A Dynamic Evaluation of the Precision of Static Heap Abstractions

OOSPLA - Reno, NV October 20, 2010

Percy Liang Omer Tripp Mayur Naik Mooly Sagiv
UC Berkeley Tel-Aviv Univ. Intel Labs Berkeley Tel-Aviv Univ.
Introduction

Broad goal: verify correctness properties of software
Introduction

Broad goal: verify correctness properties of software

Motivating domain: multi-threaded programs (race and deadlock detection)
Introduction

Broad goal: verify correctness properties of software

Motivating domain: multi-threaded programs (race and deadlock detection)
Introduction

Broad goal: verify correctness properties of software

Motivating domain: multi-threaded programs (race and deadlock detection)

client query

program → Static Analysis ← heap abstraction

no OR possible

(false positives ⇒ imprecision!)
Introduction

Broad goal: verify correctness properties of software

Motivating domain: multi-threaded programs (race and deadlock detection)

client query

program $\rightarrow$ Static Analysis $\leftarrow$ heap abstraction

no OR possible

(false positives $\Rightarrow$ imprecision!)

Heap abstraction affects **precision** and **scalability**
Introduction

Broad goal: verify correctness properties of software

Motivating domain: multi-threaded programs (race and deadlock detection)

Heap abstraction affects precision and scalability

Question: what heap abstractions should one use?
Example client: ThreadEscape

Query: Does a variable point to a thread-escaping object at a program point?
Example client: **THREAD_ESCAPE**

**Query:** Does a variable point to a thread-escaping object at a program point?

```cpp
getnew() {
    return new
}
x = getnew()
y = getnew()
y.f = new
z = new
spawn y
p: ... ? ...
```
Example client: **ThreadEscape**

**Query:** Does a variable point to a thread-escaping object at a program point?

getnew() {
    return new
}

x = getnew()
y = getnew()
y.f = new
z = new
spawn y

p: ... ? ...
Example client: **THREAD_ESCAPE**

**Query:** Does a variable point to a thread-escaping object at a program point?

```java
getnew() {
    return new
}
x = getnew()
y = getnew()
y.f = new
z = new
spawn y
p: ... ? ...
x
y
3
```
Example client: **THREAD\text{ESCAPE}\text{E}**

**Query:** Does a variable point to a thread-escaping object at a program point?

```plaintext
getnew() {
    return new
}
x = getnew()
y = getnew()
y.f = new
z = new
spawn y
p: ... ? ...
```

![Diagram](image)
Example client: **THREAD_ESCAPE**

**Query:** Does a variable point to a thread-escaping object at a program point?

```plaintext
getnew() {
    return new
}
x = getnew()
y = getnew()
y.f = new
z = new
spawn y
p: ... ? ...
```

- ![Diagram of variable points](attachment:variable_points.png)
Example client: THREAD ESCAPE

Query: Does a variable point to a thread-escaping object at a program point?

getnew() {
    return new
}
x = getnew()
y = getnew()
y.f = new
z = new
spawn y
p: ... ? ...

Diagram:
- x
- y
- z
Example client: **ThreadEscape**

Query: Does a variable point to a thread-escaping object at a program point?

```java
getnew() {
    return new
}
x = getnew()
y = getnew()
y.f = new
z = new
spawn y
p: ... ? ...
x
y
z
```

**concrete answer**

<table>
<thead>
<tr>
<th></th>
<th>no</th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
</table>

x     y     z
Example client: ThreadEscape

Query: Does a variable point to a thread-escaping object at a program point?

```java
getnew() {
    return new
}
x = getnew()
y = getnew()
y.f = new
z = new
spawn y
p: ... ? ...
```

Concrete answer: no, yes, no
Abstract answer: yes, yes, no
Heap abstractions

Heap abstraction: partitioning of concrete objects
Heap abstractions

Heap abstraction: partitioning of concrete objects
Heap abstractions

**Heap abstraction:** partitioning of concrete objects

Property holds of partition $\iff \exists o \in$ partition such that property holds of $o$
Heap abstractions

**Heap abstraction**: partitioning of concrete objects

Property holds of partition ⇔ ∃o ∈ partition such that property holds of o

Formally: heap abstraction is function α

\[ \text{concrete object } o \rightarrow\text{ abstract object } \alpha(o) \]
**Heap abstractions**

