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.


PNW PLSE 2018


Welcome!

The Morning

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

The Afternoon (Part 1)

12:00pm	Lunch	
12:15pm		
12:30pm	Lunch Talk: Project Everest: Theory meets reality	Jonathan Protzenko
12:45pm		
1:00pm	Break	
1:15pm	Featured Talks: Continuously Integrated Verified Cryptography	Mike Dodds
1:30pm	Helena: A web automation language for end users	Sarah Chasins, Ras Bodik
1:45pm	Sinking Point	Bill Zorn, Dan Grossman
2:00pm	Verified Extraction with Native Types	Stuart Pernsteiner, Eric Mullen, James R. Wilcox, Zachary Tatlock, Dan Grossman

The Afternoon (Part 2)

2:15pm	Lightning Talk Session	
2:30pm	Poster Session	
2:45pm		
3:00pm		
3:15pm		
3:30pm	Break	
3:45pm	Featured Talk: Why not both? Applications of variational programming	Eric Walkingshaw
4:00pm	Chapel Comes of Age: Productive Parallelism at Scale	Brad Chamberlain
4:15pm	Musical Ornaments	John Leo
4:30pm	Incrementalization with Data Structures	Calvin Loncaric, Michael D. Ernst
4:45pm	Wrap up and Close	
5:00pm	Group Dinner (self pay)	

Poster Session

1	Cosette: An Automated Prover for SQL	Shumo Chu, Alvin Cheung, Dan Suciu
2	Dependency Capture for Reproducible Builds	Martin Kellogg
3	Sloth: Locating Sites for Repetitive Edits with Lazy Concrete Pattern Matching on Trees	Remy Wang, Rashmi Mudduluru, Hadar Greinsmark
4	Time-Travel Diagnostics for Node.js/JavaScript	Mark Marron
5	Synchronizing the asynchronous	Thomas Henzinger, Bernhard Kragl, Shaz Qadeer
6	Experimental Design as Programs	Eunice Jun, Jared Roesch, Sarah Chasins
7	Designing Compilers and Synthesis Tools for 3D Printing	Chandrakana Nandi
8	Adaptive Program Ranking	Chenglong Wang
9	A Formal Model of Polymorphism and Inference in Rust	Joseph Eremondi, Ron Garcia
10	Relay: an IR for differentiable programming	Jared Roesch, Tianqi Chen, Steven Lyubomirsky, Zachary Tatlock, Josh Pollock, Logan Weber
11	Automated Verification of Cryptographic Protocols	James Bornholt, Ernie Cohen, K. Rustan M. Leino
12	Interactively Debugging Distributed Systems	Doug Woos
13	Helping Designers Explore the Space of Layout Variations with Constraints	Amanda Swearngin, Andrew J. Ko, James Fogarty

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, UO

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, UO

Microsoft Research

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

Abstract Interpretation: The Real World

```
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[l] = k;
    if k<1000 then
      write k;
    else
      k = i;
    endif
    a[l+1] = a[l] / 2;
  done
  write a[i];
}
```

Loops

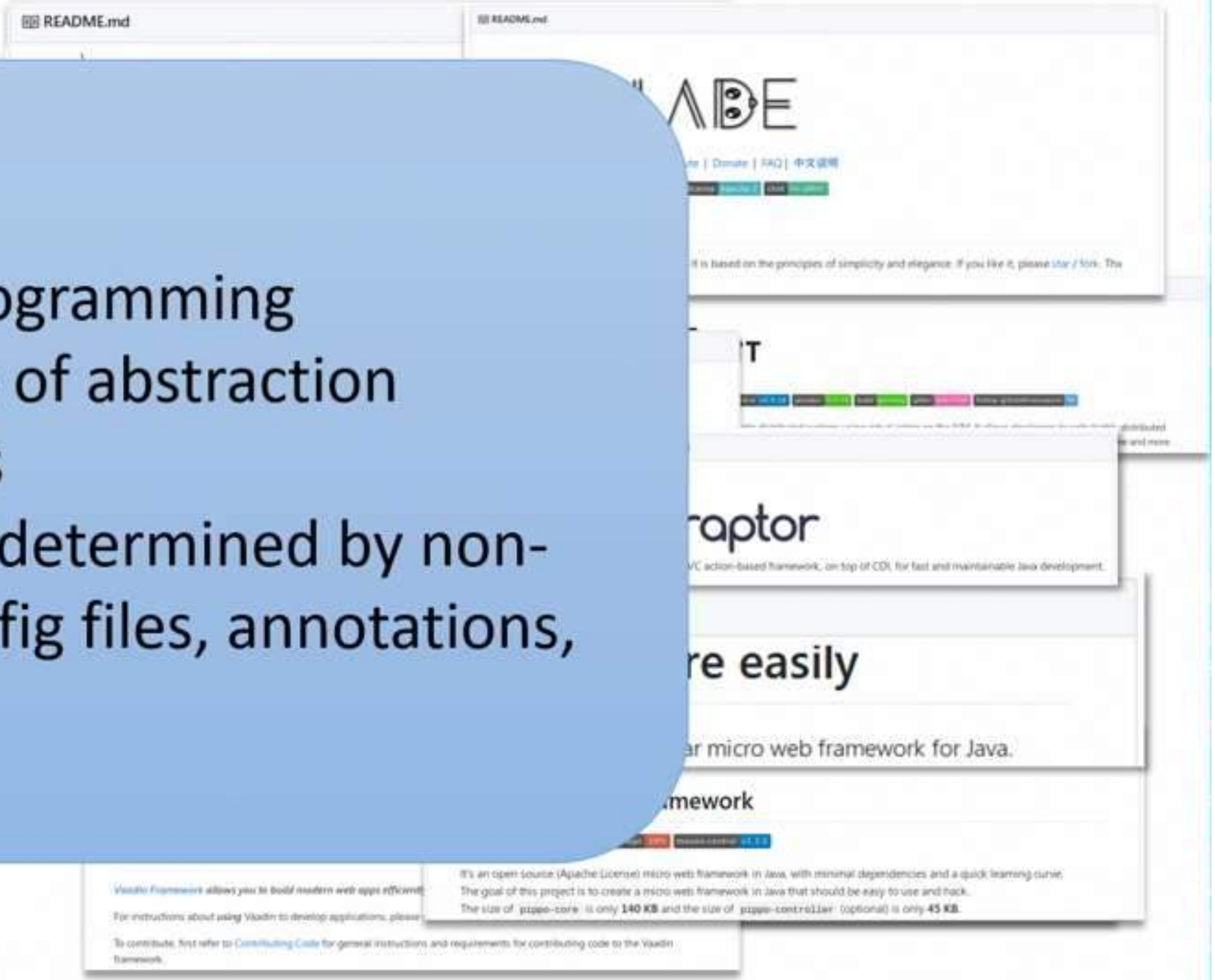
Arrays

Linear Arithmetic

Abstract Interpretation: The Real World

```
proc(a[100]) {
  read i,j;
  k = i;
  a[1] = 0;
  for l=1 to
    for m=i
      k = k
    done
    a[l] = k
    if k<100
      write
    else
      k = i;
    endif
    a[l+1] =
  done
  write a[i];
}
```

- 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. Soundiness

   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. Soundiness

Relies on unrealistic assumptions about the use of metaprogramming

2. Pessimistic (“Sound’

…sly imprecise in …ractice

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

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 configu
- Use multi abstractio
- Minimal b

Applications

- Focused on a fixed set of
- , more d" code
- ning, ps

A single, unified analysis strategy will not work

Exhaustive Path Exploration

Abstract Interpretation

Picking the Right Tool

Frameworks

- Extreme flexibility, driven by config
- Use multi abstractio
- Minimal b

Applications

- Focused on a fixed set of
- , more d" code
- ning, ps

Why not both?

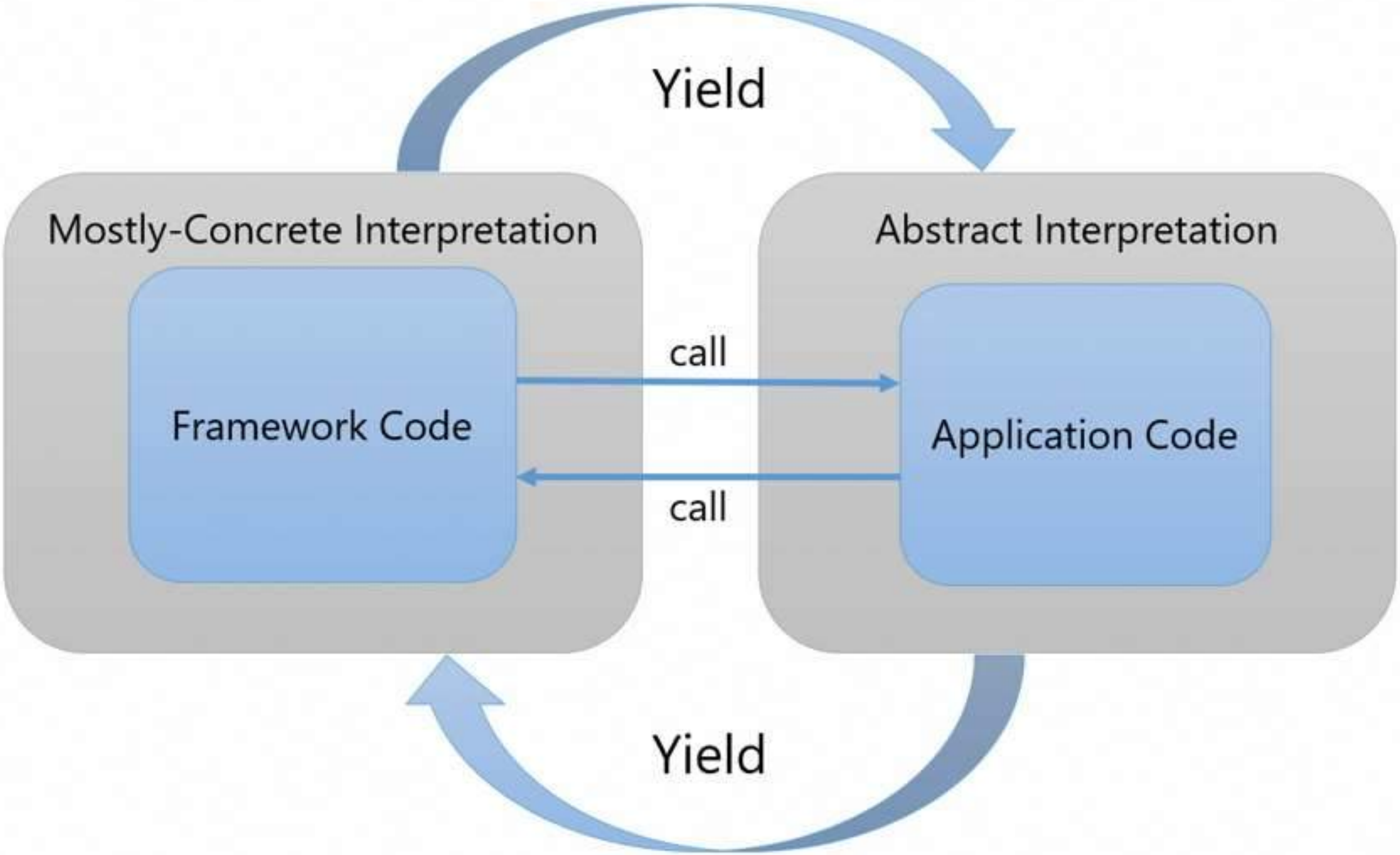
Exhaustive Path Exploration

Abstract Interpretation

Concerto

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

Concerto


Yield
Mostly-Concrete Interpretation
Framework Code
Abstract Interpretation
Application Code
call
call
Yield

