Type-State Analysis

CS 8803 FPL
Sep 26, 2012

(Slides Courtesy of Stephen Fink)

Motivation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Defect Removal Cost Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>1</td>
</tr>
<tr>
<td>Design</td>
<td>3</td>
</tr>
<tr>
<td>Code, Unit Test</td>
<td>5</td>
</tr>
<tr>
<td>Function/System Test</td>
<td>12</td>
</tr>
<tr>
<td>User Acceptance Test</td>
<td>32</td>
</tr>
<tr>
<td>Production</td>
<td>95</td>
</tr>
</tbody>
</table>

Typestate

- **Application Trends**
  - Increasing number of libraries and APIs
  - Non-trivial restrictions on permitted sequences of operations

- **Typestate**: Temporal safety properties, encoded as DFAs
  - Apply to many libraries and APIs
    - *e.g.* “Don’t use a Socket unless it is connected”

Goal

- **Typestate Verification**: statically ensure that no execution of a Java program can transition to **err**
  - Sound* (excluding concurrency)
  - Precise enough (reasonable number of false alarms)
  - Scalable
    - Handle programs of realistic size
    - Handle all Java features

* In the real world, some other caveats apply.
Challenge: Aliasing

void foo(Socket s, Socket t) {
    s.connect();
    t.getInputStream(); // potential error?
}

- Strong Updates may be required
  - Rules out solely flow-insensitive analysis

void foo(Socket s, Socket t) {
    s.connect();        // s MUST point to connected
    t = s;             // t MUST point to connected
    t.getInputStream();
}

Our Approach

- Flow-sensitive, context-sensitive abstract interpretation
  - Abstract domains combine typestate and points-to
  - Techniques for inexpensive strong updates
    - Uniqueness
    - Focus
- Staging
  - Family of abstractions of varying cost/precision
  - Early stages reduce work for latter stages

Why this is cool

- Nifty abstractions
  - Combined domain
    - More precise than 2-stage approach
    - Concentrate expensive effort where it matters
  - Parameterized hierarchy of abstractions
    - Relatively inexpensive techniques that allow precise aliasing
      - Much cheaper than shape analysis
      - More precise than usual “scalable” analyses

- It works pretty well
  - Techniques are complementary
    - Flow-sensitive functional IPA with sophisticated alias analysis on
      ~100KLOC in 20 mins.
      - Overapproximate inexpensive facts (distributive)
        - Underapproximate expensive facts (non-distributive)
        - <5% false warnings

Difficulties

- Flow-Sensitivity
- Interprocedural flow
- Context-Sensitivity
- Non-trivial Aliasing
  - Destructive updates
  - Path Sensitivity (ESP)
  - Full Java Language
    - Exceptions, Reflection, ...
- Big programs
Analysis Overview

Possible failure points

Stage 1
Stage 2
Stage 3

Initial Verification Scope

Preliminary Pointer Analysis/Call Graph Construction

Composite Typestate Verifier

Program

Verification Scope

Verifier Stage

Verification Scope

- Sound, abstract representation of program state
- Flow-sensitive propagation of abstract state
- Context-sensitive: *functional approach to interprocedural analysis* [Sharir-Pneuli 82]
  - Tabulation Solver [Reps-Horwitz-Sagiv 95]
- Hierarchy of abstractions

AS := { < Abstract Object, TypeState> }

- Two-Stage Approach
  - First alias analysis, then typestate analysis
- Abstract Object := heap partition from preliminary pointer analysis
  - e.g. allocation site
- Transfer functions
  - Straightforward from instrumented concrete semantics
  - Rely on preliminary pointer analysis to determine typestate transitions
- No Strong Updates
  - \{< l, T> \} \rightarrow \{< l, T>, < l, S(T)> \}
- Works sometimes (75%)

Base Abstraction

It works in some cases:
- Simple abstraction
- Flow-sensitive, context-sensitive solver

```
writeTo(PrintWriter w) { w.write(…); }
main() {
  PrintWriter p = new PrintWriter(…); // P
  PrintWriter q = new PrintWriter(…); // Q
  q.close();
  writeTo(p);
  if (?) {
    p.close();
    writeTo(p);
  }
  else {
    p.close();
    writeTo(p);
  }
}
```
**Unique Abstraction**