**Heap abstraction**: partitioning of concrete objects

Property holds of partition $\iff \exists o \in \text{partition} \text{ such that property holds of } o$

Formally: heap abstraction is function $\alpha$

concrete object $o \quad \longrightarrow \quad $ abstract object $\alpha(o)$

Example:

$\alpha(o) = \text{alloc-site}(o)$
Heap abstractions

**Heap abstraction:** partitioning of concrete objects

Property holds of partition $\iff \exists o \in \text{partition such that property holds of } o$

Formally: heap abstraction is function $\alpha$

$\text{concrete object } o \longrightarrow \text{abstract object } \alpha(o)$

Example:

$\alpha(o) = \langle \text{alloc-site}(o), \text{other-information}(o) \rangle$
The heap abstraction landscape

Tradeoff:

imprecise, fast
(e.g., 0-CFA)

precise, slow
(e.g., ∞-CFA)
The heap abstraction landscape

Tradeoff:
imprecise, fast
(e.g., 0-CFA)

precise, slow
(e.g., ∞-CFA)

How much precision is necessary for the given client?
The heap abstraction landscape

Tradeoff:

imprecise, fast  
(e.g., 0-CFA)  

precise, slow  
(e.g., ∞-CFA)

How much precision is necessary for the given client?
But it’s expensive to implement precise abstractions...
The heap abstraction landscape

Tradeoff:

imprecise, fast
(e.g., 0-CFA) ← precise, slow
(e.g., ∞-CFA)

How much precision is necessary for the given client?
But it’s expensive to implement precise abstractions...

Many dimensions:

$k$-CFA: call stack information
The heap abstraction landscape

Tradeoff:

imprecise, fast
(e.g., 0-CFA)  

precise, slow
(e.g., ∞-CFA)

How much precision is necessary for the given client?
But it’s expensive to implement precise abstractions...

Many dimensions:

$k$-CFA: call stack information
Object recency
Heap connectivity
etc.
The heap abstraction landscape

Tradeoff:

imprecise, fast
(e.g., 0-CFA) -- precise, slow
(e.g., ∞-CFA)

How much precision is necessary for the given client?
But it’s expensive to implement precise abstractions...

Many dimensions:

$k$-CFA: call stack information
Object recency
Heap connectivity
etc.

Question: how can we explore all these abstractions cheaply?
Main idea

**Goal:** get an idea of the utility of these abstractions without implementing expensive static analyses
Main idea

Goal: get an idea of the utility of these abstractions without implementing expensive static analyses

Key idea: use dynamic information
Main idea

Goal: get an idea of the utility of these abstractions without implementing expensive static analyses

Key idea: use dynamic information

Static: all traces (expensive)
Main idea

**Goal:** get an idea of the utility of these abstractions without implementing expensive static analyses

**Key idea:** use dynamic information

Static: all traces (expensive)  Dynamic: one trace (cheap)
Methodology

1. Run program dynamically with instrumentation

Concrete trace: $\omega_1 \omega_2 \omega_3 \omega_4 \omega_5$
Methodology

1. Run program dynamically with instrumentation
2. Compute heap abstraction on each state

Concrete trace: \( \omega_1 \omega_2 \omega_3 \omega_4 \omega_5 \)

Abstract trace: \( \omega_1^\alpha \omega_2^\alpha \omega_3^\alpha \omega_4^\alpha \omega_5^\alpha \)
Methodology

1. Run program dynamically with instrumentation
2. Compute heap abstraction on each state
3. Answer query under abstraction

Concrete trace: \( \omega_1 \omega_2 \omega_3 \omega_4 \omega_5 \)

Abstract trace: \( \omega_1^\alpha \omega_2^\alpha \omega_3^\alpha \omega_4^\alpha \omega_5^\alpha \)

Abstract query answer: no yes no yes no
Methodology

1. Run program dynamically with instrumentation
2. Compute heap abstraction on each state
3. Answer query under abstraction

Query is true ⇔ true on any state in trace

Concrete trace: \( \omega_1 \omega_2 \omega_3 \omega_4 \omega_5 \)
Abstract trace: \( \omega_1^\alpha \omega_2^\alpha \omega_3^\alpha \omega_4^\alpha \omega_5^\alpha \)
Abstract query answer: no yes no yes no \( \Rightarrow \) yes
What does this tell us?
What does this tell us?