Concerto by Example

Framework Code

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

Concerto by Example

(Mostly-)Concrete Interpreter

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

Concerto by Example

<u>(Mostly-)Concrete Interpreter</u>

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

Concerto by Example

(Mostly-)Concrete Interpreter

```
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

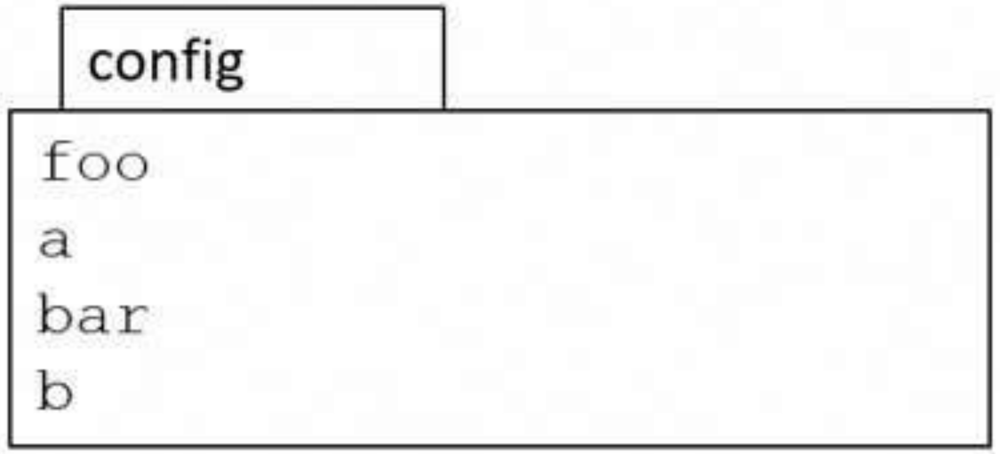
(Mostly-)Concrete Interpreter

```
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

```
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

```
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

```
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

<u>(Mostly-)Concrete Interpreter</u>

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

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

$[m \mapsto \{\text{"foo"} \mapsto \text{"a"}, \text{"bar"} \mapsto \text{"b"}\}]$

config

```
foo
a
bar
b
```

Concerto by Example

(Mostly-)Concrete Interpreter

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

Concerto by Example

(Mostly-)Concrete Interpreter

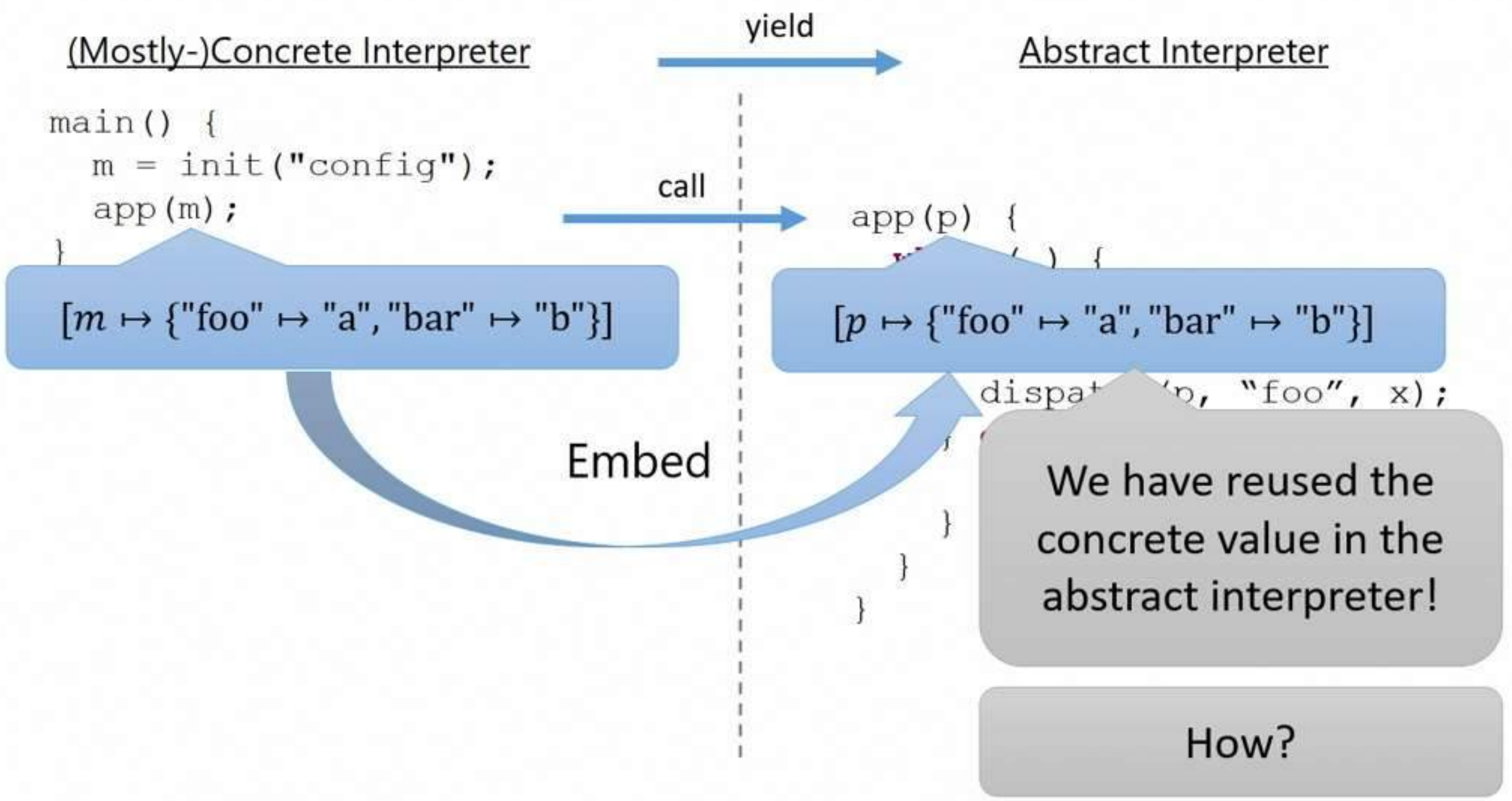
```
main() {
  m = init("config");
  app(m);
}
```

call

Application Code

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

Concerto by Example



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.

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

```
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
...
```

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()) {
    ... 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.

<u>Framework Code</u>

```
init(f) {
  conf = open(f);
  m = {};
  whil  (!conf.eof()) {
                            d();
```

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

<u>Application Code</u>

```
app(p) {
  while(…) {
    x = ★;
    if(x > 0) {
}
```

```
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.

<u>Framework Code</u>

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

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

<u>Application Code</u>

```
app(p) {
  while(…) {
    x = ★;
    if(   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 = {};
  whil  (!conf.eof()) {
```

```
type int
```

Application Code

```
app(p) {
  while(…) {
    x = ★;
    if(    0) {
```

```
type int = …|−1|0|1|…
add: int→int →int
greater: int→int→bool
...
```

Concerto by Example

(Mostly-)Concrete Interpreter

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

Abstract Interpreter

```
app(p) {
```

$[p \mapsto \{\text{"foo"} \mapsto \text{"a"}, \text{"bar"} \mapsto \text{"b"}\}]$

```
      dispatch(p, "foo", x);
    } else {
      dispatch(p, "bar", 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);
    } else {
      dispatch(p, "bar", x);
    }
  }
}
```

Concerto by Example

<u>(Mostly-)Concrete Interpreter</u>

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

<u>Abstract Interpreter</u>

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

$[p \mapsto \{"foo" \mapsto "a", "bar" \mapsto "b"\}, x \mapsto \{+\}]$

Concerto by Example

<u>(Mostly-)Concrete Interpreter</u>

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

<u>Abstract Interpreter</u>

```
app(p) {
  while(…) {
    x = ★;
    if(x > 0) {
      dispatch(p, "foo", x);
```

$[p \mapsto \{\text{"foo"} \mapsto \text{"a"}, \text{"bar"} \mapsto \text{"b"}\}, x \mapsto \{+\}]$

```
  }
}
```

Simple signedness domain

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);
```

$[p \mapsto \{"foo" \mapsto "a", "bar" \mapsto "b"\}, x \mapsto \{+\}]$

```
  }
}
```

call

Concerto by Example

(Mostly-)Concrete Interpreter ← yield — **Abstract Interpreter**

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


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

← call

```
app(p) {
  while(…) {
    x = ★;
    if(x > 0) {
      dispatch(p, "foo", x);
```

$[p \mapsto \{\text{"foo"} \mapsto \text{"a"}, \text{"bar"} \mapsto \text{"b"}\}, x \mapsto \{+\}]$

```
  }
}
```

Concerto by Example

(Mostly-)Concrete Interpreter ← yield ← Abstract Interpreter

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

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

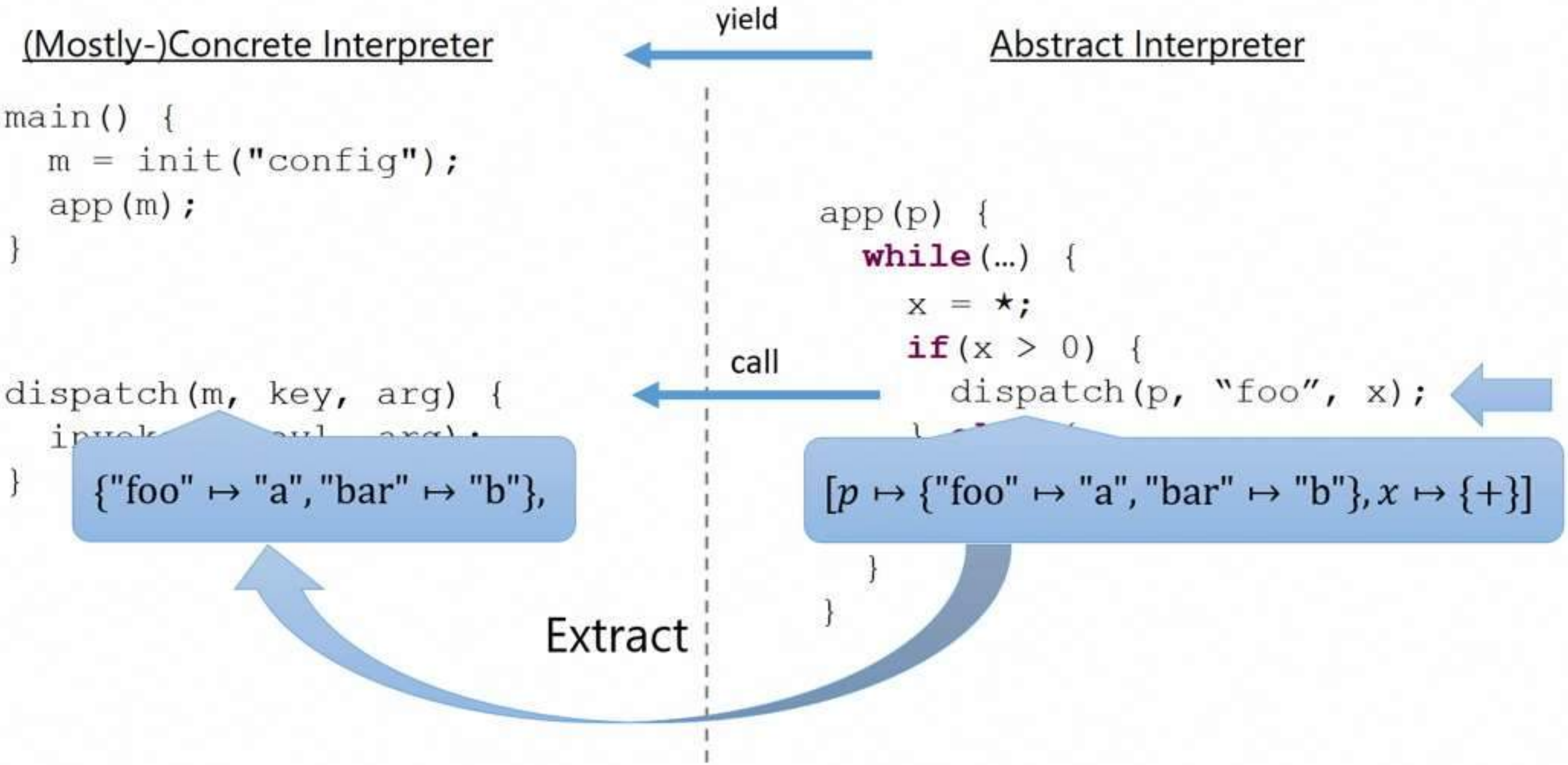
?

call ←

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

    }
}
```

$[p \mapsto \{"foo" \mapsto "a", "bar" \mapsto "b"\}, x \mapsto \{+\}]$

Concerto by Example



Concerto by Example

yield ←

<u>(Mostly-)Concrete Interpreter</u>

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

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

"foo"

<u>Abstract Interpreter</u>

call ←

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

$[p \mapsto \{\text{"foo"} \mapsto \text{"a"}, \text{"bar"} \mapsto \text{"b"}\}, x \mapsto \{+\}]$

Concerto by Example

yield ←

<u>(Mostly-)Concrete Interpreter</u>

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


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

?

