Microsoft Research

Each year Microsoft Research hosts hundreds of influential speakers from around the world including leading scientists, renowned experts in technology, book authors, and leading academics, and makes videos of these lectures freely available.

2016 © Microsoft Corporation. All rights reserved.
Welcome!
## The Morning

<table>
<thead>
<tr>
<th>Time</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>8:30am</td>
<td>Light Breakfast</td>
</tr>
<tr>
<td>8:45am</td>
<td>Welcome and Introductions</td>
</tr>
<tr>
<td>9:00am</td>
<td>Concerto: Towards a Framework for Combined Concrete and Abstract Interpretation</td>
</tr>
<tr>
<td></td>
<td>John Toman and Dan Grossman</td>
</tr>
<tr>
<td>9:15am</td>
<td>The Time for Proof Reuse is Now!</td>
</tr>
<tr>
<td></td>
<td>Talia Ringer, Nathaniel Yazdani, John Leo, Dan Grossman</td>
</tr>
<tr>
<td>9:45am</td>
<td>Puddle: An OS for Reliable High-Level Programming of Digital Microfluidic Devices</td>
</tr>
<tr>
<td></td>
<td>Max Willese, Luis Ceze, Karin Strauss</td>
</tr>
<tr>
<td>10:00am</td>
<td>Inferring Likely Distributed System State Invariant</td>
</tr>
<tr>
<td></td>
<td>Stewart Grant, Ivan Beschastnik</td>
</tr>
<tr>
<td>10:15am</td>
<td>Break</td>
</tr>
<tr>
<td>10:30am</td>
<td>Helping Designers Explore the Space of Layout Variations with Constraints</td>
</tr>
<tr>
<td></td>
<td>Amanda Swearngin, Andrew J. Ko, James Fogarty</td>
</tr>
<tr>
<td>10:45am</td>
<td>Platform-Independent Migration of Stateful JavaScript IoT Applications</td>
</tr>
<tr>
<td></td>
<td>Julien Gascon-Samson, Kumsook Jung, Karthik Pattabiraman</td>
</tr>
<tr>
<td>11:00am</td>
<td>Compiling Distributed System Specifications into Implementations</td>
</tr>
<tr>
<td></td>
<td>Matthew Do, Renato Mascarenhas, Brandon Zhang, Finn Hackett, Stewart Grant, Ivan Beschastnik</td>
</tr>
<tr>
<td>11:15am</td>
<td>What bugs and tests should we use in experiments?</td>
</tr>
<tr>
<td></td>
<td>René Just</td>
</tr>
<tr>
<td>11:45am</td>
<td>Verifying Web Pages</td>
</tr>
<tr>
<td></td>
<td>Pavel Panchevka, Adam Galler, Michael D. Ernet, Sheoh Kamil, Zachary Tallock</td>
</tr>
</tbody>
</table>
### The Afternoon (Part 1)

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Speaker(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:00pm</td>
<td>Lunch</td>
<td></td>
</tr>
<tr>
<td>12:15pm</td>
<td>Lunch Talk: Project Everest: Theory meets reality</td>
<td>Jonathan Protzenko</td>
</tr>
<tr>
<td>12:30pm</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>12:45pm</td>
<td>Lunch Talk: Contently Integrated Verified Cryptography</td>
<td>Mike Dodds</td>
</tr>
<tr>
<td>1:00pm</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>1:15pm</td>
<td>Featured Talks: Continuously Integrated Verified Cryptography</td>
<td>Mike Dodds</td>
</tr>
<tr>
<td>1:30pm</td>
<td>Helena: A web automation language for end users</td>
<td>Sarah Chasins, Ras Bodik</td>
</tr>
<tr>
<td>1:45pm</td>
<td>Sinking Point</td>
<td>Bill Zorn, Dan Grossman</td>
</tr>
<tr>
<td>2:00pm</td>
<td>Verified Extraction with Native Types</td>
<td>Stuart Pernsteiner, Eric Mullen, James R. Wilcox, Zachary Tatlock, Dan Grossman</td>
</tr>
<tr>
<td>Time</td>
<td>Event</td>
<td>Speaker(s)</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>2:15pm</td>
<td>Lightning Talk Session</td>
<td></td>
</tr>
<tr>
<td>2:30pm</td>
<td>Poster Session</td>
<td></td>
</tr>
<tr>
<td>2:45pm</td>
<td>Break</td>
<td></td>
</tr>
<tr>
<td>3:00pm</td>
<td>Featured Talk: Why not both?</td>
<td>Eric Walkingshaw</td>
</tr>
<tr>
<td></td>
<td>Applications of variational</td>
<td></td>
</tr>
<tr>
<td></td>
<td>programming</td>
<td></td>
</tr>
<tr>
<td>3:45pm</td>
<td>Chapel Comes of Age: Productive</td>
<td>Brad Chamberlain</td>
</tr>
<tr>
<td></td>
<td>Parallelism at Scale</td>
<td></td>
</tr>
<tr>
<td>4:00pm</td>
<td>Musical Ornaments</td>
<td>John Leo</td>
</tr>
<tr>
<td>4:15pm</td>
<td>Incrementalization with Data</td>
<td>Calvin Loncaric, Michael D. Ernst</td>
</tr>
<tr>
<td></td>
<td>Structures</td>
<td></td>
</tr>
<tr>
<td>4:45pm</td>
<td>Wrap up and Close</td>
<td></td>
</tr>
<tr>
<td>5:00pm</td>
<td>Group Dinner (self pay)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Title</td>
<td>Authors</td>
</tr>
<tr>
<td>---</td>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>Cosette: An Automated Prover for SQL</td>
<td>Shumo Chu, Alvin Cheung, Dan Suciu</td>
</tr>
<tr>
<td>2</td>
<td>Dependency Capture for Reproducible Builds</td>
<td>Martin Kellogg</td>
</tr>
<tr>
<td>3</td>
<td>Sloth: Locating Sites for Repetitive Edits with Lazy Concrete Pattern Matching on Trees</td>
<td>Remy Wang, Rashmi Mudduluru, Hadar Greinsmark</td>
</tr>
<tr>
<td>4</td>
<td>Time-Travel Diagnostics for Node.js/JavaScript</td>
<td>Mark Marron</td>
</tr>
<tr>
<td>5</td>
<td>Synchronizing the asynchronous</td>
<td>Thomas Henzinger, Bernhard Kragl, Shaz Qadeer</td>
</tr>
<tr>
<td>6</td>
<td>Experimental Design as Programs</td>
<td>Eunice Jun, Jared Roesch, Sarah Chasins</td>
</tr>
<tr>
<td>7</td>
<td>Designing Compilers and Synthesis Tools for 3D Printing</td>
<td>Chandrakana Nandi</td>
</tr>
<tr>
<td>8</td>
<td>Adaptive Program Ranking</td>
<td>Chenglong Wang</td>
</tr>
<tr>
<td>9</td>
<td>A Formal Model of Polymorphism and Inference in Rust</td>
<td>Joseph Eremendi, Ron Garcia</td>
</tr>
<tr>
<td>10</td>
<td>Relay: an IR for differentiable programming</td>
<td>Jared Roesch, Tianqi Chen, Steven Lyubomirsky, Zachary Tellock, Josh Pollock, Logan Weber</td>
</tr>
<tr>
<td>11</td>
<td>Automated Verification of Cryptographic Protocols</td>
<td>James Bornholt, Ernie Cohen, K. Rustan M. Leino</td>
</tr>
<tr>
<td>12</td>
<td>Interactively Debugging Distributed Systems</td>
<td>Doug Woos</td>
</tr>
<tr>
<td>13</td>
<td>Helping Designers Explore the Space of Layout Variations with Constraints</td>
<td>Amanda Swearngin, Andrew J. Ko, James Fogarty</td>
</tr>
</tbody>
</table>
Thanks!!