Note: no approximation on primitive data, method summarization, etc. (focus exclusively on the heap abstraction)
What does this tell us?

\textbf{Note:} no approximation on primitive data, method summarization, etc. (focus exclusively on the heap abstraction)

⇒ performing the most precise analysis using a given heap abstraction $\alpha$
What does this tell us?

Note: no approximation on primitive data, method summarization, etc. (focus exclusively on the heap abstraction)

⇒ performing the most precise analysis using a given heap abstraction $\alpha$

⇒ provides upper bound on precision of any static analysis using $\alpha$
Outline

- **Abstractions**: augment allocation sites with more context
  - call stack
  - object recency
  - heap connectivity
Outline

- **Abstractions**: augment allocation sites with more context
  - call stack
  - object recency
  - heap connectivity

- **Clients**: motivated by concurrency
  - **ThreadEscape**
  - **SharedAccess**
  - **SharedLock**
  - **NonStationaryField**
Outline

• **Abstractions**: augment allocation sites with more context
  – call stack
  – object recency
  – heap connectivity

• **Clients**: motivated by concurrency
  – `ThreadEscape`
  – `SharedAccess`
  – `SharedLock`
  – `NonStationaryField`

• **Benchmarks**: 9 programs from the standard Dacapo suite
Outline

• **Abstractions**: augment allocation sites with more context
  – call stack
  – object recency
  – heap connectivity

• **Clients**: motivated by concurrency
  – `ThreadEscape`
  – `SharedAccess`
  – `SharedLock`
  – `NonStationaryField`

• **Benchmarks**: 9 programs from the standard Dacapo suite

• **Results**: investigate all combinations
Abstraction: call stack [Shivers, 1988]

Common pattern: factory constructor methods

getnew() {
    h1: return new
}
p2: x = getnew()
p3: y = getnew()
spawn y
p1: ... x ...

Abstraction: call stack [Shivers, 1988]

Common pattern: factory constructor methods

```plaintext
getnew() {
    h1: return new
}
p2: x = getnew()
p3: y = getnew()
spawn y
p1: ... x ...
```

✗ Allocation sites are too weak
Abstraction: call stack [Shivers, 1988]

Abstraction $\text{ALLOC}_k$ ($k$ is call stack depth):

$$\text{call-stack-during-allocation-of}(o)[1..k]$$

Common pattern: factory constructor methods

```c
getnew() {
    h1: return new
}
p2: x = getnew()
p3: y = getnew()
spawn y
p1: ... x ...
```

× Allocation sites are too weak
Abstraction: call stack [Shivers, 1988]

Abstraction $\text{ALLOC}_k$ ($k$ is call stack depth):

$$\text{call-stack-during-allocation-of}(o)[1..k]$$

Common pattern: factory constructor methods

```java
getnew() {
    h1: return new
}

p2: x = getnew()
p3: y = getnew()
spawn y

p1: ... x ...
```

Allocation sites are too weak

Adding one level of calling context is sufficient
Abstraction: object recency [Balakrishnan & Reps, 2006]

Common pattern: server programs construct data, release to new thread

```plaintext
while (*) {
    x = new
    p1: ... x ...
    spawn x
}
```
Abstraction: object recency [Balakrishnan & Reps, 2006]

Common pattern: server programs construct data, release to new thread

while (*) {
    x = new
    p1: ... x ...
    spawn x
}

\( x \) \text{ h1 Alloc} = \infty

\text{x} \quad \text{No amount of calling context helps}

\text{ALLOC}_{k=\infty}
Abstraction: object recency [Balakrishnan & Reps, 2006]

Abstraction $\text{RECENCY}_k^r$ ($r$ is recency depth); for $r = 1$:

$$\text{recency-bit}(o)$$

Common pattern: server programs construct data, release to new thread

```java
while (*) {
    x = new
    p1: ... x ...
    spawn x
}
```

$\text{ALLOC}_{k=\infty}$

No amount of calling context helps
Abstraction: object recency [Balakrishnan & Reps, 2006]

Abstraction $\text{RECENTY}_k^r$ ($r$ is recency depth); for $r = 1$:

\[
\text{recency-bit}(o)
\]

Objects allocated: $o_1 \quad o_2 \quad o_3 \quad o_4 \quad o_5$

$\text{ALLOC}_k: \quad h_2 \quad h_4 \quad h_4 \quad h_2 \quad h_4$

Common pattern: server programs construct data, release to new thread

\[
\text{while (*)} \{
    x = \text{new}
    p_1: \ldots x \ldots
    \text{spawn} \ x
\}
\]