<u>Abstract Interpreter</u>

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

call ←

$[p \mapsto \{\text{"foo"} \mapsto \text{"a"}, \text{"bar"} \mapsto \text{"b"}\}, x \mapsto \{+\}]$

Concerto by Example

yield

<u>(Mostly-)Concrete Interpreter</u>

<u>Abstract Interpreter</u>

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

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

{+}

call

```
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 ← yield — Abstract Interpreter

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

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

{+}

Reuse the abstract value

← call

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

$[p \mapsto \{\text{"foo"} \mapsto \text{"a"}, \text{"bar"} \mapsto \text{"b"}\}, x \mapsto \{+\}]$

Concerto by Example

(Mostly-)Concrete Interpreter

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


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

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

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

```
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

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

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

```
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

<u>(Mostly-)Concrete Interpreter</u>

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

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

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

Reflectively invokes the named procedure

call →

<u>Abstract Interpreter</u>

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

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

Concerto by Example

(Mostly-)Concrete Interpreter —yield→ Abstract Interpreter

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

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

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

Reflectively invokes the named procedure

call →

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

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

Concerto by Example

(Mostly-)Concrete Interpreter —yield→ Abstract Interpreter

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

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

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

Reflectively invokes the named procedure

call →

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

?

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

Concerto by Example

(Mostly-)Concrete Interpreter —yield→ Abstract Interpreter

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

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

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

Reflectively invokes the named procedure

call →

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

?

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

Concerto by Example

(Mostly-)Concrete Interpreter —yield→ Abstract Interpreter

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

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

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

Reflectively invokes the named procedure

call →

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

{+}

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

Concerto by Example

(Mostly-)Concrete Interpreter —yield→ Abstract Interpreter

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

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

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

Reflectively invokes the named procedure

call →

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

$\{+\}$

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

Provably unreachable

Concerto by Example

<u>(Mostly-)Concrete Interpreter</u>

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

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

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

call?

<u>Abstract Interpreter</u>

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

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

Concerto by Example

(Mostly-)Concrete Interpreter

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

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

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

call?

Precise, concrete semantics and representation

Abstract Interpreter

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

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

Concerto by Example

<u>(Mostly-)Concrete Interpreter</u>

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

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

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

Precise, concrete semantics and representation

<u>Abstract Interpreter</u>

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

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

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

Target Language	**Java**
Interfaces	Exceptions
(Dynamically Sized) Arrays	Strings
Integers	Concurrency
Downcasts	Static Fields
Reflection API	Static Initializers
Loops	Inheritance
I/O	float/short/…

Proof of Concept Implementation

Target Language	**Java**
Interfaces	~~Exceptions~~
(Dynamically Sized) Arrays	~~Strings~~
Integers	~~Concurrency~~
Downcasts	~~Static Fields~~
Reflection API	~~Static Initializers~~
Loops	~~Inheritance~~
I/O	~~float/short/…~~

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

Analysis	Abstract Values	Heap/Object	Context-Sensitivity	Relational?	Path Sensitive?
Array Bounds Checker	Pentagons	Allocation Site + Context	1-CFA	✓	✓
Points-to Analysis	*None*	Type-Based	*None*	✗	✗
Taint Analysis	Taint Domain	Type-Based & Access-Path	Caller Method	✗	✗

Evaluating Concerto: Plain AI

Analysis	Analysis Time	Results
Array Bounds Checker	Timeout after 1 hour	2 false positives at timeout
Points-to Analysis	3.5 minutes	663 call-graph edges
Taint Analysis	Timeout after 1 hour	3 true positives, 6 false positives at timeout

Evaluating Concerto: Combined Interpretation

Analysis	Analysis Time	Results
Array Bounds Checker	9.18 seconds	Verified all array accesses
Points-to Analysis	5.26 seconds	266 call-graph edges
Taint Analysis	5.96 seconds	Found all true leaks

Evaluating Concerto: Combined Interpretation

Analysis	Analysis Time	Results
Array Bounds Checker	9.18 seconds	Verified all array accesses
Points-to Analysis	5.26 seconds	266 call-graph edges
Taint Analysis	[illegible] seconds	Found all true leaks

1/40th the analysis time

Evaluating Concerto: Combined Interpretation

Analysis	Analysis Time	Results
Array Bounds Checker	9.18 seconds	Verified all array accesses
Points-to Analysis	5.26 seconds	266 call-graph edges
Taint Analysis	[illegible] seconds	[illegible]nd all true leaks

1/40th the analysis time

2/3 fewer call graph edges

Evaluating Concerto: Combined Interpretation

Analysis	Analysis Time	Results
Array Bounds Checker	9.18 seconds	Verified all array accesses
Points-to Analysis	5.26 seconds	266 call-graph edges
Taint Analysis	5.96 seconds	Found all true leaks

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

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

```
Definition lenV n v :=
  n.
```

Dependent Types

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

```
Theorem app_nil_rV:
  forall A n (v : vector A n),
    appV A v nilV = v.
Proof.
  ...
Qed.
```

Dependent Types

appV A v nilV = v

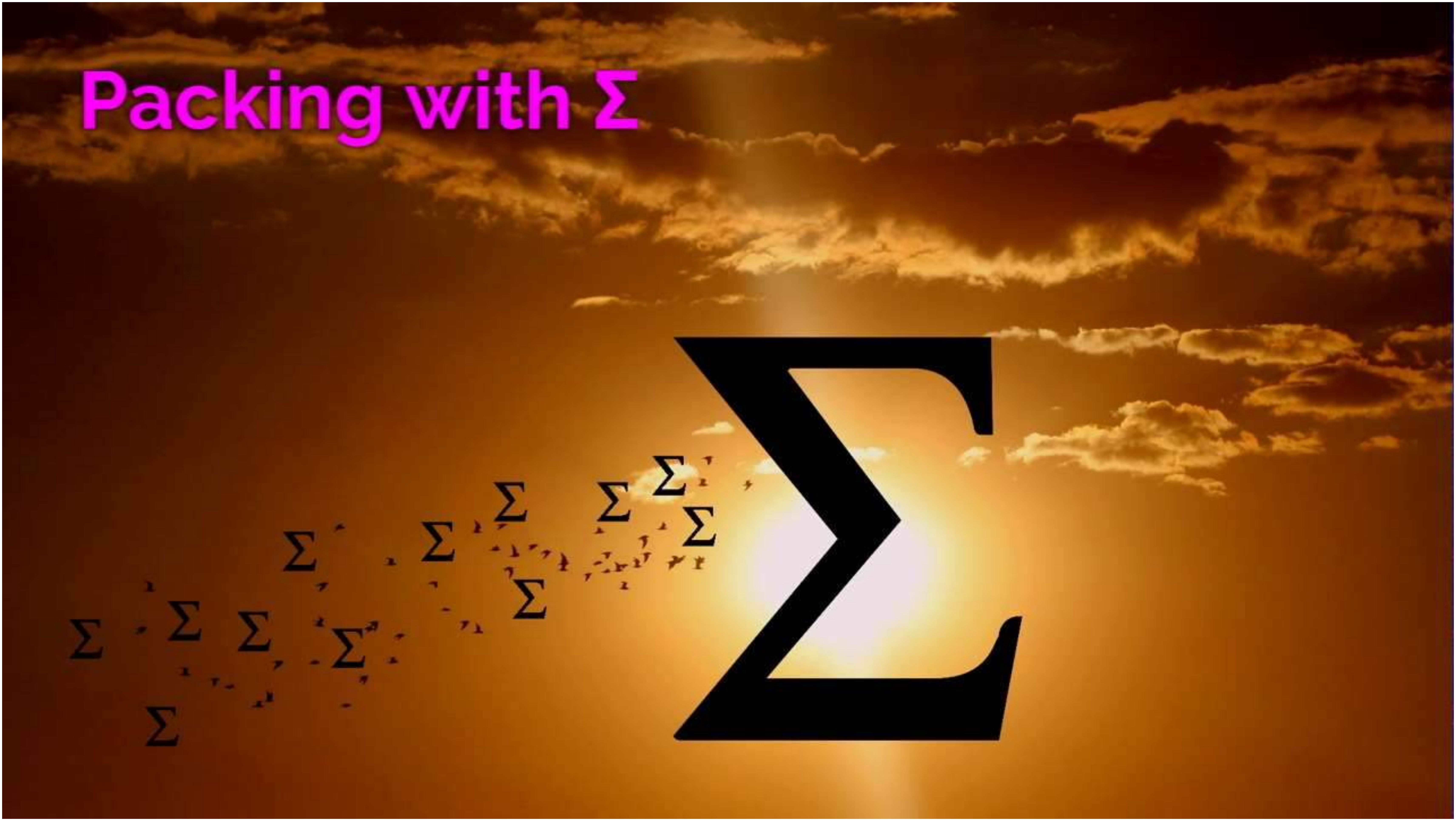
lenV (appV A v nilV) = lenV v

lenV v + lenV nilV = lenV v

n + 0 = n

Dependent Types

Theorem plus_n_0:
forall (n : nat),
n + 0 = n.
… reflexivity. ✗


Packing with Σ

Packing with Σ

(n : nat) (v : vector A n)

Packing with Σ

```
Theorem app_nil_rV:
  forall A (v : Σ n . vector A n),
    appV A v (existT (vector A) 0 (nilV A)) = v.
Proof.
  ...
Qed.
```

Packing with Σ

```
______________________________________(1/1)
existT (vector A)
  (S
     (projT1
        (vector_rect A (fun (n0 : nat) (_ : vector A n0) => {n1 : nat & vector A n1})
           (existT (vector A) 0 (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) 0 (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) 0 (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

Proof Reuse

```
Definition app A (l m : list A) := ...

Theorem app_nil_r:
  forall A (l : list A),
    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 appV A (l m : Σ n . vector A n) := ...

Theorem app_nil_rV:
  forall (A : Type) (v : sigT (vector A)),
    appV A v (existT (vector A) 0 (nilV A)) = v.
Proof.
  ...
Qed.
```

Proof Reuse

```
________________________________________(1/1)
existT (vector A)
  (S
     (projT1
        (vector_rect A (fun ([illegible]t) (_ : v[illegible] n0) => {n1 : nat & vector A n1})
           (existT (vector A) [illegible] A))
           (fun (n0 : nat) (a0 [illegible] : [illegible] n0) (IH : {n1 : nat & vector A n1}) =>
            existT (vector A) (S[illegible]1[illegible]consV A (projT1 IH) a0 (projT2 IH))) n p)))
  (consV A
     (projT1
        (vector_rect A (fun (n0 : [illegible] vector A n0) => {n1 : nat & vector A n1})
           (existT (vector A) 0 ([illegible]
           (fun (n0 : nat) (a0 [illegible]tor A n0) (IH : {n1 : nat & vector A n1}) =>
            existT (vector A) [illegible] (consV A (projT1 IH) a0 (projT2 IH))) n p)) a
     (projT2
        (vector_rect A (fun[illegible]t) [illegible]tor A n0) => {n1 : nat & vector A n1})
           (existT (vecto[illegible]ilV A))
           (fun (n0 : na[illegible]A) (_ : v[illegible] n0) (IH : {n1 : nat & vector A n1}) =>
            existT (vec[illegible] (projT1 IH[illegible]onsV A (projT1 IH) a0 (projT2 IH))) n p))) =
existT (fun n0 : nat =[illegible] A n0) (S n) (c[illegible]V A n a p)
```

Proof Reuse:
Ahead of its Time

Proof [partially obscured] ime

The '90s: La[partially obscured]

Amy Felty [partially obscured] zation and reuse [partially obscured]

Joshua E[partially obscured] i. A logical fra[partially obscured] euse. SIGSOFT [partially obscured]

Proof Reuse: Ahead of its Time

The '90s: Languages for Reuse

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

Joshua E. Caplan and Mehdi T. Harandi. A logical framework for software proof reuse. **SIGSOFT SE Notes '95**.