**Strong updates**

```
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); <S, init>, <S, connected>  
talk(s); <S, init>, <S, connected>, <S, err> × 
dispose(s); 
}
```

**Base Abstraction**

*"Don't use a Socket unless it is connected"*

```
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); <S, init, U>  
talk(s); <S, connected, U>  
dispose(s): <S, closed, U>  
    }
}
```

---

**Transfer functions:**
- **Unique := true (U)** when creating factoid at allocation site
- **Unique := false (~U)** when propagating factoid through an allocation site

**Intuition:** "Unique" = "∃ exactly one concrete instance of abstract object"

**Strong Updates allowed for e.op() when**
- Unique
- e may point to exactly one abstract object

**Works sometimes (80%)**

---

**More than just singletons?**

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

**Live analysis to the rescue**
- Preliminary live analysis oracle
- On-the-fly remove unreachable configurations
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(); /*A, init, U*/
    Socket s = makeSocket(); /*A, init, ¬U >
    open(h.s); /*A, init, ¬U > /*A, connected, ¬U >
    talk(h.s); /*A, connected, ¬U, h.s, ¬U > /*A, err, ¬U >×…
    dispose(h.s); /*A, err, ¬U >
    open(s); /*A, err, ¬U >
    talk(s);
}

What about aliasing?

class SocketHolder { Socket s; }
Socket makeSocket() { return new Socket(); // A }
init(Socket t) {
    t.connect(); /*A, init, ¬U, h.s, ¬U > /*A, init, ¬U, ¬U, ¬U >
}
talk(Socket u) {
    u.getOutputStream().write("hello"); /*A, connected, ¬U, h.s, ¬U > /*A, connected, ¬U, ¬U, ¬U >

dispose(Socket s) { h.s.close(); } main() {
while(…) {
    SocketHolder h = new SocketHolder();
    h.s = makeSocket(); /*A, init, U, h.s, ¬U > /*A, init, U, h.s, ¬U > /*A, connected, ¬U > /*A, init, ¬U, h.s, ¬U >
    init(h.s); /*A, connected, ¬U, h.s, ¬U > /*A, err, ¬U >×…
    talk(h.s); /*A, connected, ¬U, h.s, ¬U > /*A, err, ¬U >
    dispose(h.s); /*A, err, ¬U >
    init(s); /*A, err, ¬U >
    talk(s);
}

Better aliasing!

Access Path Must Abstraction

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

• Unique Abstraction
  • Must := set of symbolic access paths (e.g.,...) that must point to the object
  • May := false iff all possible access paths appear in Must set

• Flow functions
  • Only track access paths to "interesting" objects
    • Limits computational work dramatically
    • Less precise than shape analysis
  • Always sound to discard Must set and set May := true
  • Allows k-limiting. Crucial for scalability.

• Parameters
  • Width: maximum cardinality of Must Set
  • Depth: maximum length of an individual access path
  • "Interesting" objects: which objects to track precisely
    • currently: typestate objects
  • Typestate transition for e.op() if (e ∈ Must) ∨ (May ∧ mayPointTo(e,I))
  • Strong Updates
    • allowed for e.op() when e ∈ Must or unique logic allows it
  • Works sometimes [91%]

What about destructive updates?

class SocketHolder { Socket s; }
Socket makeSocket() { return new Socket(); // A }
open(Socket l) {
    l.connect(); /*A, init, U, h.s, ¬U > /*A, connected, ¬U >
}
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(); /*A, init, U, h.s, ¬U > /*A, init, U, h.s, ¬U >
    init(h.s); /*A, connected, ¬U, h.s, ¬U > /*A, init, ¬U, h.s, ¬U >
    talk(h.s); /*A, connected, ¬U, h.s, ¬U > /*A, init, ¬U, h.s, ¬U >
    dispose(h.s); /*A, init, ¬U, h.s, ¬U >
    init(s); /*A, init, ¬U, h.s, ¬U >
    talk(s);
}
Access Path Focus Abstraction