Organizing Committee

- Ben Zorn, MSR
- Tom Ball, MSR
- Zach Tatlock, UW

Program Committee

- Preston Briggs, Reservoir Labs
- Brad Chamberlain, Cray
- Vinod Grover, nVidia
- Leo de Moura, MSR
- Gail Murphy, UBC
- Todd Mytkowicz, MSR
- Wolfram Schulte, Facebook
- Aaron Tomb, Galois, Inc.
- Eric Walkingshaw, OSU
- Michal Young, UQ
Thanks!!

Organizing Committee

- Ben Zorn, MSR
- Tom Ball, MSR
- Zach Tatlock, UW

Program Committee

- Preston Briggs, Reservoir Labs
- Brad Chamberlain, Cray
- Vinod Grover, nVidia
- Leo de Moura, MSR
- Gail Murphy, UBC
- Todd Mytkowicz, MSR
- Wolfram Schulte, Facebook
- Aaron Tomb, Galois, Inc.
- Eric Walkingshaw, OSU
- Michal Young, UQ

Amanda Robles
Concerto: A Framework for Combined Concrete and Abstract Interpretation

John Toman & Dan Grossman
University of Washington
Where’s John?

List of acceptable reasons for your advisor to give your talk for you:

1. You had your first child < 3 days ago

About John:

• Graduating next year
• Work presented here will be the core of his dissertation
proc(a[100]) {
  read i,j;
  k = i;
  a[1] = 0;
  for l=1 to i do
    for m=i to j do
      k = k + m;
    done
  a[1] = k;
  if k<1000 then
    write k;
  else
    k = i;
  endif
  a[1+1] = a[1] / 2;
  done
  write a[i];
}
Abstract Interpretation: The Real World

- Pervasive use of reflection/metaprogramming
- Many, many layers of abstraction
- Enormous libraries
- Program behavior determined by non-code artifacts (config files, annotations, etc.)
Dealing with the Real World

1. Soundness
   - Relies on unrealistic assumptions about the use of metaprogramming

2. Pessimistic ("Sound") assumptions
   - Hopelessly imprecise in practice

3. Manual annotations or models
   - Requires an unsustainably large effort per framework/library
Dealing with the Real World

1. Soundness
   - Relies on unrealistic assumptions about the use of metaprogramming

2. Pessimistic (“Soundness”)
   - Possibly imprecise in practice
   - Maybe try a totally different technique?

3. Manual annotations or models
   - Requires an unsustainably large effort per framework/library
Picking the Right Tool
Picking the Right Tool

**Frameworks**
- Extreme flexibility, driven by configuration
- Use multiple layers of abstraction, reflection
- Minimal branching
Picking the Right Tool

Frameworks

- Extreme flexibility, driven by configuration
- Use multiple layers of abstraction, reflection
- Minimal branching

Exhaustive Path Exploration
Picking the Right Tool

**Frameworks**
- Extreme flexibility, driven by configuration
- Use multiple layers of abstraction, reflection
- Minimal branching

**Applications**
- Focused on a fixed set of tasks
- Less indirection, more “straightforward” code
- Complex branching, unbounded loops

Exhaustive Path Exploration
Picking the Right Tool

**Frameworks**
- Extreme flexibility, driven by configuration
- Use multiple layers of abstraction, reflection
- Minimal branching

**Applications**
- Focused on a fixed set of tasks
- Less indirection, more "straightforward" code
- Complex branching, unbounded loops

---

Exhaustive Path Exploration  Abstract Interpretation
Picking the Right Tool

Frameworks
- Extreme flexibility, driven by configuration
- Use multiple layers of abstraction, reflection
- Minimal branching

Applications
- Focused on a fixed set of tasks
- Less indirection, more “straightforward” code
- Complex branching, unbounded loops

Exhaustive Path Exploration
Abstract Interpretation
# Picking the Right Tool

<table>
<thead>
<tr>
<th>Frameworks</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌ Extreme flexibility, driven by configuration</td>
<td>❌ Focused on a fixed set of tasks</td>
</tr>
<tr>
<td>❌ Use multiple layers of abstraction, reflection</td>
<td>❌ Less indirection, more “straightforward” code</td>
</tr>
<tr>
<td>❌ Minimal branching</td>
<td>❌ Complex branching, unbounded loops</td>
</tr>
</tbody>
</table>

- Exhaustive Path Exploration
- Abstract Interpretation
Picking the Right Tool

Frameworks
- Extreme flexibility, driven by configuration
- Use multiple, more abstracted layers of code
- Minimal build time

Applications
- Focused on a fixed set of applications
- “Code as you go” and “build and ship” code

A single, unified analysis strategy will not work

Exhaustive Path Exploration

Abstract Interpretation
Picking the Right Tool

Frameworks
- Extreme flexibility, driven by configuration
- Use multiple programming languages
- Minimal boilerplate code

Applications
- Focused on a fixed set of abstraction
- More structured, less boilerplate code
- Minimal boilerplate code

Why not both?