Proo ne

2000s: P

Nicolas ng
data st natural
numbe

Gilles B
isomor ent
type th

Proof Reuse: Ahead of its Time

2000s: Proof Reuse for Proof Engineers

Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. **TYPES 2000**.

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

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMark challenge. **TPHOLs '05**.

Proof Reuse:
The Time is Now!

Proo !

2010s:

Large F
Domair
Transp
Ornam
Machin
Examp
Proof C
Proof D

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 app A (l m : list A) := ...

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

Proof Reuse: The Time is Now!

2010s: Ornaments at our Disposal

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

Theorem app_nil_rV:
  forall (A : Type) (v : sigT (vector A)),
    appV A v (existT (vector A) 0 (nilV A)) = v.
...
```

Proof Reuse: The Time is Now!

2010s: Ornaments at our Disposal

Conor McBride. Ornamental algebras, algebraic ornaments. **2010.**

Thomas Williams and Didier Rémy. A principled approach to ornamentation in ML. **POPL 2018.**

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”

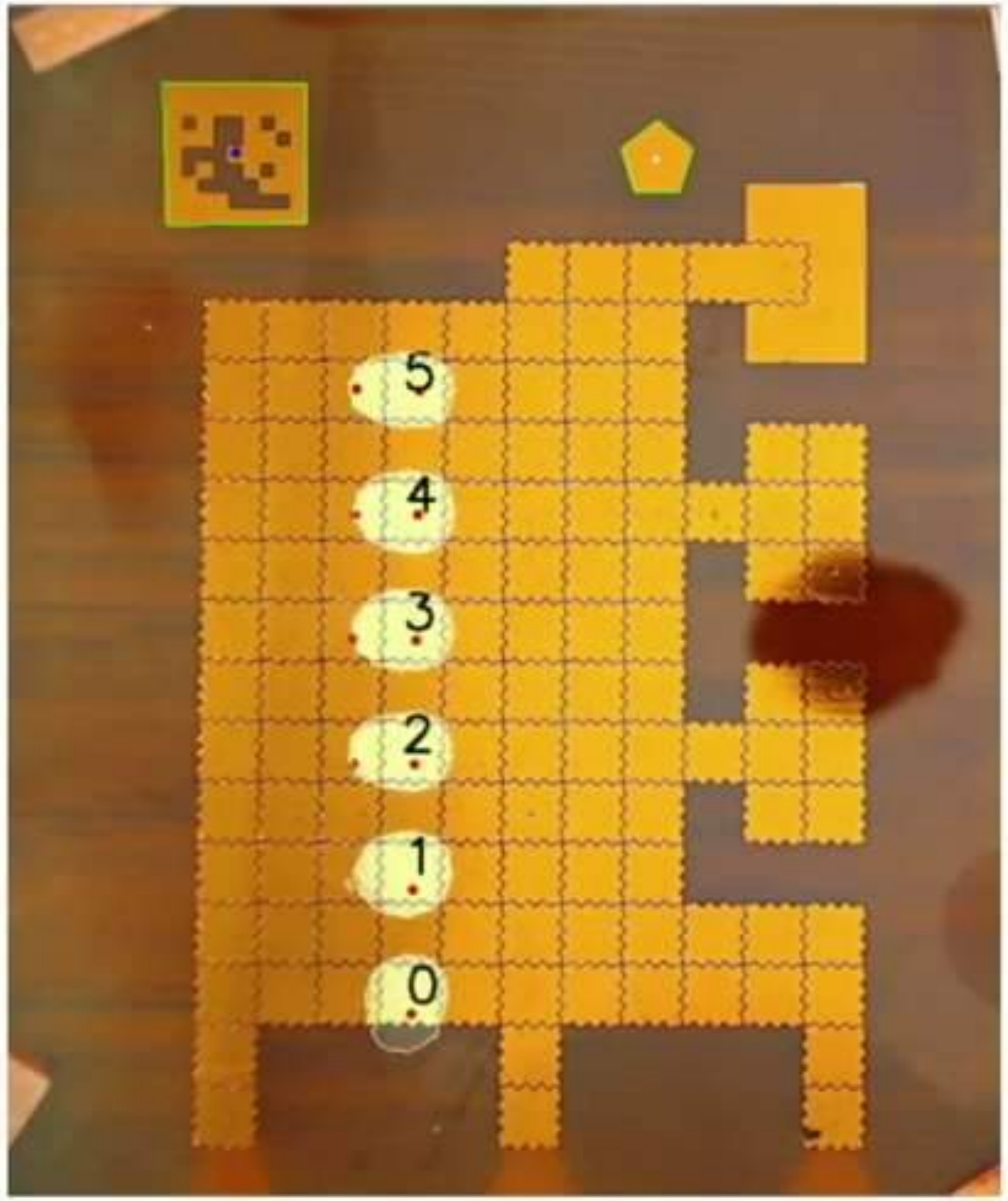
“Not suitable for extended use”

- 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

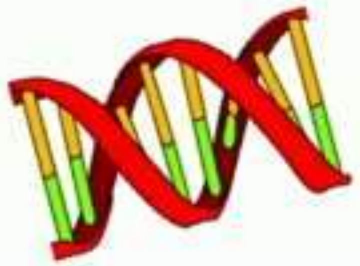
Fluidics?



Abstraction Gap

Experiment

Microfluidic Chips



Molecular Computing	Experiment	Medical Diagnostics
Synthetic DNA	Chemistry	Medicine

Extensible Fluidic Semantics

Hardware Abstraction

Microfluidic Chips

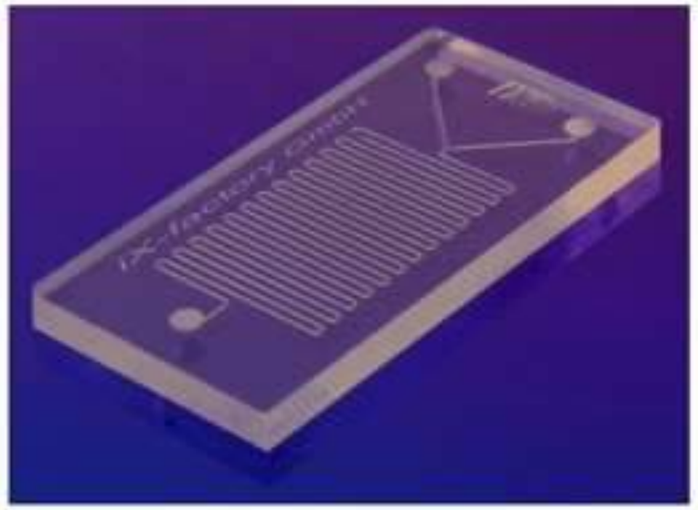
Outline

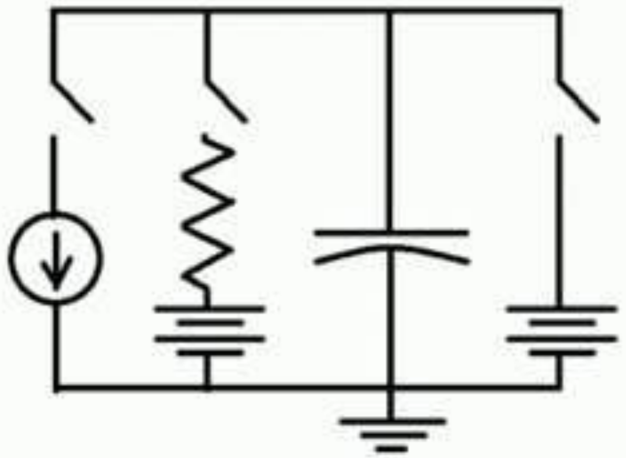
Extensible Fluidic Semantics

Hardware Abstraction

Microfluidic Chips

Microfluidics



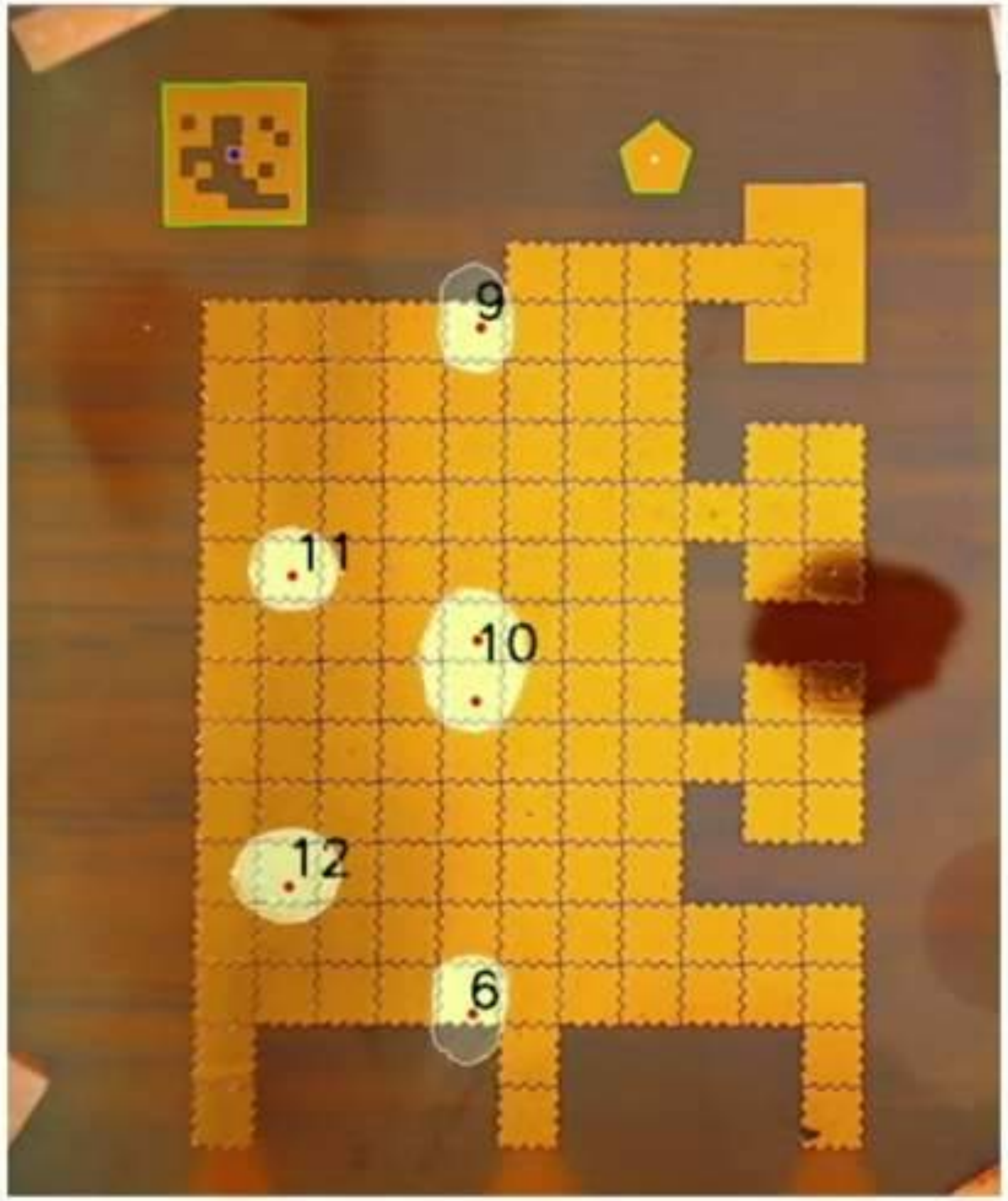
Digital Microfluidics

Pros 👍

- General purpose
- Extensible
- Parallel

Cons 👎

- Hard to program
- Error prone



Programming microfluidic devices is hard!

precision

error handling

location tracking

hardware specific

resource management

parallelism

concurrency

domain specific

probabilistic results

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!

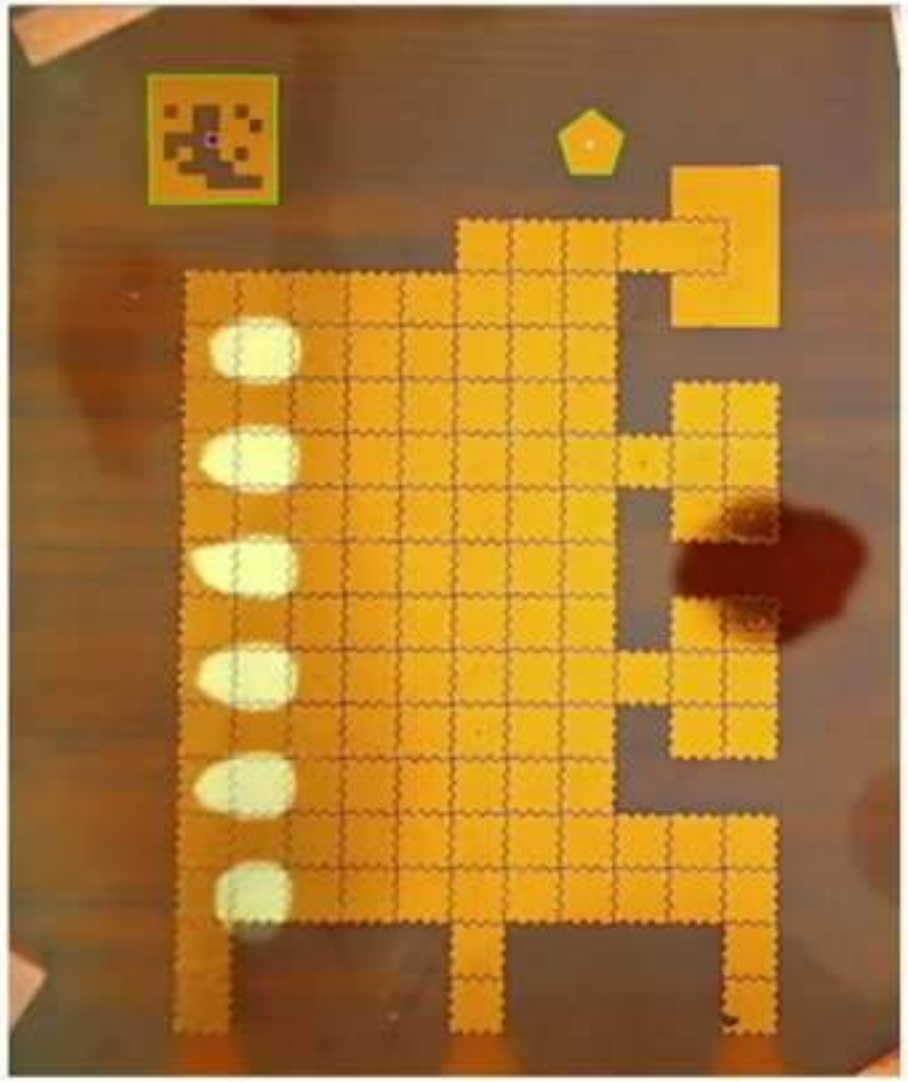
Dynamism

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

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

data dependent
control flow

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

Linearity