No amount of calling context helps

\[
\times
\]
Abstraction: object recency [Balakrishnan & Reps, 2006]

Abstraction $\text{Recency}_r^k$ ($r$ is recency depth); for $r = 1$:

$$\text{recency-bit}(o)$$

Objects allocated: $o_1 \quad o_2 \quad o_3 \quad o_4 \quad o_5$

$\text{ALLOC}_k$: $h_2 \quad h_4 \quad h_4 \quad h_2 \quad h_4$

recency-bit: $0 \quad 0 \quad 0 \quad 1 \quad 1$

Common pattern: server programs construct data, release to new thread

```java
while (*) {
    x = new
    p1: ... x ...
    spawn x
}
```

$\text{ALLOC}_{k=\infty}$

✗ No amount of calling context helps
Abstraction: object recency [Balakrishnan & Reps, 2006]

Abstraction $\text{Recency}_k^r$ ($r$ is recency depth); for $r = 1$:

\[
\text{recency-bit}(o)
\]

- Objects allocated: $o_1$ $o_2$ $o_3$ $o_4$ $o_5$
- $\text{ALLOC}_k$: $h_2$ $h_4$ $h_4$ $h_2$ $h_4$
- recency-bit: $0$ $0$ $0$ $1$ $1$

Common pattern: server programs construct data, release to new thread

```java
while (*) {
    x = new p1: ... x ...
    spawn x
}
```

$\text{ALLOC}_{k=\infty}$

$\text{Recency}_{r=1}$

- $\times$ No amount of calling context helps
- $\checkmark$ Recency makes the proper distinctions
Abstraction: heap connectivity [Sagiv et al., 2002]

Common pattern: build linked list data structures

h1: s = new
    spawn s
h2: x = new
    y = x
    while (*) {
    h3: z = new
        y.f = z
        if (x.f == y)
            s.f = z
            y = z
        }
    x = x.f
p1: ... x ...
Abstraction: heap connectivity [Sagiv et al., 2002]

**Common pattern**: build linked list data structures

```plaintext
h1: s = new
    spawn s
h2: x = new
    y = x
    while (*) {
        h3: z = new
            y.f = z
            if (x.f == y)
                s.f = z
                y = z
        x = x.f
    } 
p1: ... x ...
```

⚠️ No amount of recency helps

\[ \text{Recency } r = \infty \]
Abstraction: heap connectivity [Sagiv et al., 2002]

ReachFrom\(k\): set of alloc. sites reaching \(\text{ALLOC}_k(o)\)

Common pattern: build linked list data structures

\(h1\): \(s = \text{new spawn } s\)
\(h2\): \(x = \text{new } y = x\)
while (*) {
\(h3\): \(z = \text{new } y.f = z\)
if \((x.f == y)\)
\(s.f = z\)
\(y = z\)
\}
\(x = x.f\)

\(p1\): \(\ldots x \ldots\)

\(\times\) No amount of recency helps
Abstraction: heap connectivity [Sagiv et al., 2002]

**ReachFrom**$_k$: set of alloc. sites reaching $\text{ALLOC}_k(o)$

**PointedToBy**$_k$: set of alloc. sites reaching $\text{ALLOC}_k(o)$ in 1 step

**Common pattern**: build linked list data structures

```
h1: s = new
    spawn s
h2: x = new
    y = x
    while (*) {
        h3: z = new
            y.f = z
            if (x.f == y)
                s.f = z
                y = z
        }
    x = x.f
p1: ... x ...
```

- **Recency** $r = \infty$

$\times$ No amount of recency helps
Abstraction: heap connectivity [Sagiv et al., 2002]

**REACHFROM**$_k$: set of alloc. sites reaching $ALLOC_k(o)$

**POINTEDTOBY**$_k$: set of alloc. sites reaching $ALLOC_k(o)$ in 1 step

Common pattern: build linked list data structures

```plaintext
h1: s = new
    spawn s
h2: x = new
    y = x
    while (*) {
    h3: z = new
        y.f = z
        if (x.f == y)
            s.f = z
            y = z
    }
    x = x.f
p1: ... x ...
```