Exhaustive Path Exploration
Abstract Interpretation
Concerto

A hybrid analysis framework that enables the precise analysis of framework-based applications without any manual modeling.
Concerto
Concerto by Example

Framework Code

```java
main() {
    m = init("config");
    app(m);
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```c
main() {
    m = init("config");
    app(m);
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}

init(f) {
    conf = open(f);
    m = {};
    while(!conf.eof()) {
        m[conf.read()] = conf.read();
    }
    return m;
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}

init(f) {
    conf = open(f);
    m = {};
    while(!conf.eof()) {
        m[conf.read()] = conf.read();
    }
    return m;
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```c
main() {
  m = init("config");
  app(m);
}

init(f) {
  conf = open(f);
  m = {};
  while(!conf.eof()) {
    m[conf.read()] = conf.read();
  }
  return m;
}
```

```
+---+
| config |
|      |
| foo   |
| a     |
| bar   |
| b     |
+---+
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}

init(f) {
    conf = open(f);
    m = {};
    while(!conf.eof()) {
        m[conf.read()] = conf.read();
    }
    return m;
}
```

Available at analysis time

```
config
  foo
  a
  bar
  b
```
Concerto by Example

(Mostly-)Concrete Interpreter

```c
main() {
    m = init("config");
    app(m);
}

init(f) {
    conf = open(f);
    m = {};
    while(!conf.eof()) {
        m[conf.read()] = conf.read();
    }
    return m;
}
```

```
  config
    foo
    a
    bar
    b
```
Concerto by Example

(Mostly-)Concrete Interpreter

```c
main() {
  m = init("config");
  app(m);
}
```

```c
init(f) {
  conf = open(f);
  m = {};
  while(!conf.eof()) {
    m[conf.read()] = conf.read();
  }
  return m;
}
```

```
[m ↦ {"foo" ↦ "a", "bar" ↦ "b"}]
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}
```

Application Code

```java
app(p) {
    while(...) {
        x = *
        if(x > 0) {
            dispatch(p, "foo", x);
        } else {
            dispatch(p, "bar", x);
        }
    }
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}
```

Abstract Interpreter

```java
[p \mapsto \{"foo" \mapsto "a", "bar" \mapsto "b"\}]
```

We have reused the concrete value in the abstract interpreter!

How?
State Separation Assumption

The framework and application “own” disjoint sets of types, and framework types are opaque to the application and vice versa.
State Separation Assumption

The framework and application “own” disjoint sets of types, and framework types are opaque to the application and vice versa.

Framework types are manipulated only in the framework, and similarly for application types.
State Separation Assumption

The framework and application “own” disjoint sets of types, and framework types are opaque to the application and vice versa.
State Separation Assumption

The framework and application “own” disjoint sets of types, and framework types are opaque to the application and vice versa.

```plaintext
Framework Code

init(f) {
    conf = open(f);
    m = {};
    while(!conf.eof()) {
        m[conf.read()] = conf.read();
    }
    return m;
}
```
State Separation Assumption

The framework and application “own” disjoint sets of types, and framework types are opaque to the application and vice versa.

Framework Code

```java
init(f) {
    conf = open(f);
    m = {};
    while (!conf.eof()) {
        key = conf.key();
        val = conf.value();
        m[key] = val;
    }
}
```

type map = (str * str) list
put: map→(str * str)→map
get: map→str → str
...

State Separation Assumption

The framework and application “own” disjoint sets of types, and framework types are opaque to the application and vice versa.

```plaintext
Framework Code

init(f) {
    conf = open(f);
    m = {};
    while (!conf.eof()) {
        read() to conf.read();
    }
}

type map = (str * str) list
put: map→(str * str)→map
get: map→str → str
...

Application Code

app(p) {
    while(...) {
        x = *
        if(x > 0) {
            dispatch(p, "foo", x);
        } else {
            dispatch(p, "bar", x);
        }
    }
}
```
State Separation Assumption

The framework and application “own” disjoint sets of types, and framework types are opaque to the application and vice versa.

Framework Code

```plaintext
init(f) {
    conf = open(f);
    m = {};
    while (!conf.eof()) {
        key = conf.read();
        val = conf.read();
        m[key] = val;
    }
}
```

Application Code

```plaintext
app(p) {
    while(...) {
        x = *;
        if(x > 0) {
            type map = (str * str) list
            put: map→(str * str)→map
            get: map→str → str
            ...
            type map
        }
    }
```
State Separation Assumption

The framework and application “own” disjoint sets of types, and framework types are opaque to the application and vice versa.

```plaintext

Framework Code

init(f) {
    conf = open(f);
    m = {};
    while (!conf.eof()) {
        read() = conf.read();
    }

type map = (str * str) list
put: map→(str * str)→map
get: map→str → str
...

Application Code

app(p) {
    while(...) {
        x = *;
        if(x == 0) {

            type int = ...|−1|0|1|...
add: int→int→int
greater: int→int→bool
...
```
State Separation Assumption

The framework and application “own” disjoint sets of types, and framework types are opaque to the application and vice versa.

```
Framework Code

init(f) {
    conf = open(f);
    m = {};
    while (!conf.eof()) {
        m[conf.read()] = conf.read();
    }
}

type int

Application Code

app(p) {
    while(...) {
        x = *;
        if(x > 0) {
            type int = ...| -1| 0 | 1 | ...
            add: int→int → int
            greater: int→int→bool
            ...
        }
    }
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}
```

Abstract Interpreter

```java
app(p) {
    [p -> {"foo" -> "a", "bar" -> "b"}]
    dispatch(p, "foo", x);
    else {
        dispatch(p, "bar", x);
    }
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```cpp
main() {
    m = init("config");
    app(m);
}
```

Abstract Interpreter

```cpp
app(p) {
    while(...) {
        x = *;
        if(x > 0) {
            dispatch(p, "foo", x);
        } else {
            dispatch(p, "bar", x);
        }
    }
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
  m = init("config");
  app(m);
}
```

Abstract Interpreter

```java
app(p) {
  while(...) {
    x = *
    if(x > 0) {
      dispatch(p, "foo", x);
    }
  }
}
```

```json
[p -> {
  "foo" -> "a",
  "bar" -> "b"
}, x -> {+}]
```
Concerto by Example

(Mostly-)Concrete Interpreter

```
main() {
  m = init("config");
  app(m);
}
```

Abstract Interpreter

```
app(p) {
  while(...) {
    x = *;
    if(x > 0) {
      dispatch(p, "foo", x);
    }
  }
}
```

```
[p ↔ {
  "foo" ↔ "a",
  "bar" ↔ "b"
}, x ↔ {+}]
```