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

- **Access Path Must Abstraction**
  - **MustNot**:= set of symbolic access paths that must not point to the object

- **Flow functions**
  - **Focus** operation when "interesting" things happen
    - "materialization", "focus", "case splitting"
    - e.op() on \(<A, T, u, \text{Must, May, MustNot}>\), generate 2 factoids:
      - \(<A, \delta(T), u, \text{Must} \cup \{e\}, \text{May}, \text{MustNot}>\)
      - \(<A, T, u, \text{Must, May, MustNot} \cup \{e\}>\)

- **Interesting Operations**
  - Typestate changes
  - Observable polymorphic dispatch
  - Allows k-limiting, Crucial for scalability
  - Allowed to limit exponential blowup due to focus
  - Current heuristic: discard MustNot before each focus operation

- Works sometimes (95.6%)

Access Path Focus Abstraction

Recover from destructive updates

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

Intraprocedural Verifier

- **Single-procedure version of Access Path Focus abstraction**
- **Worst-case assumptions at method entry, calls**
  - Mitigated by live analysis
- **Works sometimes (66%)**
Flow-Insensitive Pruning

- From alias oracle, build typestate DFA for each abstract object
- Prune verification scope by DFA reachability
- It works sometimes (30%)

Sparsification

- Separation (solve for each abstract object separately)
- “Slicing”: discard branches of supergraph that cannot affect abstract semantics
  - Identify program variables that might appear k-limited access path
    - K-step reachability from typestate objects from prelim. pointer analysis
  - Identify call graph nodes that might
    - modify these variables
    - cause typestate transitions (depends on incoming verification scope)
  - Discard any nodes that cannot (transitively) affect abstract interpretation
  - Reduces median supergraph size by 50X

Preliminary Pointer Analysis/Call Graph Construction

- Typestate verifiers rely on call graph, fallback alias oracle
- Current implementation: flow-insensitive, partially context-sensitive pointer-analysis
  - 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
    - Arraycopy, clone, ...
  - Heuristics for reflection (e.g. Livshits et al 2005)
- Details matter a lot
  - e.g. context-insensitive preliminary; later stages time out, terrible precision
Typestate Properties for J2SE libraries

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumeration</td>
<td>Call hasNextElement before nextElement</td>
</tr>
<tr>
<td>InputStream</td>
<td>Do not read from a closed InputStream</td>
</tr>
<tr>
<td>Iterator</td>
<td>Do not call next without first checking hasNext</td>
</tr>
<tr>
<td>KeyStore</td>
<td>Always initialize a KeyStore before using it</td>
</tr>
<tr>
<td>PrintStream</td>
<td>Do not use a closed PrintStream</td>
</tr>
<tr>
<td>PrintWriter</td>
<td>Do not use a closed PrintWriter</td>
</tr>
<tr>
<td>Signature</td>
<td>Follow initialization phases for Signature</td>
</tr>
<tr>
<td>Socket</td>
<td>Do not use a Socket until it is connected</td>
</tr>
<tr>
<td>Stack</td>
<td>Do not peek or pop an empty Stack</td>
</tr>
<tr>
<td>URLConn</td>
<td>Illegal operation performed when already connected</td>
</tr>
<tr>
<td>Vector</td>
<td>Do not access elements of an empty Vector</td>
</tr>
</tbody>
</table>