```
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

```
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

Termination?

```
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
}
```

Other Stuff?

```
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
}
```

Termination?

```
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

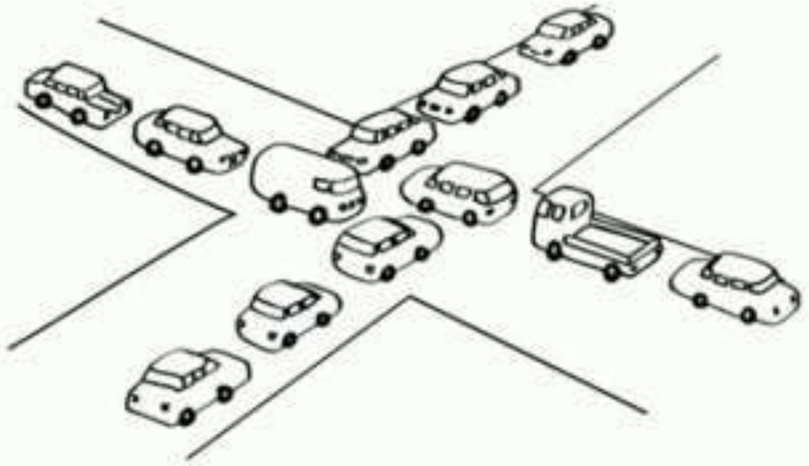
Stewart Grant[§], Hendrik Cech[¶], Ivan Beschastnikh[§]
University of British Columbia[§], University of Bamberg[¶]

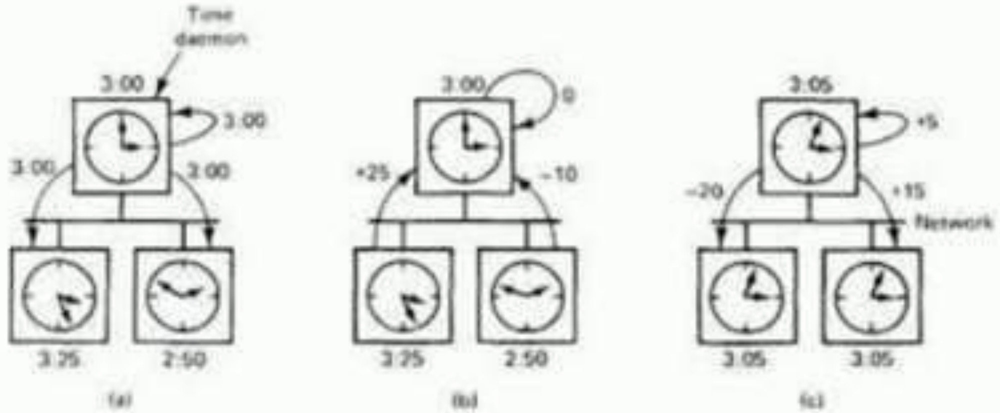
Distributed Systems are pervasive

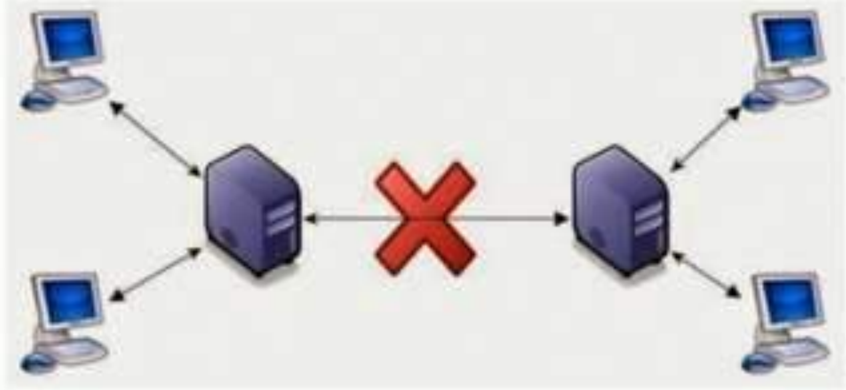
- Graph processing
- Stream processing
- Distributed databases
- Failure detectors
- Cluster schedulers
- Version control
- ML frameworks
- Blockchains
- KV stores
- ...

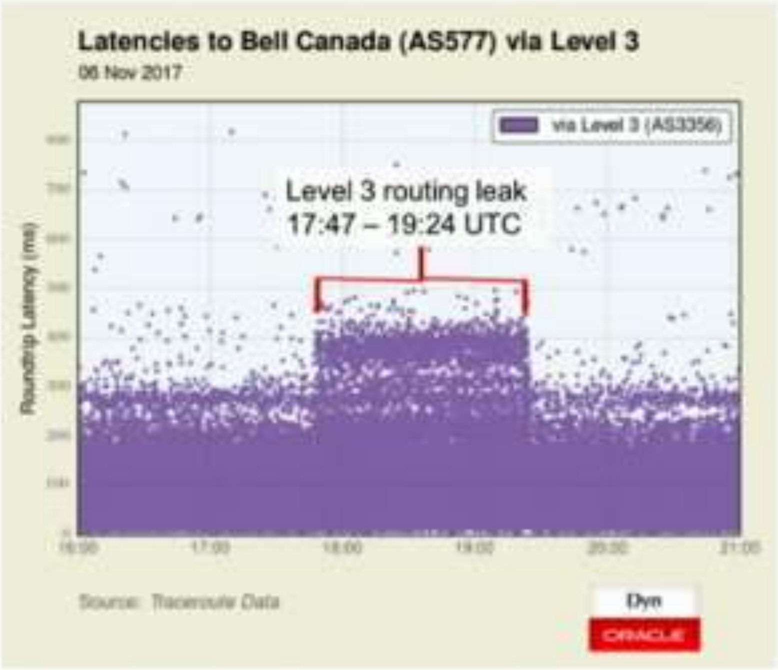
Distributed Systems are Notoriously Difficult to Build

- 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**]**

← **Require Specifications**

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.

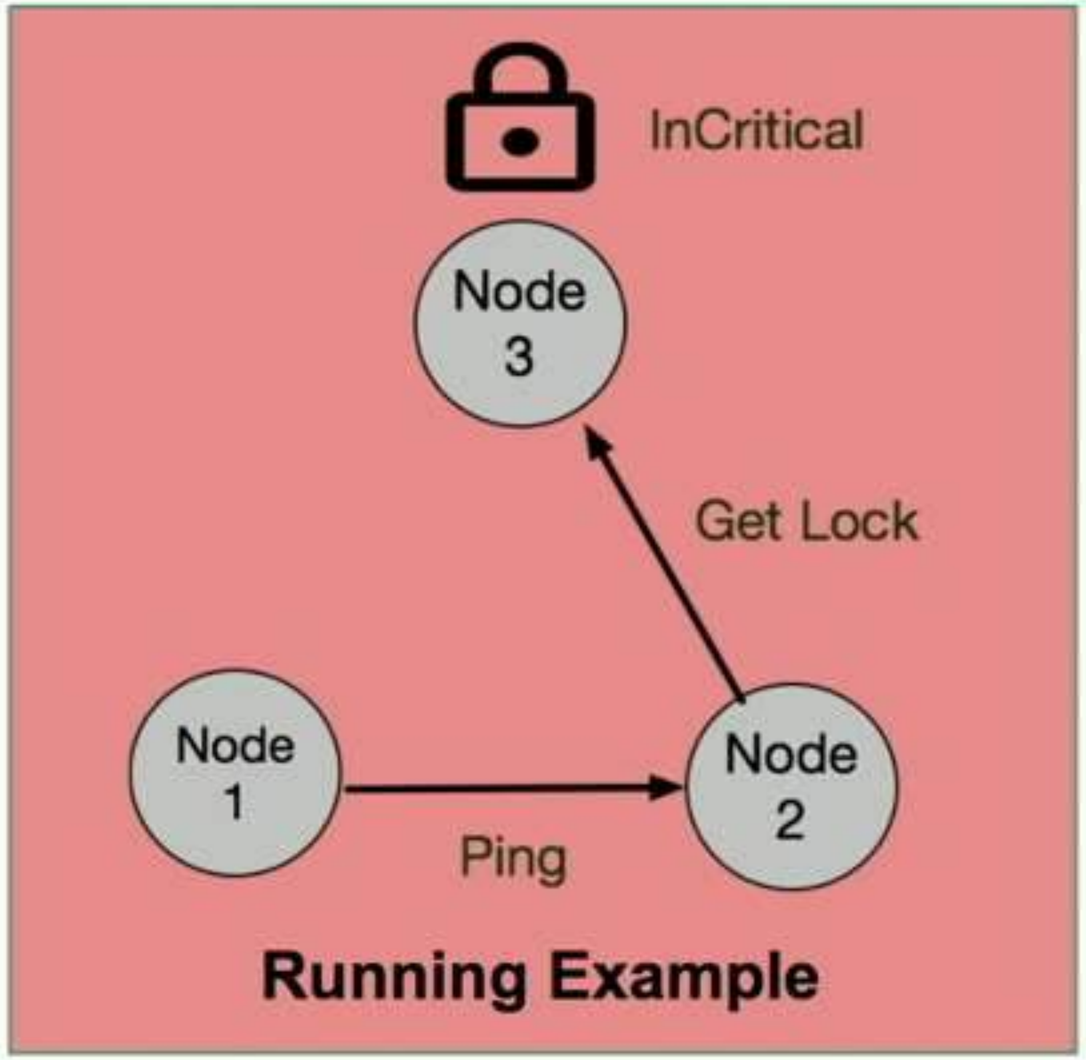
- N nodes
- Message passing
- Lossy, reorderable channels
- Joins and failures