Simple signedness domain
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m[key], arg);
}
```

Abstract Interpreter

```java
app(p) {
    while(...) {
        x = *
        if(x > 0) {
            dispatch(p, "foo", x);
        }
    }
}

[p → {"foo" → "a", "bar" → "b"}, x → {+}]
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m[key], arg);
}
```

(Abstract) Interpreter

```java
app(p) {
    while(...) {
        x = *;
        if(x > 0) {
            dispatch(p, "foo", x);
        }
    }
}
```

```
[p \mapsto \{"foo" \mapsto "a", "bar" \mapsto "b"\}, x \mapsto \{+\}]
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke([key], arg);
}
```

Abstract Interpreter

```java
app(p) {
    while(...) {
        x = *;
        if(x > 0) {
            dispatch(p, "foo", x);
        }
    }
}
```

[p ↦ {"foo" ↦ "a", "bar" ↦ "b"}, x ↦ {+}]
Concerto by Example

(Mostly-)Concrete Interpreter

```c
main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m, key, arg);
} {
    "foo" => "a", "bar" => "b"},
```

Abstract Interpreter

```c
app(p) {
    while(...) {
        x = *;
        if(x > 0) {
            dispatch(p, "foo", x);
        }
    }
    [p => {"foo" => "a", "bar" => "b"}, x => {+}]}
```

Extract
Concerto by Example

(Mostly-)Concrete Interpreter

```c
main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m[key], arg);
}
```

Abstract Interpreter

```c
app(p) {
    while(...) {
        x = *
        if(x > 0) {
            dispatch(p, "foo", x);
        }
    }
}
```

[p → {"foo" → "a", "bar" → "b"}, x → {+}]
Concerto by Example

(Mostly-)Concrete Interpreter

```javascript
main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m[key], arg);
}
```

Abstract Interpreter

```javascript
app(p) {
    while(...) {
        x = *;
        if(x > 0) {
            dispatch(p, "foo", x);
        }
    }
}
```

[p ↦ {"foo" ↦ "a", "bar" ↦ "b"}, x ↦ {+}]

?
Concerto by Example

(Mostly-)Concrete Interpreter

main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m[key], arg);
}

Abstract Interpreter

app(p) {
    while(...) {
        x = *
        if(x > 0) {
            dispatch(p, "foo", x);
        }
    }
}

[p ↦ {"foo" ↦ "a", "bar" ↦ "b"}, x ↦ {+}]
Concerto by Example

(Mostly-)Concrete Interpreter

```
main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m[key], arg);
}
```

Abstract Interpreter

```
app(p) {
    while(...) {
        x = *
        if(x > 0) {
            dispatch(p, "foo", x);
        }
    }
}
```

Reuse the abstract value

\[ p \mapsto \{ "foo" \mapsto "a", "bar" \mapsto "b" \}, x \mapsto \{ + \} \]
Concerto by Example

(Mostly-)Concrete Interpreter

```c
main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m[key], arg);
}
```

Abstract Interpreter

```c
app(p) {
    while(...) {
        x = *
        if(x > 0) {
            dispatch(p, "foo", x);
        } else {
            dispatch(p, "bar", x);
        }
    }
}
```

Reflectively invokes the named procedure
Concerto by Example

(Mostly-)Concrete Interpreter

```plaintext
main() {
    m = init("config");
    app(m);
}
```

```
[m ↦ {"foo" ↦ "a","bar" ↦ "b"}, arg↦{+},key↦"foo"]
```

```
dispatch(m, key, arg) {
    invoke(m[key], arg);
}
```

Reflectively invokes the named procedure

Abstract Interpreter

```
app(p) {
    while(...) {
        x = *
        if(x > 0) {
            dispatch(p, "foo", x);
        } else {
            dispatch(p, "bar", x);
        }
    }
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
    
    dispatch(m, key, arg) {
        invoke(m[key], arg);
    }
}
```

Abstract Interpreter

```java
app(p) {
    while(...) {
        x = *;
        if(x > 0) {
            dispatch(p, "foo", x);
        } else {
            dispatch(p, "bar", x);
        }
    }
}
```

[\(m \mapsto \{"foo" \mapsto "a", "bar" \mapsto "b"\}, arg \mapsto \{+\}, key \mapsto "foo"\}]

Reflectively invokes the named procedure
Concerto by Example

(Mostly-)Concrete Interpreter

main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m[key], arg);
}

[\{m \mapsto \{"foo" \mapsto "a","bar" \mapsto "b"\}, \text{arg} \mapsto \{+, \text{key} \mapsto "foo"\}\}]

Abstract Interpreter

app(p) {
    ...
}

call

a(k) {
    if(k < 0) {
        fail();
    } else {
        ... }
}

Reflectively invokes the named procedure
Concerto by Example

(Mostly-)Concrete Interpreter

```
main() {
    m = init("config");
    app(m);
    dispatch(m, key, arg) {
        invoke(m[key], arg);
    }
}
```

\[ m \mapsto \{ \text{"foo" \mapsto "a", "bar" \mapsto "b"}, \text{arg} \mapsto \{+}, \text{key} \mapsto \text{"foo"} \}\]

yield

Abstract Interpreter

```
app(p) {
    ...
}
```

call

```
a(k) {
    if(k < 0) {
        fail();
    } else {
        ...
    }
}
```

Reflectively invokes the named procedure
Concerto by Example

(Mostly-)Concrete Interpreter

main() {
    m = init("config");
    app(m);
}

dispatch(m, key, arg) {
    invoke(m[key], arg);
}

[\langle m \mapsto \{\text{foo} \mapsto \text{a}, \text{bar} \mapsto \text{b}, \text{arg} \mapsto +, \text{key} \mapsto \text{foo}\}\rangle]

Abstract Interpreter

call

app(p) {
    ...?
    a(k) {
        if(k < 0) {
            fail();
        } else {
            ... }
    }
}

Reflectively invokes the named procedure
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
}
```

Abstract Interpreter

```java
app(p) {
    ...
    ...
    a(k) {
        if(k < 0) {
            fail();
        } else {
            ... 
        }
    }
}
```