Recency $r = \infty$

Reachability makes proper distinctions

× No amount of recency helps

✓ Reachability makes proper distinctions
Clients

**ThreadEscape**: Does variable $v$ point to an object potentially reachable from another thread?
Clients

**ThreadEscape**: Does variable $v$ point to an object potentially reachable from another thread?

**SharedAccess**: Does variable $v$ point to an object actually accessed by multiple threads?
Clients

**ThreadEscape**: Does variable $v$ point to an object potentially reachable from another thread?

**SharedAccess**: Does variable $v$ point to an object actually accessed by multiple threads?

**SharedLock**: Does variable $v$ point to an object which is locked by multiple threads?
Clients

**ThreadEscape**: Does variable $v$ point to an object potentially reachable from another thread?

**SharedAccess**: Does variable $v$ point to an object actually accessed by multiple threads?

**SharedLock**: Does variable $v$ point to an object which is locked by multiple threads?

**NonStationaryField**: for a field $f$, does there exist an object $o$ such that $o.f$ is written to after $o.f$ is read from? (generalization of $\text{final}$ in Java from [Unkel & Lam, 2008])
Clients

**ThreadEscape**: Does variable $v$
point to an object potentially reachable from another thread?

**SharedAccess**: Does variable $v$
point to an object actually accessed by multiple threads?

**SharedLock**: Does variable $v$
point to an object which is locked by multiple threads?

**NonStationaryField**: for a field $f$, does there exist an object $o$ such that
$o.f$ is written to after $o.f$ is read from?
(generalization of `final` in Java from [Unkel & Lam, 2008])

Motivated by race and deadlock detection.
Benchmarks

9 Java programs from the DaCapo benchmark suite (version 9.12):

- **antlr**: A parser generator and translator generator
- **avrora**: A simulation and analysis framework for AVR microcontrollers
- **batik**: A Scalable Vector Graphics (SVG) toolkit
- **fop**: An output-independent print formatter
- **hsqldb**: An SQL relational-database engine
- **luindex**: A text indexing tool
- **lusearch**: A text search tool
- **pmd**: A source-code analyzer
- **xalan**: An XSLT processor for transforming XML

290–1357 classes, 1.7K–6.8K methods, 133K–512K bytecodes, 5–46 threads
Experiments

Precision:

\[ 0\% \leq \frac{\text{number of queries } q \text{ such that } q \text{ is true (concrete)}}{\text{number of queries } q \text{ such that } q^\alpha \text{ is true (abstract)}} \leq 100\% \]
Experiments

Precision:

\[ 0\% \leq \frac{\text{number of queries } q \text{ such that } q \text{ is true (concrete)}}{\text{number of queries } q \text{ such that } q^\alpha \text{ is true (abstract)}} \leq 100\% \]

Questions:

- What abstraction works best for a given client?
- What is the effect of the \( k \) in \( k \)-CFA?
- What is the effect of the recency depth \( r \)?
- How scalable are the high-precision abstractions?
### General results: ThreadEscape

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Alloc</th>
<th>$\text{ALLOC}_{k=5}$</th>
<th>Recency</th>
<th>ReachFrom</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>48.6</td>
<td>85.0</td>
<td>81.0</td>
<td>100.0</td>
</tr>
<tr>
<td>avrora</td>
<td>54.7</td>
<td>62.3</td>
<td>69.2</td>
<td>77.8</td>
</tr>
<tr>
<td>batik</td>
<td>13.5</td>
<td>15.1</td>
<td>20.9</td>
<td>20.6</td>
</tr>
<tr>
<td>fop</td>
<td>36.3</td>
<td>99.3</td>
<td>42.8</td>
<td>41.3</td>
</tr>
<tr>
<td>hsqldb</td>
<td>62.6</td>
<td>69.0</td>
<td>94.3</td>
<td>?</td>
</tr>
<tr>
<td>luindex</td>
<td>6.3</td>
<td>97.2</td>
<td>6.8</td>
<td>6.8</td>
</tr>
<tr>
<td>lusearch</td>
<td>14.3</td>
<td>90.0</td>
<td>19.0</td>
<td>19.6</td>
</tr>
<tr>
<td>pmd</td>
<td>12.4</td>
<td>87.1</td>
<td>14.9</td>
<td>14.6</td>
</tr>
<tr>
<td>xalan</td>
<td>64.0</td>
<td>78.9</td>
<td>78.7</td>
<td>76.6</td>
</tr>
<tr>
<td>average</td>
<td>34.8</td>
<td>76.0</td>
<td>47.5</td>
<td>44.7</td>
</tr>
</tbody>
</table>

