Lecture 08(b) – Typestate Verification

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”*

```
init  connect()  connected  close()  closed

getInputStream()  getOutputStream()

init  connect()  connected  close()  closed

getInputStream()  getOutputStream()

init  connect()  connected  close()  closed

getInputStream()  getOutputStream()

err  *
```

```
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);
   }
}
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

\[ \left[ x. \text{connect}() \right] \]

\[ x \rightarrow 0, \]

\[ \langle 0, \text{init} \rangle \rightarrow \langle 0, \text{connected} \rangle \]

\[ \sigma = \{ \langle 01, \text{init} \rangle , \langle 02, \text{closed} \rangle , \langle 03, \text{init} \rangle , \ldots \} \]
Instrumented Concrete Semantics

\[ L \subseteq \text{objects} \]
\[ \nu \in \text{Val} = \text{objects} \cup \{ \text{null} \} \]
\[ \rho \in \text{Env} = \text{VarId} \rightarrow \text{Val} \]
\[ h \in \text{Heap} = \text{objects} \times \text{FieldId} \rightarrow \text{Val} \]

\[ \text{state} = \langle L, \rho, h \rangle \in 2^{\text{objects}} \times \text{Env} \times \text{Heap} \]

Typestate property DFA \( \langle \Sigma, Q, \delta, \text{init}, Q\setminus \{\text{err}\} \rangle \)

\[ \text{typestate} : L \rightarrow Q \]

\[ \text{istate} = \langle L, \rho, h, \text{typestate} \rangle \]
Abstract State

\[ \sigma = \{ <o_1, \text{init}>, <o_2, \text{closed}>, <o_3, \text{init}>, \ldots \} \]

\[ \sigma^# = \{ <\text{AS}_1, \text{init}>, <\text{AS}_1, \text{closed}> \} \]
Base Abstraction

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

```java
open(Socket s) { s.connect();}
talk(Socket s) { s.getOutputStream()).write("hello"); }
dispose(Socket s) { s.close(); }
main() {
    Socket s = new Socket(); // <S, init, U>
    open(s);                  // <S, connected, U>
    talk(s);                  // <S, connected, U>
    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, init, U>
        open(s); <S, connected, U> <S, closed, ¬U> <S, init, ¬U>
        talk(s); <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);<A, init, U>
        talk(h.s);<A, init, ¬U >
        dispose(h.s);<A, init, ¬U > <A, connected, ¬U >
        open(s);<A, err, ¬U > × ....
        talk(s);
    }
}
Access Path Based Abstractions

**Access Path Must**

\[ \{ <\text{Abstract Object, TypeState, UniqueBit, MustSet}, \text{MayBit}> \} \]

- \( \text{MustSet} := \) set of symbolic access paths (x.f.g....) that \textit{must} point to the object
- \( \text{MayBit} := \) “\textit{must set is incomplete. Must fall back to may-alias oracle}”
  - Strong Updates allowed for \( e \).\textit{op()} when \( e \in \text{Must} \) or unique logic allows

**Access Path Focus**

\[ \{ <\text{Abstract Object, TypeState, UniqueBit, MustSet, MayBit, MustNotSet}> \} \]

- \( \text{MustNotSet} := \) set of symbolic access paths that \textit{must not} point to the object
- \( \text{Focus} \) operation when interesting things happen
  - generate 2 tuples, a \textit{Must} information case and a \textit{MustNot} information case

- Only track access paths to “interesting” objects
- Sound flow functions to lose precision in \textit{MustSet, MustNotSet}
  - Allows \( k \)-limiting. Crucial for scalability.
## Access Path Based Abstractions