[\(m \mapsto \{\text{foo} \mapsto \text{a}, \text{bar} \mapsto \text{b}\}, \text{arg} \mapsto \{+\}, \text{key} \mapsto \text{foo}\}]

dispatch(m, key, arg) {
    invoke(m[key], arg);
}

Reflectively invokes the named procedure
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
  m = init("config");
  app(m);
}

dispatch(m, key, arg) {
  invoke(m[key], arg);
}
```

Abstract Interpreter

```java
app(p) {
  ...
  {+
    ...
  }
  a(k) {
    if (k < 0) {
      fail();
    } else {
      ...
    }
  }
}
```

Reflectively invokes the named procedure
Concerto by Example

(Mostly-)Concrete Interpreter

```cpp
main() {
    m = init("config");
    app(m);
}
```

dispatch(m, key, arg) {
    invoke(m[key], arg);
}

```
[m → {"foo" → "a", "bar" → "b"}, arg→{+}, key→"foo"]
```

yield

Abstract Interpreter

```cpp
app(p) {
    ...
    {+}
    ...
}
```

```cpp
a(k) {
    if(k < 0) {
        fail();
    } else {
        ..
    }
}
```

Reflectively invokes the named procedure

Provably unreachable
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
    dispatch(m, key, arg) { 
        invoke(m[key], arg);
    }
}
```

```
[m ↦ {"foo" ↦ "a", "bar" ↦ "b"}, arg ↦ {+}, key ↦ "foo"]
```

Abstract Interpreter

```java
a(k) {
    if(k < 0) {
        fail();
    } else {
        ...
    }
}
```

```java
b(k) {
    if(k < 0) {
        ...
    } else {
        fail();
    }
}
```
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);

dispatch(m, key, arg) {
    invoke(m[key], arg);
}
```

[\(m \mapsto \{\)foo\mapsto"a",bar\mapsto"b",arg\mapsto\{+,key\mapsto"foo"\}\}]

Abstract Interpreter

```java
a(k) {
    if(k < 0) {
        fail();
    } else { ... }
}

b(k) {
    if(k < 0) {
        ...
    } else { 
        fail();
    }
}
```

call?

Precise, concrete semantics and representation
Concerto by Example

(Mostly-)Concrete Interpreter

```java
main() {
    m = init("config");
    app(m);
    
    [m ⇔ \{"foo" ⇔ "a", "bar" ⇔ "b", arg⇒{+}, key⇒"foo"\}]
    
    dispatch(m, key, arg) {
        invoke(m[key], arg);
    }
}
```

Abstract Interpreter