### Main points:

- $\text{ALLOC}$ can be very imprecise
- $\text{ALLOC}_{k=5}$ works best most of the time
### General results: **NonStationaryField**

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Alloc</th>
<th>Alloc(_k=5)</th>
<th>Recency</th>
<th>ReachFrom</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>59.1</td>
<td>60.1</td>
<td>91.0</td>
<td>78.3</td>
</tr>
<tr>
<td>avrora</td>
<td>33.2</td>
<td>33.6</td>
<td>93.6</td>
<td>77.2</td>
</tr>
<tr>
<td>batik</td>
<td>35.8</td>
<td>36.1</td>
<td>99.5</td>
<td>65.3</td>
</tr>
<tr>
<td>fop</td>
<td>42.0</td>
<td>44.9</td>
<td>90.9</td>
<td>68.2</td>
</tr>
<tr>
<td>hsqldb</td>
<td>45.4</td>
<td>49.5</td>
<td>94.6</td>
<td>?</td>
</tr>
<tr>
<td>luindex</td>
<td>78.0</td>
<td>84.2</td>
<td>94.8</td>
<td>94.8</td>
</tr>
<tr>
<td>lusearch</td>
<td>38.2</td>
<td>38.2</td>
<td>64.9</td>
<td>56.5</td>
</tr>
<tr>
<td>pmd</td>
<td>37.8</td>
<td>39.9</td>
<td>96.4</td>
<td>69.4</td>
</tr>
<tr>
<td>xalan</td>
<td>44.0</td>
<td>44.5</td>
<td>90.4</td>
<td>74.2</td>
</tr>
<tr>
<td>average</td>
<td>45.9</td>
<td>47.9</td>
<td>90.7</td>
<td>73.0</td>
</tr>
</tbody>
</table>

**Main points:**

- Call stack useless, reachability helps a bit
- **Recency** offers huge improvement: captures temporal properties
Main points:

- Phase transition: sharp increase in precision beyond $k \approx 5$
- Synergy of information: $\text{REACHFROM}$ requires high $k$ to be precise
Effect of recency depth

**ThreadEscape on batik:**

<table>
<thead>
<tr>
<th></th>
<th>$r = 0$</th>
<th>$r = 1$</th>
<th>$r = 2$</th>
<th>$r = 3$</th>
<th>$r = 4$</th>
<th>$r = 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 0$</td>
<td>13.5</td>
<td>20.9</td>
<td>21.4</td>
<td>22.1</td>
<td>22.5</td>
<td>22.6</td>
</tr>
<tr>
<td>$k = \infty$</td>
<td>15.1</td>
<td>23.4</td>
<td>99.0</td>
<td>99.0</td>
<td>99.0</td>
<td>99.0</td>
</tr>
</tbody>
</table>

**Main points:**
- Increasing recency depth beyond 1 helps, but maxes out quickly
- Synergy of information: need both large $k$ and large $r$ for success
Main points:

- Reachability is quite expensive, **Recency** is cheap
- **Random** is surprisingly effective on **NonStationaryField**, but **Recency** is better
Summary

• Goal: determine good heap abstractions to use in static analysis

• Dynamic analysis enables us to quickly explore many heap abstractions
Summary

• Goal: determine good heap abstractions to use in static analysis

• Dynamic analysis enables us to quickly explore many heap abstractions

• Heap abstraction has large impact on precision
  – Best abstraction depends on how its properties fit the client
  – Non-trivial interactions between dimensions
Summary

• Goal: determine good heap abstractions to use in static analysis

• Dynamic analysis enables us to quickly explore many heap abstractions

• Heap abstraction has large impact on precision
  – Best abstraction depends on how its properties fit the client
  – Non-trivial interactions between dimensions

• Hopefully will serve as a useful guide for developers of static analyses
Summary

- Goal: determine good heap abstractions to use in static analysis

- Dynamic analysis enables us to quickly explore many heap abstractions

- Heap abstraction has large impact on precision
  - Best abstraction depends on how its properties fit the client
  - Non-trivial interactions between dimensions

- Hopefully will serve as a useful guide for developers of static analyses

Thank you!