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
  Socket s = new Socket(); //S   <S, init, U>
  open(s);
                                 <S, connected, U>
  talk(s);
  dispose(s);                    <S, connected, U> ✓
}
```

Unique Abstraction

```
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
 while (…) {
  Socket s =new Socket();//S
  open(s);
  talk(s);
  dispose(s);
 }
}
```

while (…) {		*<S, closed, U>*
Socket s =new Socket();//S	*<S, init, U>*	*<S, closed, ¬U> <S, init, ¬U >*
open(s);	*<S, connected, U>*	*<S, closed, ¬U> <S, init, ¬U > <S, connected, ¬U>*
talk(s);	*<S, connected, U>*	*<S, err, ¬U> × ….*
dispose(s);	*<S, closed, U>*	

Object liveness analysis to the rescue

- Preliminary live analysis oracle
- On-the-fly removal unreachable configurations using liveness info

Unique Abstraction

```
class SocketHolder {Socket s; }
Socket makeSocket() { return new Socket(); // A }
open(Socket s) {
  s.connect();
}
talk(Socket s) {
  s.getOutputStream()).write("hello");
}
dispose(Socket s) { h.s.close(); }
main() {
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();
     Socket s = makeSocket();
     open(h.s);
     talk(h.s);
     dispose(h.s);
     open(s);
     talk(s);
  }
}
```

<A, init, U>

<A, init , ¬U >

<A, init, ¬U > <A, connected, ¬U >

<A, err, ¬U> × ….

Access Path Based Abstractions

Access Path Must ***{ < Abstract Object, TypeState, UniqueBit, MustSet, MayBit> }***

MustSet := set of symbolic access paths (x.f.g….) that *must* point to the object
MayBit := "must set is incomplete. Must fall back to may-alias oracle"

- Strong Updates allowed for *e.op()* when *e* $\in$ *Must* or unique logic allows

Access Path Focus ***{ < Abstract Object, TypeState, UniqueBit, MustSet, MayBit, MustNotSet> }***

MustNotSet **:=** set of symbolic access paths that *must not* point to the object

Focus operation when interesting things happen

- **generate 2 tuples,** a ***Must*** information case and a ***MustNot*** information case

- **Only track access paths to "interesting" objects**
- **Sound flow functions to *lose precision in MustSet, MustNotSet***
  - Allows k-limiting. Crucial for scalability.

Access Path Based Abstractions

Statement	**Resulting abstract tuples for incoming** **$\langle$o,unique, typestate,AP_M,May,$AP_{MN}\rangle$**
e.op()	$\langle$o,unique,δ(typestate,op),AP_M,May,$AP_{MN}\rangle$ if $e \notin AP_{MN}$ and ($e \in AP_M$ or May) $\langle$o,unique,typestate,AP_M,May,$AP_{MN}\rangle$ if $e \in AP_{MN}$ or ($e \notin AP_M$ $\wedge$ $\neg$(unique $\wedge$ pt(e) = o) $\wedge$ May)

Access Path Based Abstractions

Statement	Resulting abstract tuples for incoming $\langle$o,unique, typestate,AP_M,May,$AP_{MN}\rangle$
v = new T()	$\langle$o,false,typestate,$AP_M - \{v.\pi \mid \pi \in \Pi\}$,May,$AP_{MN} \cup \{ v \}$ $\rangle$ $\langle$o,false,init,{v} ,false,ø$\rangle$
v = null	$AP'_M := AP_M - \{v.\pi \mid \pi \in \Pi\}$ $AP'_{MN} := AP_{MN} \cup \{ v \}$
v.f = null	$AP'_M := AP_M - \{e'.f.\pi \mid mayAlias(e',v), \pi \in \Pi\}$ $AP'_{MN} := AP_{MN} \cup \{ v.f \}$
v = e	$AP'_M := AP_M \cup \{v.\pi \mid e.\pi \in AP_M\}$ $AP'_{MN} := AP_{MN} - \{ v.\pi \mid e.\pi \notin AP_{MN}\}$
v.f = e	$AP'_M := AP_M \cup \{v.f.\pi \mid e.\pi \in AP_M\}$ $AP'_{MN} := AP_{MN} - \{v.f \mid e \notin AP_{MN}\}$ May' := May $\vee$ ($\exists v.f.\pi \in AP'_M$ and $\exists p \in \Pi$. mayAlias(v,p) $\wedge$ $p.f.\pi \notin AP_M$)

Π = set of all possible access paths in the program

Access Path Abstraction

```
class SocketHolder {Socket s; }
Socket makeSocket() { return new Socket(); // A }
open(Socket s) {
  s.connect();
}
talk(Socket s) {
  s.getOutputStream()).write("hello");
}
dispose(Socket s) { h.s.close(); }
main() {
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();
     Socket s = makeSocket();
     open(h.s);
     talk(h.s);
     dispose(h.s);
     open(s);
     talk(s);
  }
}
```

<A, init, U, {h.s}, ¬May, {} >

<A, init , ¬U, {h.s}, ¬May, {} > …

<A, connected , ¬U, {h.s}, ¬May, {} >

✓

What can we do when we lose access-path information?