```java
a(k) {
    if(k < 0) {
        fail();
    } else {
        ... 
    }
}

b(k) {
    if(k < 0) {
        ...
    } else {
        fail();
    }
}
```

Precise, concrete semantics and representation
Concerto: In Summary

• Interleaved mostly-concrete and abstract interpretation

• *State Separation:* Assumption that application types are opaque to the framework and vice versa

• When yielding to the AI, concrete values are embedded into the abstract interpreter

• Symmetrically, abstract values are embedded into the mostly-concrete interpreter
Soundness of Combined Interpretation

- Formalized and proved sound semantics of mostly-concrete interpretation
- Proved the soundness for general framework for combining arbitrary interpreters
- Combined abstract and mostly-concrete interpretation is a special case of the above, from which soundness follows immediately
## Proof of Concept Implementation

<table>
<thead>
<tr>
<th><strong>Target Language</strong></th>
<th><strong>Java</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces</td>
<td>Exceptions</td>
</tr>
<tr>
<td>(Dynamically Sized) Arrays</td>
<td>Strings</td>
</tr>
<tr>
<td>Integers</td>
<td>Concurrency</td>
</tr>
<tr>
<td>Downcasts</td>
<td>Static Fields</td>
</tr>
<tr>
<td>Reflection API</td>
<td>Static Initializers</td>
</tr>
<tr>
<td>Loops</td>
<td>Inheritance</td>
</tr>
<tr>
<td>I/O</td>
<td>float/short/...</td>
</tr>
</tbody>
</table>
# Proof of Concept Implementation

<table>
<thead>
<tr>
<th>Target Language</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces</td>
<td>Exceptions</td>
</tr>
<tr>
<td>(Dynamically Sized) Arrays</td>
<td>Strings</td>
</tr>
<tr>
<td>Integers</td>
<td>Concurrency</td>
</tr>
<tr>
<td>Downcasts</td>
<td>Static Fields</td>
</tr>
<tr>
<td>Reflection API</td>
<td>Static Initializers</td>
</tr>
<tr>
<td>Loops</td>
<td>Inheritance</td>
</tr>
<tr>
<td>I/O</td>
<td>float/short/...</td>
</tr>
</tbody>
</table>
# Proof of Concept Implementation

## Target Language
- Interfaces
- (Dynamically Sized) Arrays
- Integers
- Downcasts
- Reflection API
- Loops
- I/O

## Java
- Exceptions
- Sufficient for capturing difficult to analyze framework behavior!
- Inheritance
- float/short/…
YAWN: Your Analysis’ Worst Nightmare

A simple framework that supports:

- Dependency injection
- Embedded Lisp Interpreter with an FFI
- Implicit Flow

... all of which rely on a configuration file
Evaluating Concerto: The Setup

- Wrote a simple web application against YAWN
- Implemented three abstract interpreters using standard AI domains/techniques
- Compared the results of running the abstract interpreters alone and with Concerto
Evaluating Concerto: Abstract Interpreters

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Abstract Values</th>
<th>Heap/Object</th>
<th>Context-Sensitivity</th>
<th>Relational?</th>
<th>Path Sensitive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Bounds Checker</td>
<td>Pentagons</td>
<td>Allocation Site + Context</td>
<td>1-CFA</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Points-to-Analysis</td>
<td>None</td>
<td>Type-Based</td>
<td>None</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Taint Analysis</td>
<td>Taint Domain</td>
<td>Type-Based &amp; Access-Path</td>
<td>Caller Method</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Evaluating Concerto: Plain AI

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Analysis Time</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Bounds Checker</td>
<td>Timeout after 1 hour</td>
<td>2 false positives at timeout</td>
</tr>
<tr>
<td>Points-to Analysis</td>
<td>3.5 minutes</td>
<td>663 call-graph edges</td>
</tr>
<tr>
<td>Taint Analysis</td>
<td>Timeout after 1 hour</td>
<td>3 true positives, 6 false positives at timeout</td>
</tr>
</tbody>
</table>
Evaluating Concerto: Combined Interpretation

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Analysis Time</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Bounds Checker</td>
<td>9.18 seconds</td>
<td>Verified all array accesses</td>
</tr>
<tr>
<td>Points-to Analysis</td>
<td>5.26 seconds</td>
<td>266 call-graph edges</td>
</tr>
<tr>
<td>Taint Analysis</td>
<td>5.96 seconds</td>
<td>Found all true leaks</td>
</tr>
</tbody>
</table>
## Evaluating Concerto: Combined Interpretation

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Analysis Time</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Bounds Checker</td>
<td>9.18 seconds</td>
<td>Verified all array accesses</td>
</tr>
<tr>
<td>Points-to Analysis</td>
<td>5.26 seconds</td>
<td>266 call-graph edges</td>
</tr>
<tr>
<td>Taint Analysis</td>
<td>4.36 seconds</td>
<td>Found all true leaks</td>
</tr>
</tbody>
</table>

1/40\(^{th}\) the analysis time
## Evaluating Concerto: Combined Interpretation

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Analysis Time</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Bounds Checker</td>
<td>9.18 seconds</td>
<td>Verified all array accesses</td>
</tr>
<tr>
<td>Points-to Analysis</td>
<td>5.26 seconds</td>
<td>266 call-graph edges</td>
</tr>
<tr>
<td>Taint Analysis</td>
<td>2.69 seconds</td>
<td>Found all true leaks</td>
</tr>
</tbody>
</table>

- 1/40th the analysis time
- 2/3 fewer call graph edges
Evaluating Concerto: Combined Interpretation

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Analysis Time</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Bounds Checker</td>
<td>9.18 seconds</td>
<td>Verified all array accesses</td>
</tr>
<tr>
<td>Points-to Analysis</td>
<td>5.26 seconds</td>
<td>266 call-graph edges</td>
</tr>
<tr>
<td>Taint Analysis</td>
<td>5.96 seconds</td>
<td>Found all true leaks</td>
</tr>
</tbody>
</table>
Future Work

• Extend and scale Concerto to the full Java language

• Evaluate Concerto on real-world analyses and frameworks

• Generalize our formalisms to support other definitions of soundness
Concerto: A Framework for Combined Concrete and Abstract Interpretation

John Toman & Dan Grossman
University of Washington
The Time for Proof Reuse is Now!

Talia Ringer
With work by Nathaniel Yazdani, John Leo, and Dan Grossman
Dependent Types

Inductive list A :=
| nil : list A
| cons : A -> list A-> list A.

Inductive vector A :=
| nilV : vector A 0
| consV :
  forall (n : nat),
  A ->
  vector A n ->
  vector A (S n).
Dependent Types

Fixpoint len | := ...
  | nil => 0
  | cons a l => S (length l).

Definition lenV n v := n.

Theorem nil_cons :
 forall x l, nil <> cons x l.
Dependent Types

Theorem `app Nil r`:
for all \( A \) (\( l : \text{list} \ A \) ),
\( \text{app} \ A \ l \ \text{nil} = l \).

Proof.

... 

Qed.

Theorem `app Nil r V`:
for all \( A \) \( n \) (\( v : \text{vector} \ A \ n \) ),
\( \text{appV} \ A \ v \ \text{nilV} = v \).

Proof.

... 

Qed.
Dependent Types

\[
\begin{align*}
\text{appV } A \text{ v nilV } &= \text{v} \\
\text{lenV } (\text{appV } A \text{ v nilV}) &= \text{lenV v} \\
\text{lenV } v + \text{lenV } \text{ nilV} &= \text{lenV v} \\
\text{n } + \text{0 } &= \text{n}
\end{align*}
\]
Dependent Types

Theorem plus_n_0:
for all (n : nat),
  \( n + 0 = n \).

... reflexivity. \( \times \)
Packing with $\Sigma$
Packing with $\Sigma$

$$(n : \text{nat}) \ (v : \text{vector A n})$$
Packing with $\Sigma$

**Theorem** app_nil_rV:

forall $A$ ($v : \Sigma n . \text{vector} A n$),

appV $A$ $v$ (existT (vector $A$) 0 (nilV $A$)) = $v$.

**Proof.**

...

Qed.
Packing with $\Sigma$

```plaintext
existT (vector A) (S
  (projT1
    (vector_rect A (fun (n0 : nat) (_ : vector A n0) => {n1 : nat & vector A n1})
      (existT (vector A) O (nilV A))
      (fun (n0 : nat) (a0 : A) (_, vector A n0)) (IH : {n1 : nat & vector A n1}) =>
        existT (vector A) (S (projT1 IH)) (consV A (projT1 IH) a0 (projT2 IH)) n p)))
(consV A (projT1
  (vector_rect A (fun (n0 : nat) (_ : vector A n0) => {n1 : nat & vector A n1})
    (existT (vector A) O (nilV A))
    (fun (n0 : nat) (a0 : A) (_, vector A n0)) (IH : {n1 : nat & vector A n1}) =>
      existT (vector A) (S (projT1 IH)) (consV A (projT1 IH) a0 (projT2 IH)) n p)) a
(projT2
  (vector_rect A (fun (n0 : nat) (_ : vector A n0) => {n1 : nat & vector A n1})
    (existT (vector A) O (nilV A))
    (fun (n0 : nat) (a0 : A) (_, vector A n0)) (IH : {n1 : nat & vector A n1}) =>
      existT (vector A) (S (projT1 IH)) (consV A (projT1 IH) a0 (projT2 IH)) n p)) =
  existT (fun n0 : nat => vector A n0) (S n) (consV A n a p)
)```

Inductive vector A :=
| nilV : vector A 0
| consV :
  | forall (n : nat),
  | A ->
  | vector A n ->
  | vector A (S n).
“Basically a nightmare” ★★★★★
- Dominique Larchey-Wendlin, Proof Search Expert

“Almost no one should be using [them] for anything” ★★★★★
- Adam Chlipala, Author of “Certified Programming with Dependent Types”

“Not suitable for extended use” ★★★★★
- Emilio Jesús Gallego Arias, Coq Contributor
Proof Reuse to the Rescue
Definition \texttt{app A (l m : list A) := ...}

Theorem \texttt{app\_nil\_r:}
\[
\forall A \ (l : \text{list A}), \quad \text{app A l nil} = l.
\]
Proof.

...  

Qed.
Proof Reuse

Definition appV A (l m : Σ n . vector A n) := ...

Theorem app_nil_r:
  forall A (l : list A),
  app A l nil = l.

Proof.

... ...