<table>
<thead>
<tr>
<th>Statement</th>
<th>Resulting abstract tuples for incoming (\langle o, \text{unique}, \text{typestate}, \text{AP}<em>M, \text{May}, \text{AP}</em>{MN} \rangle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.op()</td>
<td>(\langle o, \text{unique}, \delta \text{(typestate, op)}, \text{AP}<em>M, \text{May}, \text{AP}</em>{MN} \rangle) if (e \notin \text{AP}_{MN}) and ((e \in \text{AP}_M \text{ or May}))</td>
</tr>
<tr>
<td></td>
<td>(\langle o, \text{unique}, \text{typestate}, \text{AP}<em>M, \text{May}, \text{AP}</em>{MN} \rangle) if (e \in \text{AP}_{MN}) or ((e \notin \text{AP}_M \text{ and } \neg \text{(unique } \wedge \text{ pt}(e) = o) \wedge \text{ May}))</td>
</tr>
</tbody>
</table>
## Access Path Based Abstractions

<table>
<thead>
<tr>
<th>Statement</th>
<th>Resulting abstract tuples for incoming $\langle o, \text{unique}, \text{typeofstate}, \text{AP}<em>M, \text{May}, \text{AP}</em>{MN} \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v = \text{new } T()$</td>
<td>$\langle o, \text{false}, \text{typeofstate}, \text{AP}<em>M - {v.\pi \mid \pi \in \Pi}, \text{May}, \text{AP}</em>{MN} \cup {v} \rangle$ $\langle o, \text{false}, \text{init}, {v}, \text{false}, \emptyset \rangle$</td>
</tr>
<tr>
<td>$v = \text{null}$</td>
<td>$\text{AP}'<em>M := \text{AP}<em>M - {v.\pi \mid \pi \in \Pi}$ $\text{AP}'</em>{MN} := \text{AP}</em>{MN} \cup {v}$</td>
</tr>
<tr>
<td>$v.f = \text{null}$</td>
<td>$\text{AP}'<em>M := \text{AP}<em>M - {e'.f.\pi \mid \text{mayAlias}(e',v), \pi \in \Pi}$ $\text{AP}'</em>{MN} := \text{AP}</em>{MN} \cup {v.f}$</td>
</tr>
<tr>
<td>$v = e$</td>
<td>$\text{AP}'<em>M := \text{AP}<em>M \cup {v.\pi \mid e.\pi \in \text{AP}<em>M}$ $\text{AP}'</em>{MN} := \text{AP}</em>{MN} - {v.\pi \mid e.\pi \notin \text{AP}</em>{MN}}$</td>
</tr>
<tr>
<td>$v.f = e$</td>
<td>$\text{AP}'<em>M := \text{AP}<em>M \cup {v.f.\pi \mid e.\pi \in \text{AP}<em>M}$ $\text{AP}'</em>{MN} := \text{AP}</em>{MN} - {v.f \mid e \notin \text{AP}</em>{MN}}$ $\text{May}' := \text{May} \lor (\exists v.f.\pi \in \text{AP}'_M \text{ and } \exists p \in \Pi. \text{mayAlias}(v,p) \land p.f.\pi \notin \text{AP}_M)$</td>
</tr>
</tbody>
</table>

$\Pi = \text{set of all possible access paths in the program}$
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);         <A, init, U, {h.s}, ¬May, {}>
        talk(h.s);          <A, init, ¬U, {h.s}, ¬May, {}> ... 
        dispose(h.s);        <A, connected, ¬U, {h.s}, ¬May, {}>
        open(s);             ✓
        talk(s);
    }
}
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);
  }
}
Access Paths with Focus

\[ AS := \{ <\text{Abstract Object}, \text{TypeState}, \text{Unique}, \text{Must}, \text{May}, \text{MustNot}> \} \]

- 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 \( \langle A, T, u, \text{Must}, \text{May}, \text{MustNot} \rangle \), generate 2 factoids:
    - \( \langle A, \delta(T), u, \text{Must U} \{e\}, \text{May}, \text{MustNot} \rangle \)
    - \( \langle A, T, u, \text{Must}, \text{May}, \text{MustNot U} \{e\} \rangle \)
- 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().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);
  }
}

Access Path Focus

{ <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

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

IBM Intellistation Zpro 2x3GHz Xeon
3.62 GB RAM/ Win XP
IBM J2RE 1.4.2 / -Xmx800M

APMust = 1677
APFocus = 4275
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, ...