Serf

Goal: Infer key correctness and safety properties

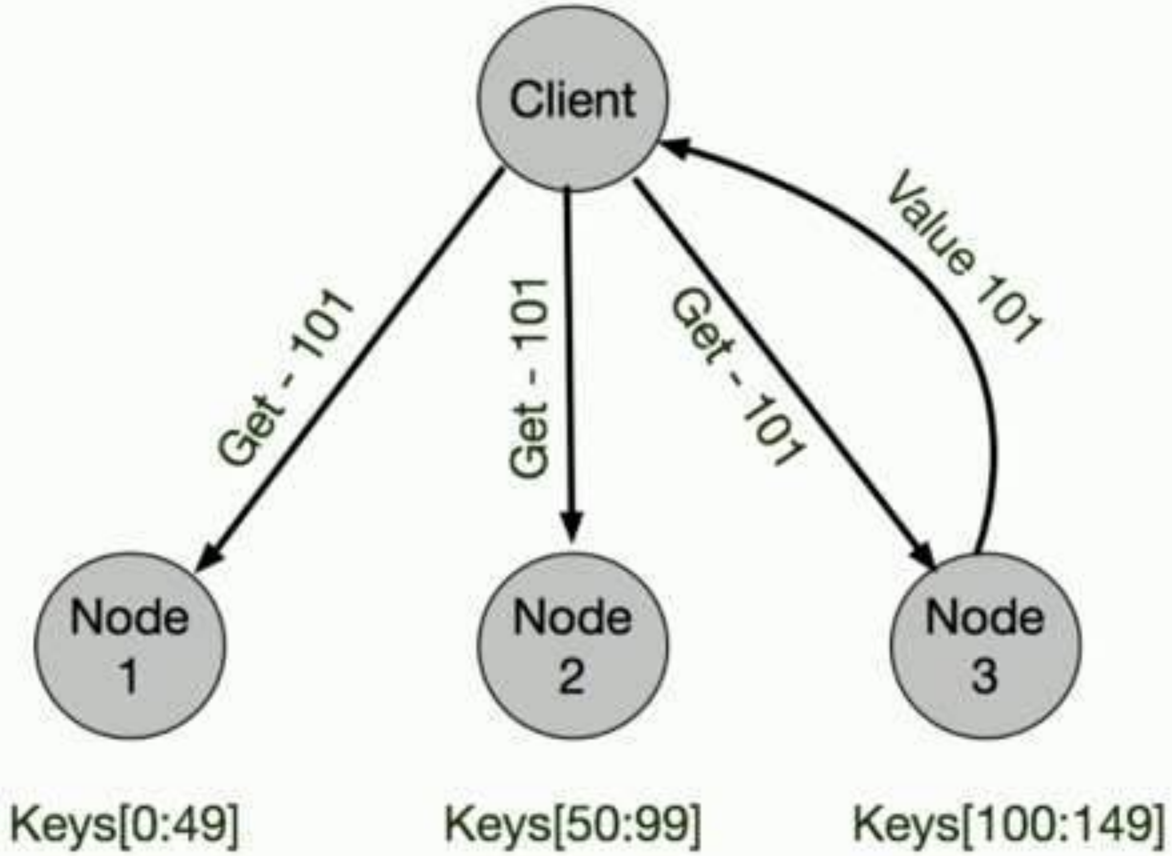
Mutual exclusion:

`∀nodes InCritical <= 1`



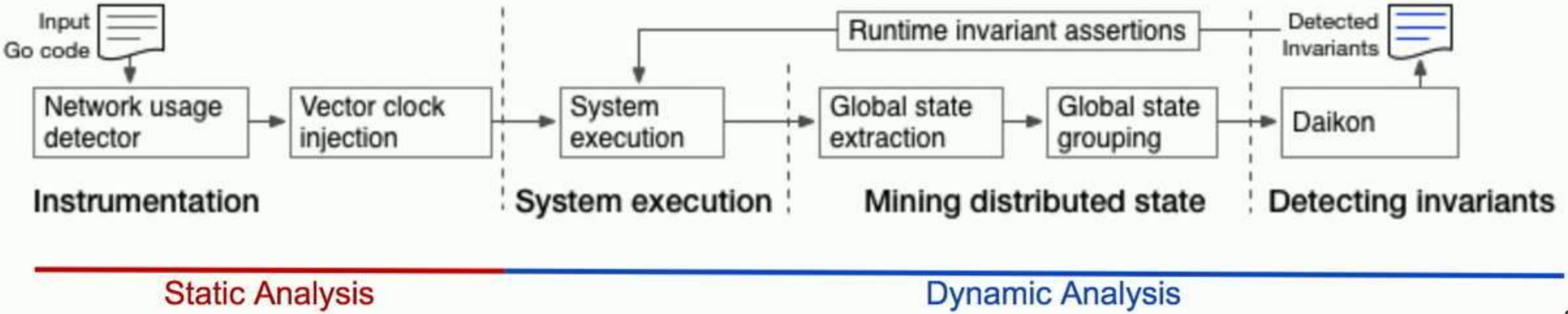
Key Partitioning:

∀nodes i, j keys_i != keys_j



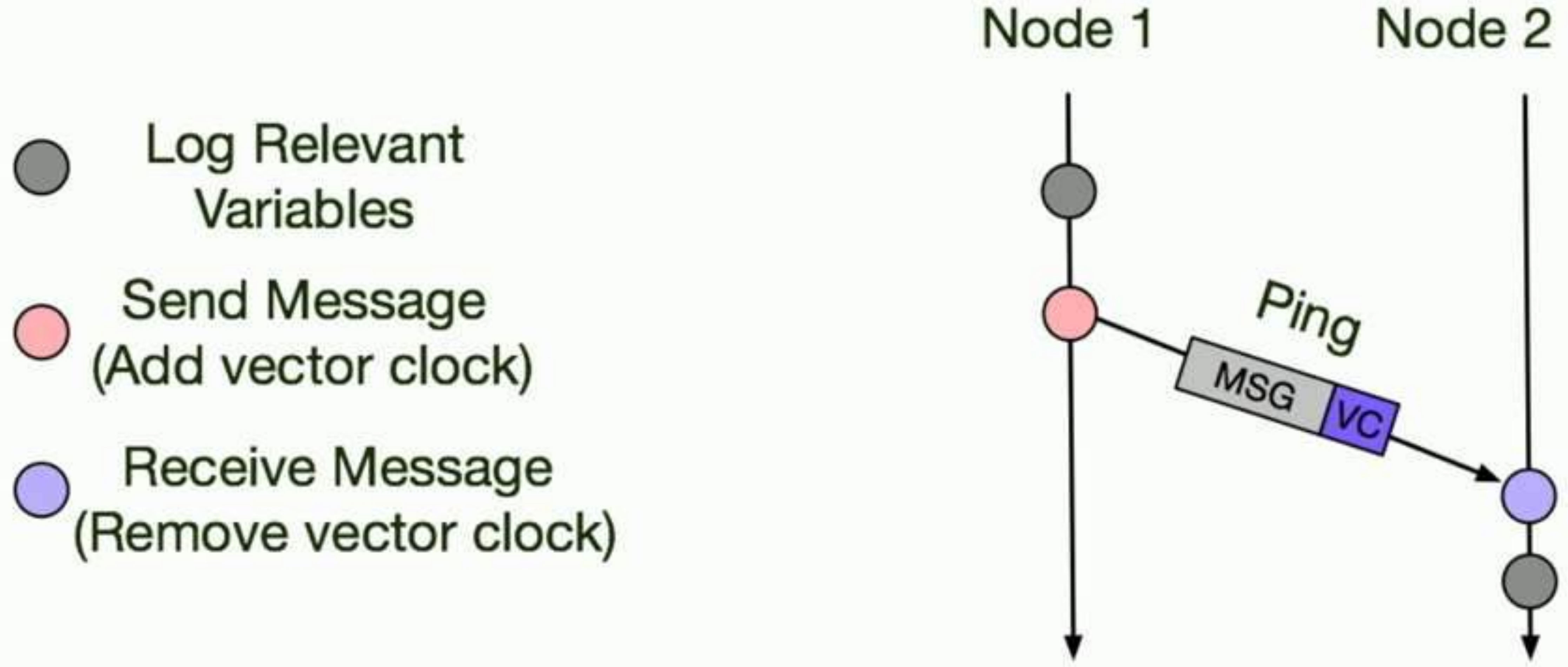
Today’s talk

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



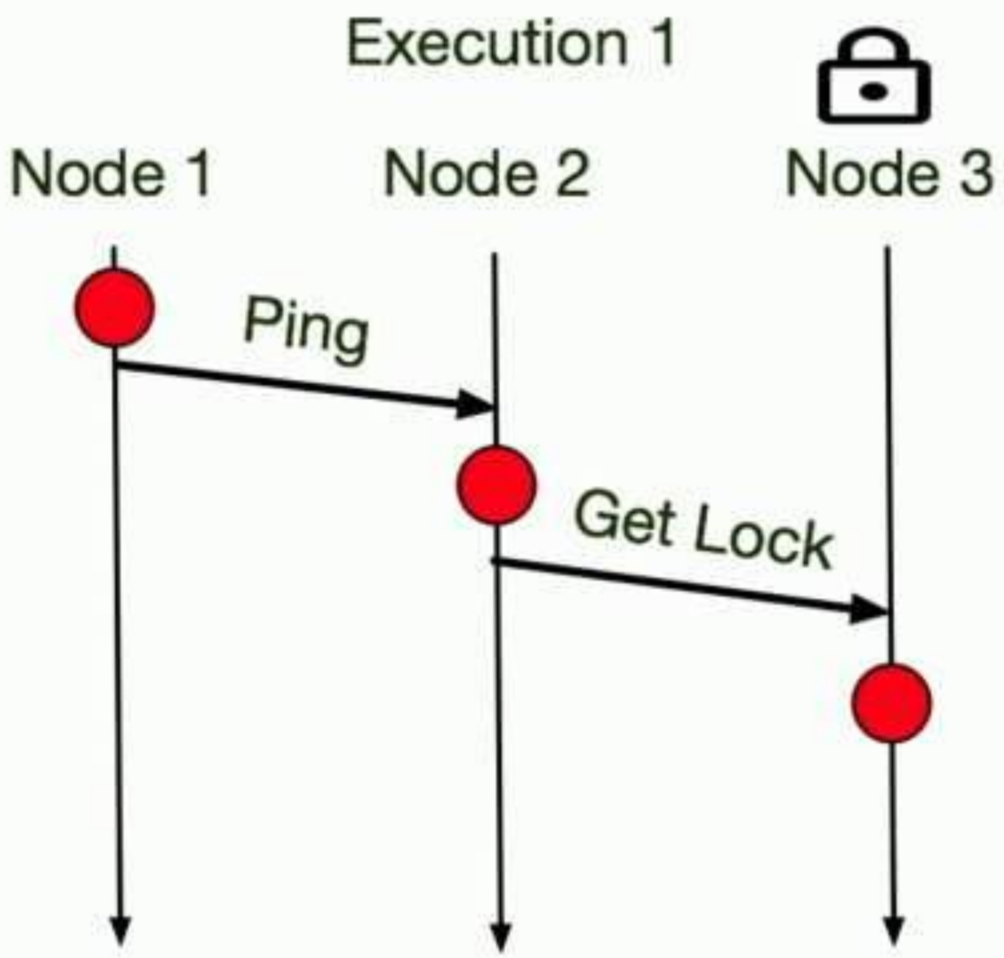
Capturing Distributed State Automatically