Qed.
Proof Reuse

Definition \( \text{appV} \ A \ (l \ m : \Sigma \ n . \ \text{vector} \ A \ n) := \ldots \)

Theorem \( \text{app\_nil\_rV} \):

\[
\text{forall} \ (A : \text{Type}) \ (v : \text{sigT} \ (\text{vector} \ A)), \\
\text{appV} \ A \ v \ (\text{existT} \ (\text{vector} \ A) \ 0 \ (\text{nilV} \ A)) = v.
\]

Proof.

\[
\ldots
\]

Qed.
Proof Reuse

existT (vector A)
(S)
(proj1
 (vector_rect A (fun (n0 : nat) (v0 : vector A n0) => {n1 : nat & vector A n1}))
 (existT (vector A) (A n0) (fun (n0 : nat) (a0 : vector A n0) (IH : {n1 : nat & vector A n1}) =>
 existT (vector A) (S n) (consV A (proj1 IH) a0 (proj2 IH)) n p)))
(consV A
 (proj1
 (vector_rect A (fun (n0 : nat) (v0 : vector A n0) => {n1 : nat & vector A n1}))
 (existT (vector A) 0 (fun (n0 : nat) (a0 : vector A n0) (IH : {n1 : nat & vector A n1}) =>
 existT (vector A) (S n) (consV A (proj1 IH) a0 (proj2 IH)) n p)) a
 (proj2
 (vector_rect A (fun (n0 : nat) (v0 : vector A n0) => {n1 : nat & vector A n1}))
 (existT (vector A) n0 (fun (n0 : nat) (v0 : vector A n0) (IH : {n1 : nat & vector A n1}) =>
 existT (vector A) (proj1 IH) (consV A (proj1 IH) a0 (proj2 IH)) n p))) =
 existT (fun n0 : nat => {n : nat & vector A n0} (S n) (consV A n a p))
Proof Reuse: Ahead of its Time
Proof Reuse: Ahead of its Time
The ‘90s: Languages for Reuse

Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. LPAR ’94.

Proof Reuse: Ahead of its Time
2000s: Proof Reuse for Proof Engineers


Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. *FoSSaCS ’01*. 
Proof Reuse: Ahead of its Time
2000s: Proof Reuse for Proof Engineers

Proof Reuse: The Time is Now!
Proof Reuse: The Time is Now!

2010s: Technology at our Disposal (a sample)

Large Proof Developments
Domain-Specific Frameworks
Transport & HoTT
Ornaments
Machine Learning
Example-Based Synthesis
Proof Generalization
Proof Differencing
Proof Reuse: The Time is Now!

2010s: Technology at our Disposal (a sample)

Large Proof Developments
Domain-Specific Frameworks
Transport & HoTT
Ornaments
Machine Learning
Example-Based Synthesis
Proof Generalization
Proof Differencing
Proof Reuse: The Time is Now!

2010s: Ornaments at our Disposal

**Inductive** list A :=
| nil : list A
| cons : A -> list A -> list A.

**Inductive** vector A :=
| nilV : vector A 0
| consV :
  | forall (n : nat), A -> vector A n -> vector A (S n).
Proof Reuse: The Time is Now!
2010s: Ornaments at our Disposal

Definition \( \text{app} \ A \ (l \ m : \ \text{list} \ A) := \ldots \)

Theorem \( \text{app\_nil\_r} : \)
\[ \forall A \ (l : \ \text{list} \ A), \]
\[ \text{app} \ A \ l \ \text{nil} = l. \]

\ldots
Proof Reuse: The Time is Now!

2010s: Ornaments at our Disposal

Definition $\text{appV } A \ (l \ m : \Sigma \ n . \ \text{vector } A \ n) := \ldots$

Theorem $\text{app\_nil\_rV}$:
forall (A : Type) (v : sigT (vector A)),
$\text{appV } A \ v \ (\text{existT } (\text{vector } A) \ 0 \ (\text{nilV } A)) = v.$

\ldots
Proof Reuse: The Time is Now!
2010s: Ornaments at our Disposal


Proof Reuse: The Time is Now!
2010s: Technology at our Disposal (a sample)

- Large Proof Developments
- Domain-Specific Frameworks
- Transport & HoTT
- Ornaments
- Machine Learning
- Example-Based Synthesis
- Proof Generalization
- Proof Differencing
“Basically a nightmare” ★★★★★
- Dominique Larchey-Wendlin, Proof Search Expert

“Almost no one should be using [them] for anything” ★★★★☆
- Adam Chlipala, Author of “Certified Programming with Dependent Types”

“Not suitable for extended use” ★★★★☆
- Emilio Jesús Gallego Arias, Coq Contributor
“Mathematicians around the world could collaborate by depositing proofs and constructions in the computer, and ... it would be up to the computer to locate the equivalence between formulations and [to] transport the constructions from one context to another.”

- IAS Memorial Service on Vladimir Voevodsky’s Vision from 2006
Fluidics?
Abstraction Gap

Experiment

Microfluidic Chips
Outline

- Extensible Fluidic Semantics
- Hardware Abstraction
- Microfluidic Chips
Digital Microfluidics

Pros 😊
- General purpose
- Extensible
- Parallel

Cons 😞
- Hard to program
- Error prone
Programming microfluidic devices is hard!

- precision
- error handling
- location tracking
- hardware specific
- concurrency
- domain specific
- probabilistic results
- resource management
- parallelism
Outline

- Extensible Fluidic Semantics
- Hardware Abstraction
- Microfluidic Chips
What we want

No locations!

Automatic error handling!

```
def foo(a, b):
    # mix in 2:1 ratio
    ab = mix(a, b, 2)

    while get_pH(ab) > 7:
        heat(ab)
        acidify(ab)

Control flow!
```
Dynamism

def foo(a,b):
    # mix in 2:1 ratio
    ab = mix(a, b, 2)
    while get_pH(ab) > 7:
        heat(ab)
        acidify(ab)
Dynamism

On-the-fly error correction
Dynamism

Dynamic error correction

High level programming constructs

No static reasoning about resource usage
Where we are now

def foo(a,b):
    # mix in 2:1 ratio
    ab = mix(a, b, 2)

    while get_pH(ab) > 7:
        heat(ab)
        acidify(ab)
Outline

Extensible Fluidic Semantics

- Hardware Abstraction
- Microfluidic Chips
def foo(a,b,c):
    # mix in 2:1 ratio
    ab = mix(a, b, 2)

    while get_pH(ab) > 7:
        heat(ab)
        acidify(ab)

    ac = mix(a, c)

long running

Already consumed!
Volume Polymorphism

```python
def foo(a, b):
    # mix in 2:1 ratio
    ab = mix(a, b, 2)
    ab, _ = split(ab)

    while get_pH(ab) > 7:
        heat(ab)
        acidify(ab)

    return ab
```