```
class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket l) {
   l.connect();
}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { h.s.close(); }
main() {
  Set<SocketHolder> set = new HashSet<SocketHolder>();
  while(…) {
     SocketHolder h = new SocketHolder();
     h.s = makeSocket();
     set.add(h);
  } ;
  for (Iterator<SocketHolder> it = set.iterator(); …) {
     Socket g = it.next().s;
     open(g);
     talk(g);
     dispose(g);
  }
}
```

<A, init, U, {h.s}, ¬May>

<A, init , U, {h.s}, May>

<A, init, ¬U, {}, May >

<A, init, ¬U, {}, May >, <A, connected, ¬U, {}, May >

<A, err, ¬U, {}, May > …

Access Paths with Focus

AS := { < Abstract Object, TypeState, Unique, Must, May, MustNot> }

- Access Path Must Abstraction +
  - ***MustNot*** **:=** set of access paths that *must not* point to the object
- **Focus** operation when "interesting" things happen
  - "case splitting"
  - ***e.op()*** **on** ***⟨A, T, u, Must, May, MustNot⟩*****, generate 2 factoids:**
    - ***⟨A, δ(T), u, Must U {e}, May, MustNot ⟩***
    - ***⟨A, T, u, Must, May, MustNot U {e} ⟩***
  - Interesting Operations
    - Typestate changes
  - Allows k-limiting. Crucial for scalability
  - Allowed to limit exponential blowup due to focus
    - Current heuristic: discard *MustNot* before each focus operation
    - TODO: More general solution ("blur", "normalization")