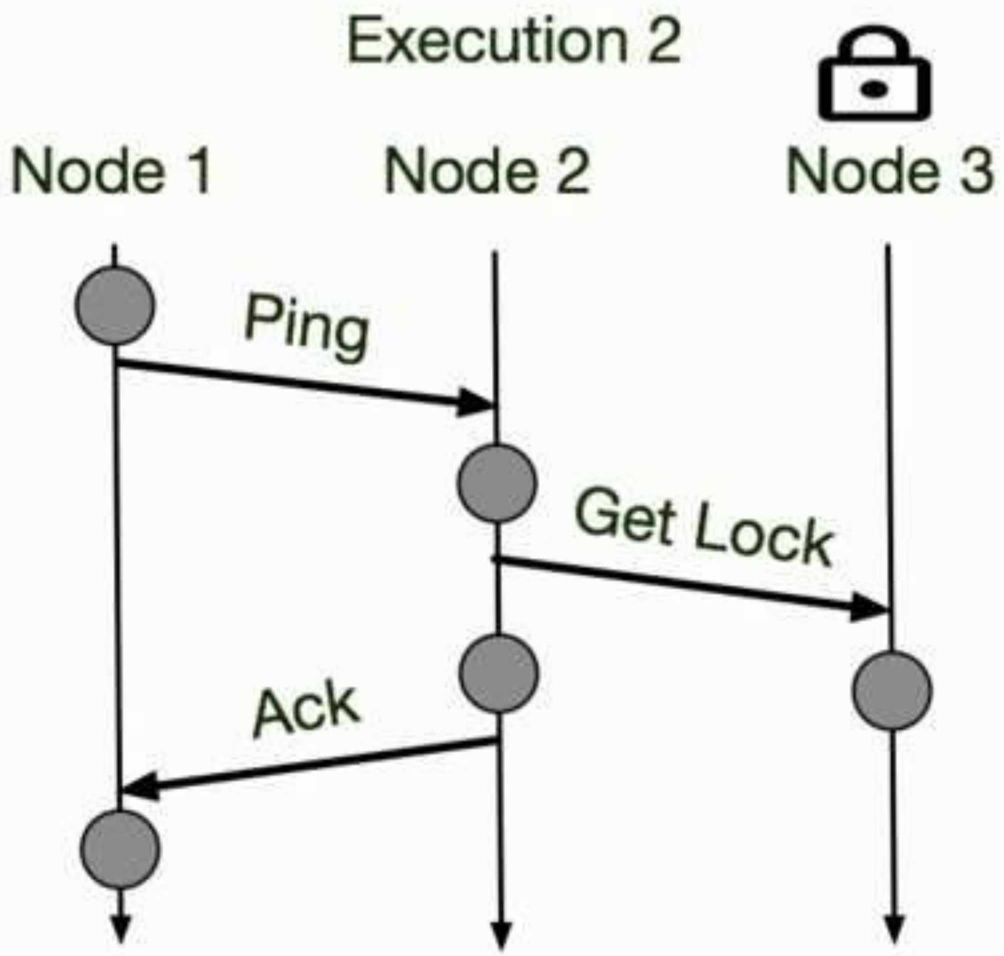
1. Interprocedural Program Slicing
2. Logging Code Injection
3. Vector Clock Injection



Reasoning About Global State

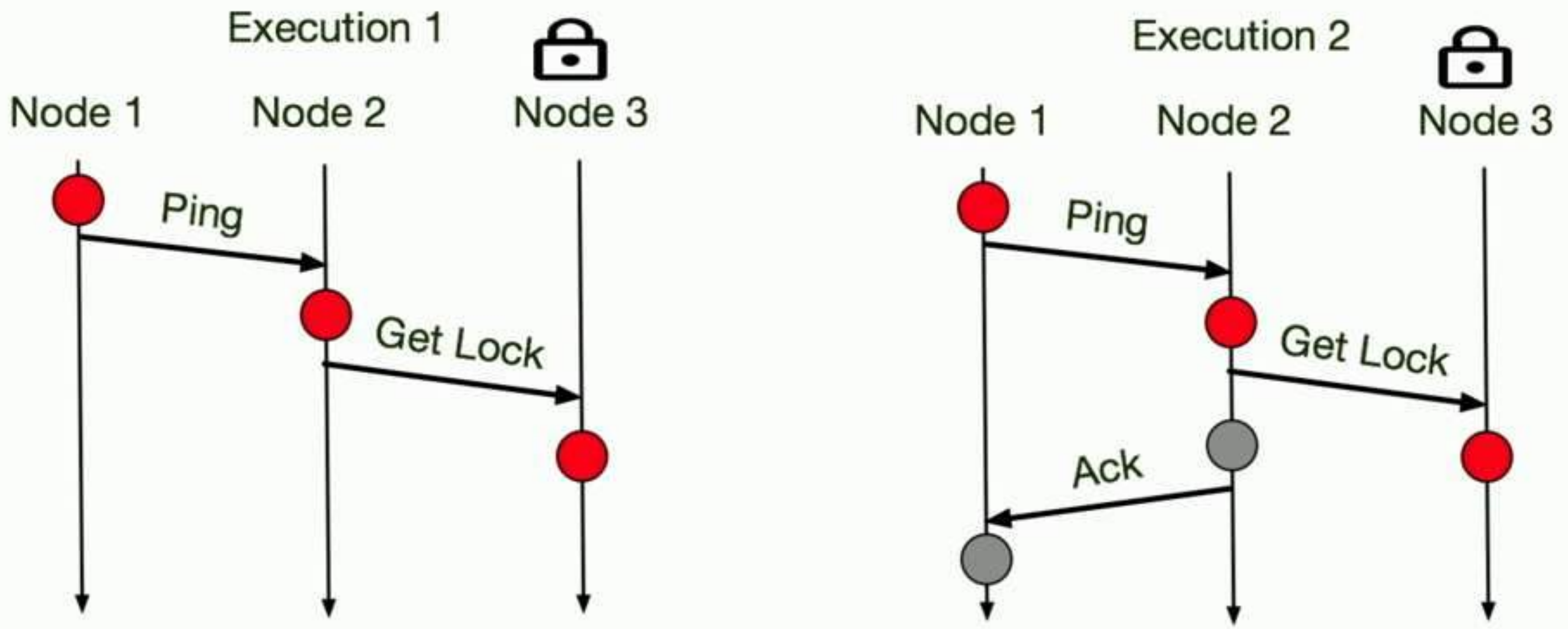
- Consistent Cuts
- Ground States
- State Bucketing





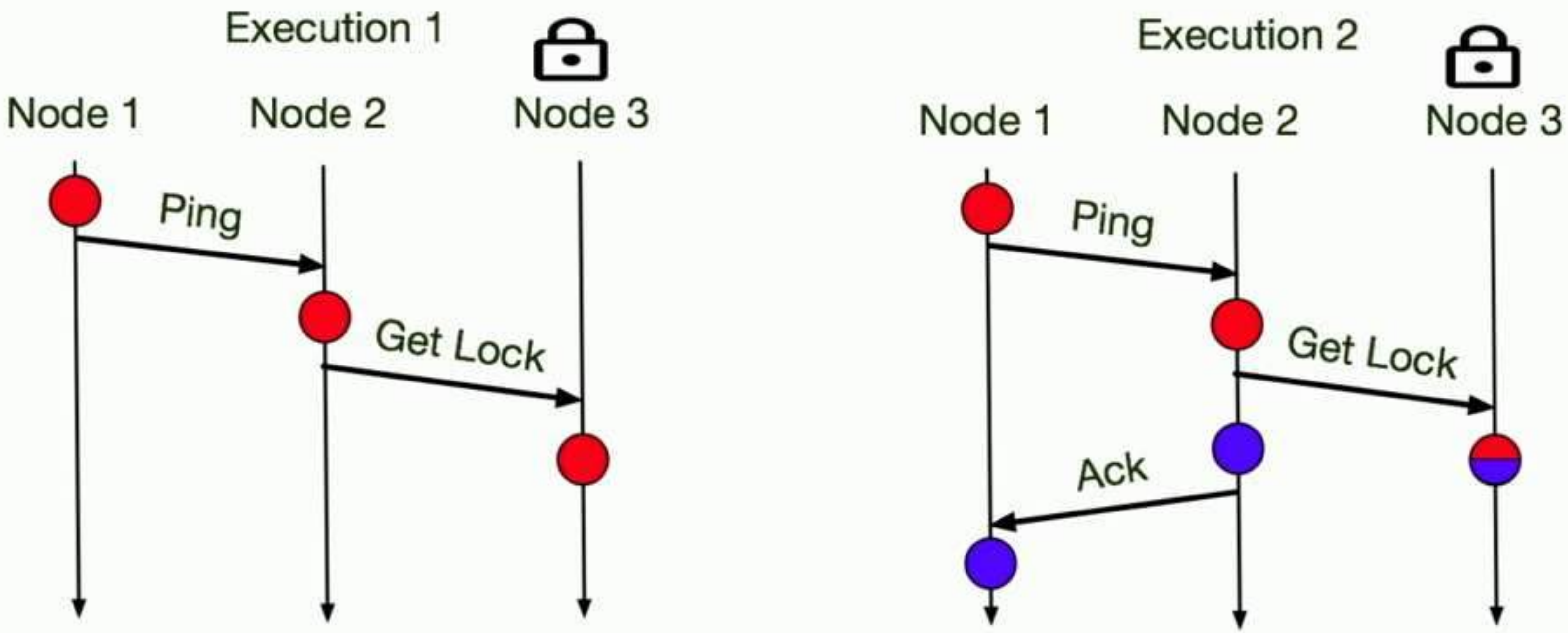
Reasoning About Global State

- Consistent Cuts
- Ground States
- State Bucketing



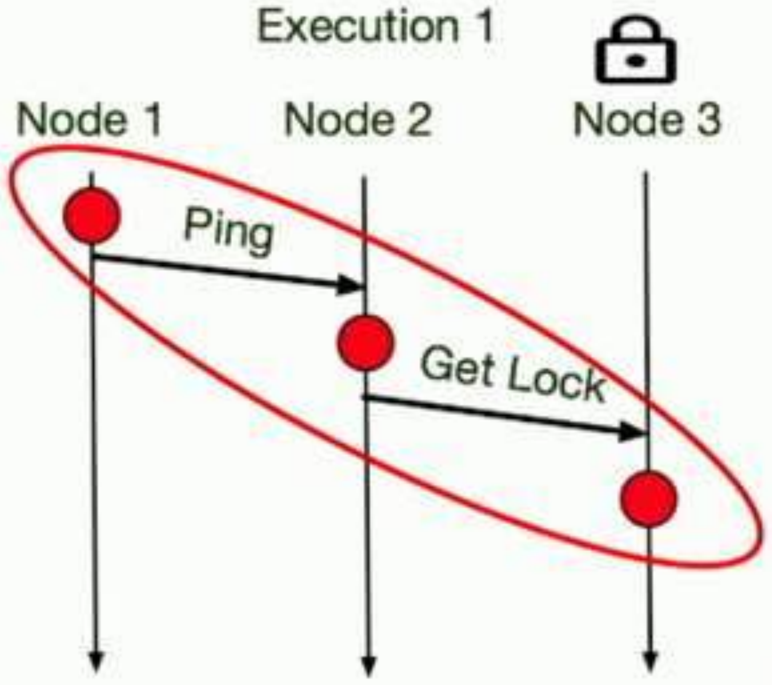
Reasoning About Global State

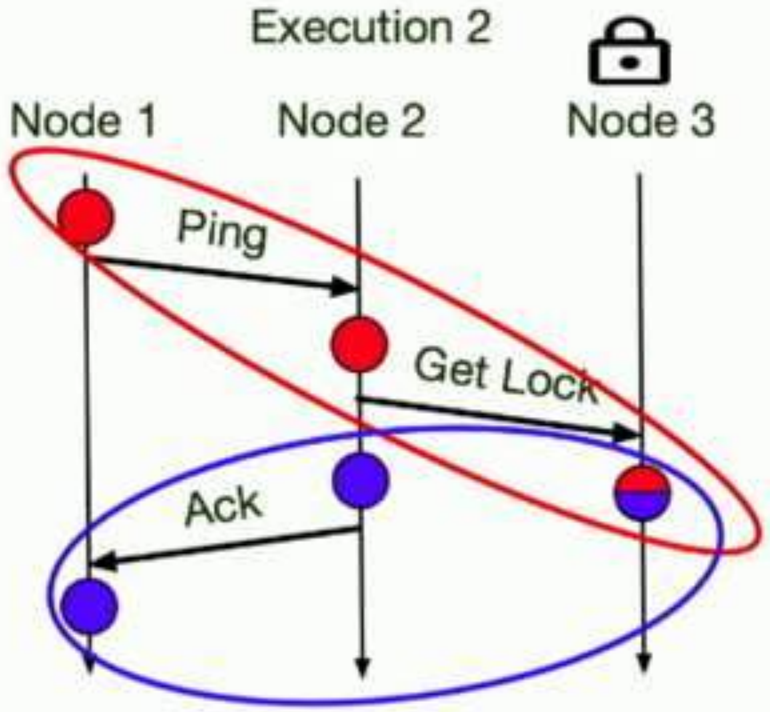
- Consistent Cuts
- Ground States
- State Bucketing