- a: A, b: B
- ab: A + B, A = 2*B
- A + B > min_split
def foo(a, b):
    # mix in 2:1 ratio
    ab = mix(a, b, 2)
    state = Map Droplet {
        ph : Real
    }
    while get_pH(ab) > 7: 
        heat(ab)
        acidify(ab)
def foo(a,b):
    # mix in 2:1 ratio
    ab = mix(a, b, 2)

    while get_pH(ab) > 7:
        heat(ab)
        acidify(ab)

    state = Map Droplet {
        ph : Real
        temp : Real
        volume : Real
    }
def foo(a, b):
    # mix in 2:1 ratio
    ab = mix(a, b, 2)

    while get_pH(ab) > 7:
        heat(ab)
        acidify(ab)

    many intrinsic chemical properties of a sample
    procedures, not primitives
Termination?

while get_pH(ab) > 7:
    heat(ab)
    acidify(ab)

@ensures( abs(x.pH - retval) < 0.1 )
def get_pH(x):
    ...

@ensures( x.pH - old_x.pH > 0.5 )
def acidify(x):
    ...
Thanks!

Precision loss & approximation

Using chemical/biological models

HCI

Experimental design

misl.cs.washington.edu
Inferring and Asserting Distributed System Invariants

https://bitbucket.org/bestchai/dinv

Stewart Grant§, Hendrik Cech¶, Ivan Beschastnikh§
University of British Columbia§, University of Bamberg¶
Distributed Systems are pervasive

- Graph processing
- Stream processing
- Distributed databases
- Failure detectors
- Cluster schedulers
- Version control
- ML frameworks
- Blockchains
- KV stores
- ...
Distributed Systems are Notoriously Difficult to Build

- Concurrency
- No Centralized Clock
- Partial Failure
- Network Variance
Today’s state of the art (building robust dist. sys)

**Verification** - [ (verification) IronFleet SOSP’15, VerdiPLDI’15, Chapar POPL’16, (modeling), Lamport et.al SIGOPS’02, Holtzman IEEE TSE’97]

**Bug Detection** - [MODIST NSDI’09, Demi NSDI’16,]

**Runtime Checkers** - [D3S NSDI’18,]

**Tracing** - [PivotTracing SOSP’15, XTrace NSDI’07, Dapper TR’10,]

**Log Analysis** - [ShiViz CACM ’16]

**Takeaway:** Little work has been done to infer distributed specs automatically
Avenger [SRDS’11], CSight [ICSE’14]
Design goal: handle **real** distributed systems

**Wanted: distributed state invariants**

Make the fewest assumptions about the system as possible.

- N nodes
- Message passing
- Lossy, reorderable channels
- Joins and failures
Goal: Infer key correctness and safety properties

Mutual exclusion:
\( \forall \text{nodes} \ \text{InCritical} \leq 1 \)

Key Partitioning:
\( \forall \text{nodes} \ i, j \ \text{keys}_i \neq \text{keys}_j \)

Running Example

Diagram showing a network of nodes with operations like Get Lock, Ping, and key assignments. The client interacts with nodes 1, 2, and 3, with keys distributed in the ranges [0:49], [50:99], and [100:149] respectively.
Today’s talk

- Automatic distributed invariant inference (techniques & challenges)
- Runtime checking: distributed assertions
- Evaluation: 4 large scale distributed systems
Capturing Distributed State Automatically

1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection

- Log Relevant Variables
- Send Message (Add vector clock)
- Receive Message (Remove vector clock)
Reasoning About Global State

- Consistent Cuts
- Ground States
- State Bucketing
Reasoning About Global State

- Consistent Cuts
- Ground States
- State Bucketing
Reasoning About Global State

- Consistent Cuts
- Ground States
- State Bucketing
Reasoning About Global State

- Consistent Cuts
- Ground States
- State Bucketing

Node_3_InCritical == True
Node_2_InCritical != Node_3_InCritical
Node_2_InCritical == Node_1_InCritical
Distributed Asserts

- Distributed asserts enforce invariants at runtime
- Snapshots are constructed using approximate synchrony
- Asserter constructs global state by aggregating snapshots
Evaluated Systems

Etcd: Key-Value store running Raft - 120K LOC

Serf: large scale gossiping failure detector - 6.3K LOC

Taipei-Torrent: Torrent engine written in Go - 5.8L LOC

Groupcache: Memcached written in Go - 1.7K LOC
## Etcd ~ 120K Lines of Code

<table>
<thead>
<tr>
<th>System and Targeted property</th>
<th>Dinv-inferred invariant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raft Strong Leader principle</td>
<td>( \forall \text{ follower } i, \text{ len}(\text{leader log}) \geq \text{ len}(i's \log) )</td>
<td>All appended log entries must be propagated by the leader</td>
</tr>
<tr>
<td>Raft Log matching</td>
<td>( \forall \text{ nodes } i, j \text{ if } i-\log[c] = j-\log[c] \rightarrow \forall (x \leq c), i-\log[x] = j-\log[x] )</td>
<td>If two logs contain an entry with the same index and term, then the logs are identical on all previous entries.</td>
</tr>
<tr>
<td>Raft Leader agreement</td>
<td>If ( \exists \text{ node } i, \text{ s.t. } i \text{ leader, than } \forall j \neq i, j \text{ follower} )</td>
<td>If a leader exists, then all other nodes are followers.</td>
</tr>
</tbody>
</table>

Injected Bugs for each invariant caught with assertions
Limitations and future work

Limitations

- Dinv’s dynamic analysis is incomplete
- Ground state sampling is poor on loosely coupled systems
- Temporal invariants are not supported

Future work

- Extend analysis to temporal invariants
- Bug Isolation
- Distributed test case generation
- Mutation testing/analysis based on mined invariants
Contributions

Analysis for distributed Go systems

- Automatic **distributed state** invariant inference
  - Static identification of distributed state
  - Automatic static instrumentation
  - Post-execution merging of distributed states
- Runtime checking: distributed assertions

Repo: https://bitbucket.org/bestchai/dinv

Demo: https://www.youtube.com/watch?v=n9fH9ABJ6S4