- Works sometimes (95.6%)

Access Path with Focus

```
class SocketHolder {  Socket s;  }
Socket makeSocket() { return new Socket(); // A }
open(Socket t) {
   t.connect();
}
talk(Socket s) {
  s.getOutputStream().w
}
dispose(Socket s) { h.s.close(); }
main() {
 Set<SocketHolder> set = new HashSet<SocketHolder>();
 while(…) {
  SocketHolder h = new SocketHolder();
  h.s = makeSocket();
  set.add(h);
 }
 for (Iterator<SocketHolder> it = set.iterator(); …) {
  Socket g = it.next().s;
  open(g);
  talk(g);
 }
}
```

<A, init, ¬U, {}, May, {} >

<A, init, ¬U, {}, May, {¬ t} >, <A, connected, ¬U, {t}, May, {}>

<A, init, ¬U, {}, May , {¬g,¬s}>, <A, connected, ¬U, {g,s}, May, {}> ✓

<A, init, U, {h.s}, ¬May, {}>

<A, init , U, {h.s}, May ,{}>

<A, init, ¬U, {}, May , {}>

<A, init, ¬U, {}, May , {¬g}>, <A, connected, ¬U, {g}, May, {}>

Access Path Focus Abstraction

{ < Abstract Object, TypeState, UniqueBit, MustSet, MayBit, MustNotSet> }

Implementation Details Matter

Sparsification

Separation (solve for each abstract object separately)

"Pruning": discard branches of supergraph that cannot affect abstract semantics

- **Reduces median supergraph size by 50X**

Preliminary
Pointer Analysis / Call Graph Construction

Details matter a lot

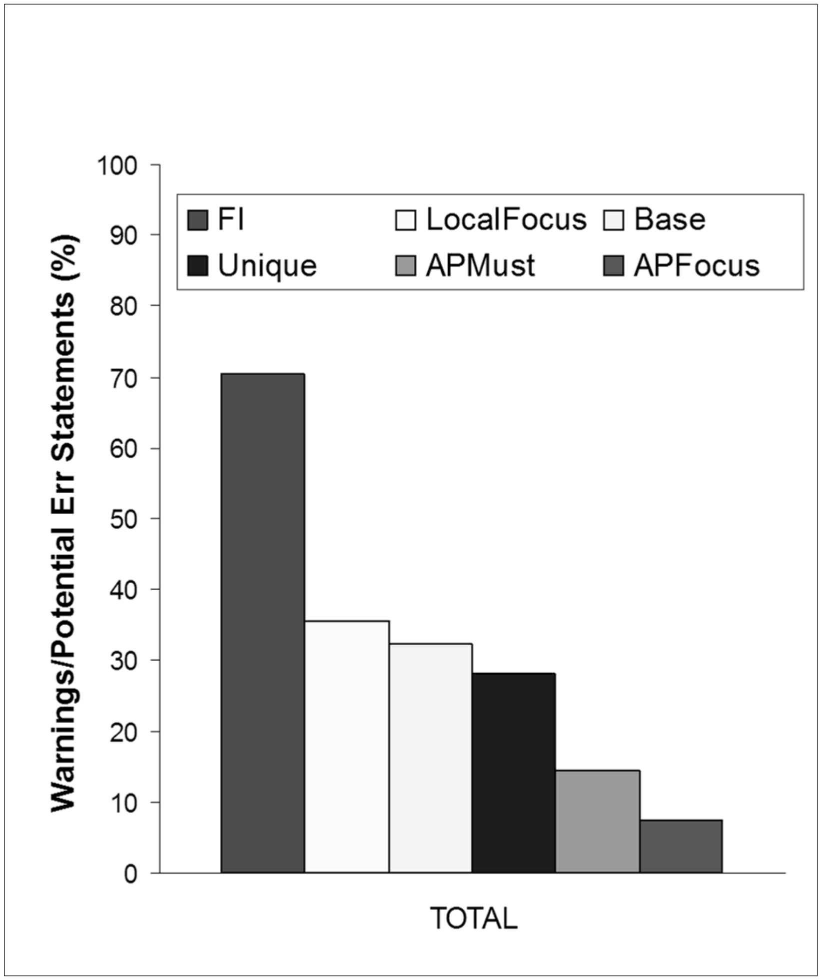
- if *context-insensitive* preliminary, stages time out, terrible precision