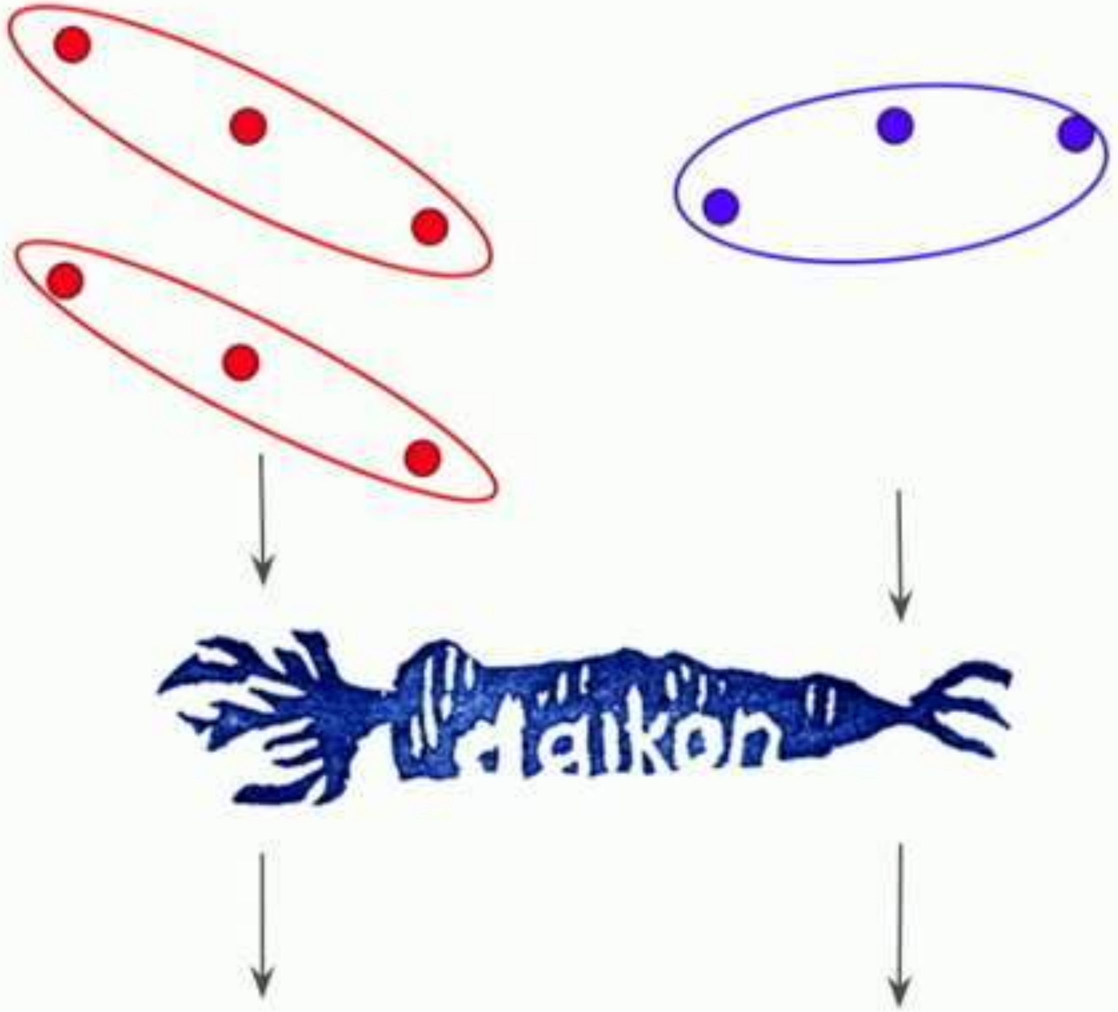


Reasoning About Global State

- Consistent Cuts
- Ground States
- State Bucketing



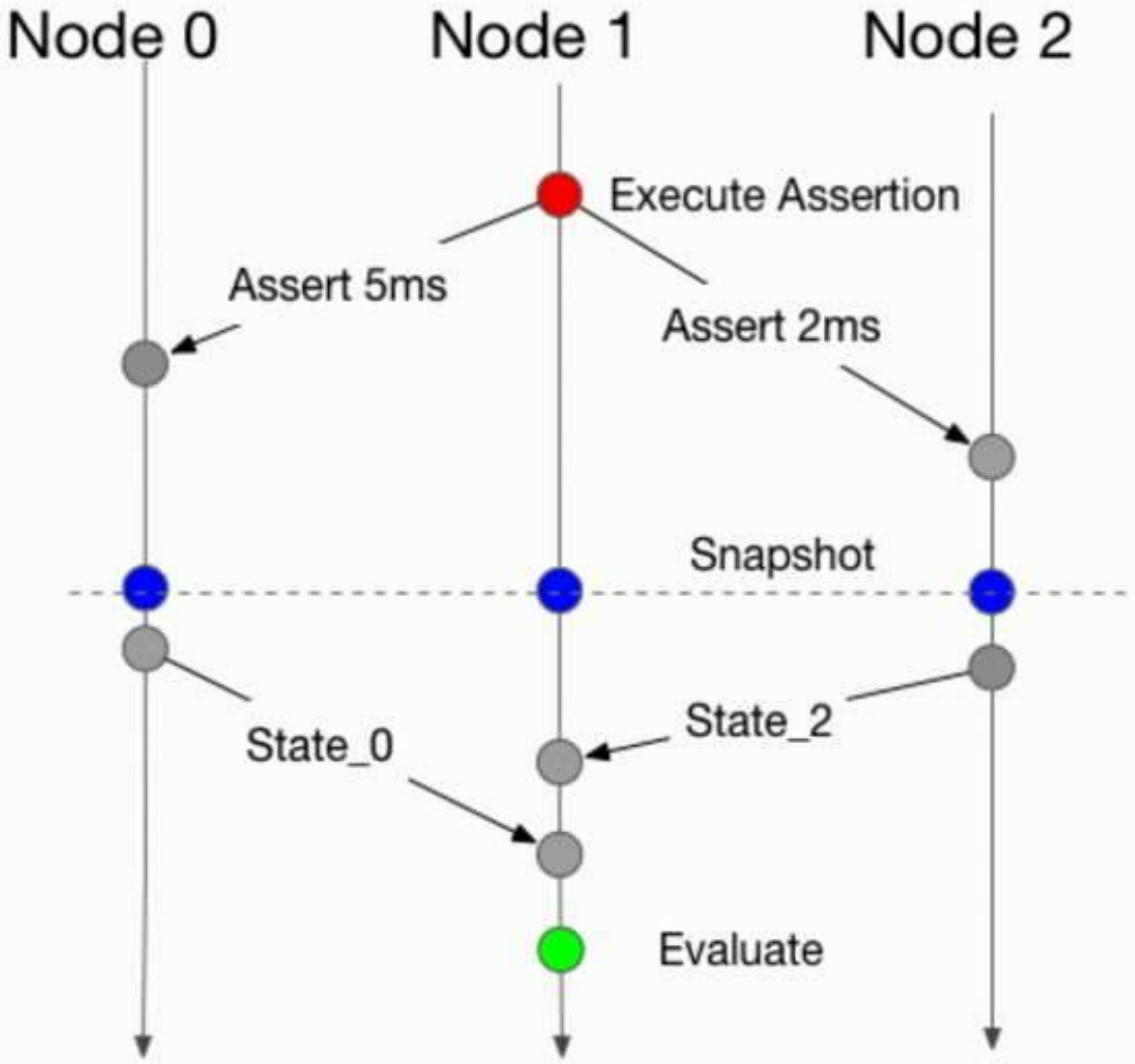




Node_**3**_InCritical == **True**
Node_**2**_InCritical != Node_**3**_InCritical
Node_**2**_InCritical == Node_**1**_InCritical

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

System and Targeted property	Dinv-inferred invariant	Description
Raft **Strong Leader principle**	**∀ follower *i*, len(leader log) ≥ len(*i*'s log)**	**All appended log entries must be propagated by the leader**
Raft Log matching	∀ nodes *i*, *j* if *i*-log[*c*] = *j*-log[*c*] → ∀(*x* ≤ *c*), *i*-log[*x*] = *j*-log[*x*]	If two logs contain an entry with the same index and term, then the logs are identical on all previous entries.
Raft Leader agreement	If ∃ node *i*, s.t *i* leader, than ∀ *j* ≠ *i*, *j* follower	If a leader exists, then all other nodes are followers.

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

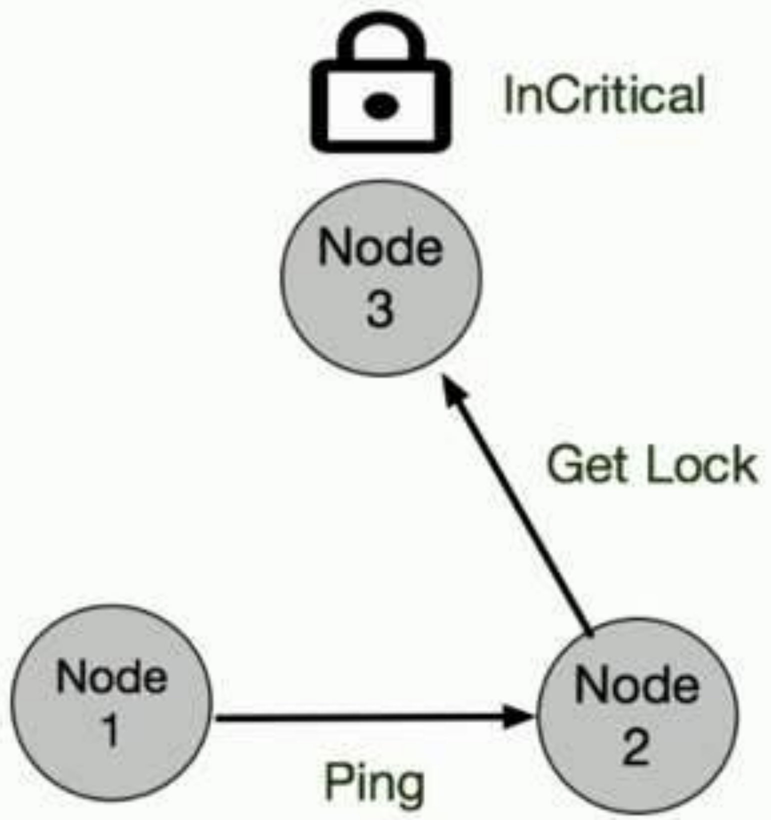
Future work

- 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



`∀nodes InCritical <= 1`

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

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