Benchmark Classes

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Classes</th>
<th>Methods</th>
<th>Bytecode Statements</th>
<th>Contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcel</td>
<td>1,723</td>
<td>7,130</td>
<td>1,474,264</td>
<td>13,725</td>
</tr>
<tr>
<td>gj</td>
<td>230</td>
<td>2,362</td>
<td>139,305</td>
<td>2,521</td>
</tr>
<tr>
<td>java_cup</td>
<td>123</td>
<td>661</td>
<td>53,296</td>
<td>990</td>
</tr>
<tr>
<td>jbidwatcher</td>
<td>1,182</td>
<td>4,994</td>
<td>1,029,507</td>
<td>7,030</td>
</tr>
<tr>
<td>jhotdraw</td>
<td>1,688</td>
<td>6,337</td>
<td>1,400,640</td>
<td>11,203</td>
</tr>
<tr>
<td>jlex</td>
<td>111</td>
<td>473</td>
<td>44,736</td>
<td>776</td>
</tr>
<tr>
<td>jpat-p</td>
<td>64</td>
<td>225</td>
<td>17,783</td>
<td>269</td>
</tr>
<tr>
<td>kawa-c</td>
<td>612</td>
<td>3,027</td>
<td>141,527</td>
<td>3,438</td>
</tr>
<tr>
<td>ij2j</td>
<td>838</td>
<td>4,247</td>
<td>877,077</td>
<td>6,438</td>
</tr>
<tr>
<td>lucene</td>
<td>1,783</td>
<td>6,694</td>
<td>1,474,334</td>
<td>12,576</td>
</tr>
<tr>
<td>portelec</td>
<td>1,800</td>
<td>6,737</td>
<td>1,481,249</td>
<td>13,430</td>
</tr>
<tr>
<td>rhino-a</td>
<td>196</td>
<td>1,293</td>
<td>92,225</td>
<td>1,645</td>
</tr>
<tr>
<td>sablecc-j</td>
<td>391</td>
<td>2,144</td>
<td>96,201</td>
<td>2,747</td>
</tr>
<tr>
<td>schroeder-m</td>
<td>1,459</td>
<td>5,215</td>
<td>1,387,432</td>
<td>9,682</td>
</tr>
<tr>
<td>soot-c</td>
<td>665</td>
<td>2,764</td>
<td>144,554</td>
<td>3,272</td>
</tr>
<tr>
<td>specjvm98</td>
<td>965</td>
<td>4,673</td>
<td>979,159</td>
<td>8,152</td>
</tr>
<tr>
<td>symjpack-t</td>
<td>74</td>
<td>305</td>
<td>80,508</td>
<td>351</td>
</tr>
<tr>
<td>toba-s</td>
<td>163</td>
<td>760</td>
<td>65,415</td>
<td>1,169</td>
</tr>
<tr>
<td>tvia</td>
<td>346</td>
<td>2,077</td>
<td>139,474</td>
<td>12,874</td>
</tr>
</tbody>
</table>

Warnings/PPFs (%)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>PPFs</th>
<th>FI</th>
<th>LocalFocus</th>
<th>Base</th>
<th>Unique</th>
<th>APMust</th>
<th>APFocus</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcel</td>
<td>234</td>
<td>46</td>
<td>388</td>
<td>158</td>
<td>101</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>gj</td>
<td>502</td>
<td>150</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>java_cup</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>jbidwatcher</td>
<td>855</td>
<td>2</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>jhotdraw</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>jlex</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>jpat-p</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>kawa-c</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>92</td>
<td>92</td>
</tr>
<tr>
<td>ij2j</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>lucene</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>portelec</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>rhino-a</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>sablecc-j</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>schroeder-m</td>
<td>98</td>
<td>98</td>
<td>98</td>
<td>98</td>
<td>98</td>
<td>98</td>
<td>98</td>
</tr>
<tr>
<td>soot-c</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>specjvm98</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>symjpack-t</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>toba-s</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>tvia</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
<td>101</td>
</tr>
</tbody>
</table>
Running time

![Running time graph](image)

Limitations

- **Limitations of analysis (~50%)**
  - Aliasing
  - Path sensitivity
  - Return values
    
    ```java
    if (!stack.isEmpty()) stack.pop();
    vector.get(vector.size()-1);
    ```
    Not always straightforward (encapsulation)
    ```java
    if (!foo.isAnEmptyFoo()) foo.pop();
    ```

- **Limitations of typestate abstraction (~50%)**
  - Application logic bypasses DFA, still OK
    ```java
    if (isALeBlueMoon) stack.pop();
    vector.get(numberOfPixels/2);
    try {
        emptyStack.pop();
        vector.get(numberOfPixels/2);
    } catch (EmptyStackException e) {
        System.out.println("I expected that.");
    }
    ```

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: 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, Hackel-Rugina 05, TVLA (Sagiv-Reps-Wilhelm), …