Current implementation:

- Subset-based, field-sensitive Andersen's
- SSA local representation
- On-the-fly call graph construction
- Unlimited object sensitivity for
  - Collections
  - Containers of typestate objects (e.g. *IOStreams*)
- One-level call-string context for some library methods
- Heuristics for reflection (e.g. Livshits et al 2005)

Precision

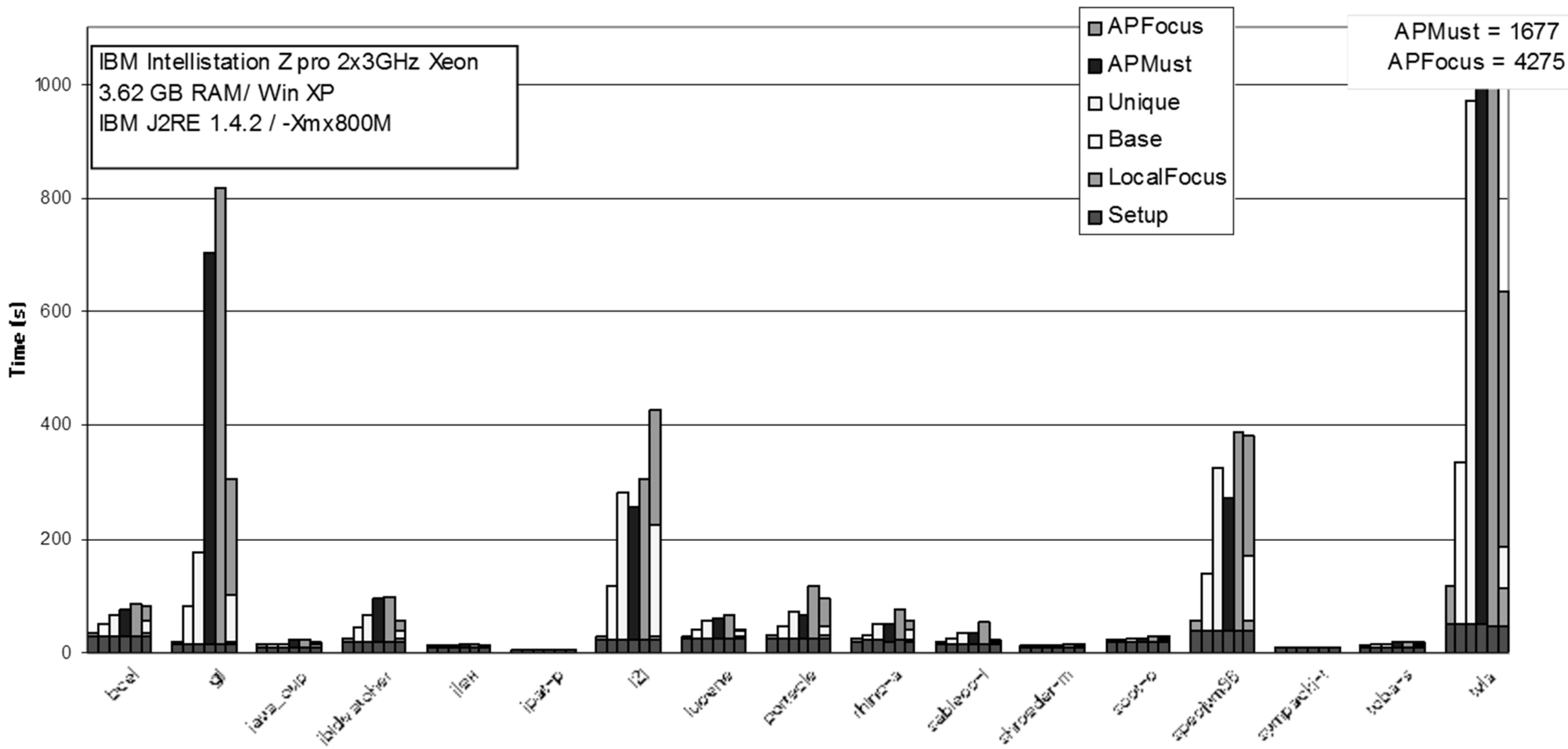
11 typestate properties from Java standard libraries
17 moderate-sized benchmarks *[~5K – 100K LOC]*



Sources of False Positives

- **Limitations of analysis**
  - Aliasing
  - Path sensitivity
  - Return values
- **Limitations of typestate abstraction**
  - Application logic bypasses DFA, still OK

Running time



Some Related Work

- ESP
  - Das *et al.* PLDI 2002
    - Two-phase approach to aliasing (unsound strong updates)
    - Path-sensitivity ("property simulation")
  - Dor *et al.* ISSTA 2004
    - Integrated typestate and alias analysis
    - Tracks overapproximation of *May* aliases

- Type Systems
  - Vault/Fugue
    - Deline and Fähndrich 04: `adoption` and `focus`
  - CQUAL
    - Foster *et al*. 02: linear types
    - Aiken *et al.* 03: `restrict` and `confine`

- Alias Analysis
  - Landi-Ryder 92, Choi-Burke-Carini 93, Emami-Ghiya-Hendren 95, Wilson-Lam 95, ….
  - Shape Analysis: Chase-Wegman-Zadeck 90, Hackett-Rugina 05, Sagiv-Reps-Wilhelm 99, …