A Dynamic Evaluation of the Precision of Static Heap Abstractions

OOSPLA - Reno, NV October 20, 2010

Percy Liang
UC Berkeley

Omer Tripp
Tel-Aviv Univ.

Mayur Naik
Intel Labs Berkeley

Mooly Sagiv
Tel-Aviv Univ.

Introduction

Broad goal: verify correctness properties of software

Introduction

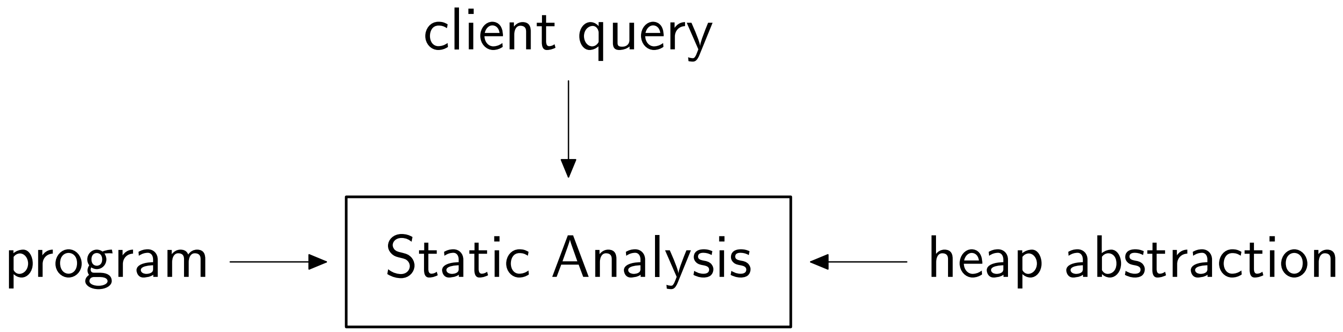
Broad goal: verify correctness properties of software

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)



client query

program

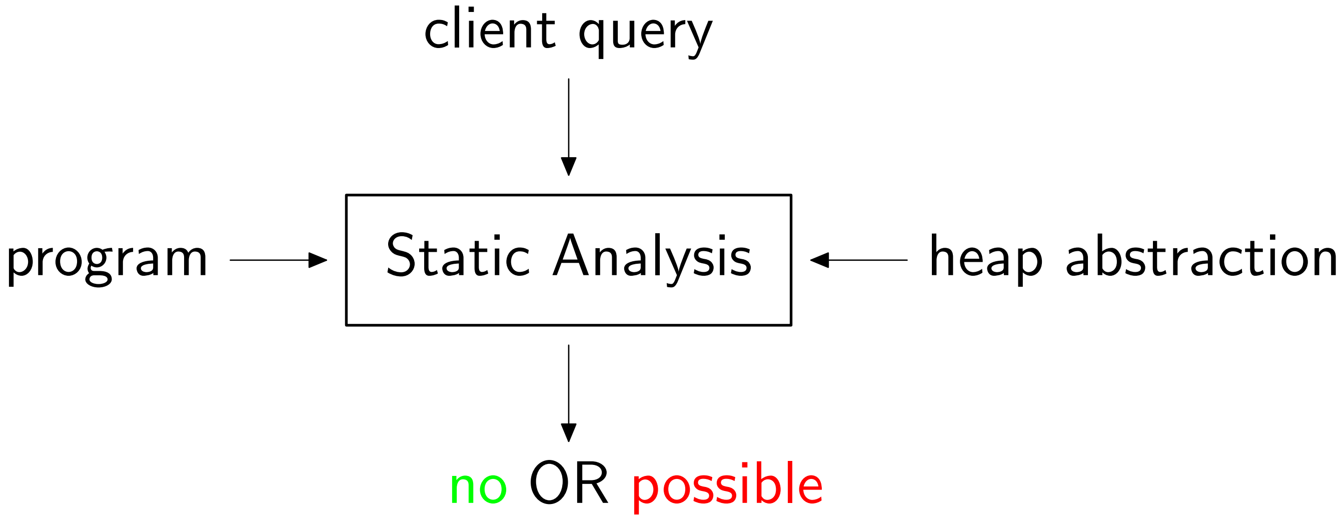
Static Analysis

heap abstraction

Introduction

Broad goal: verify correctness properties of software

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



client query

program

Static Analysis

heap abstraction

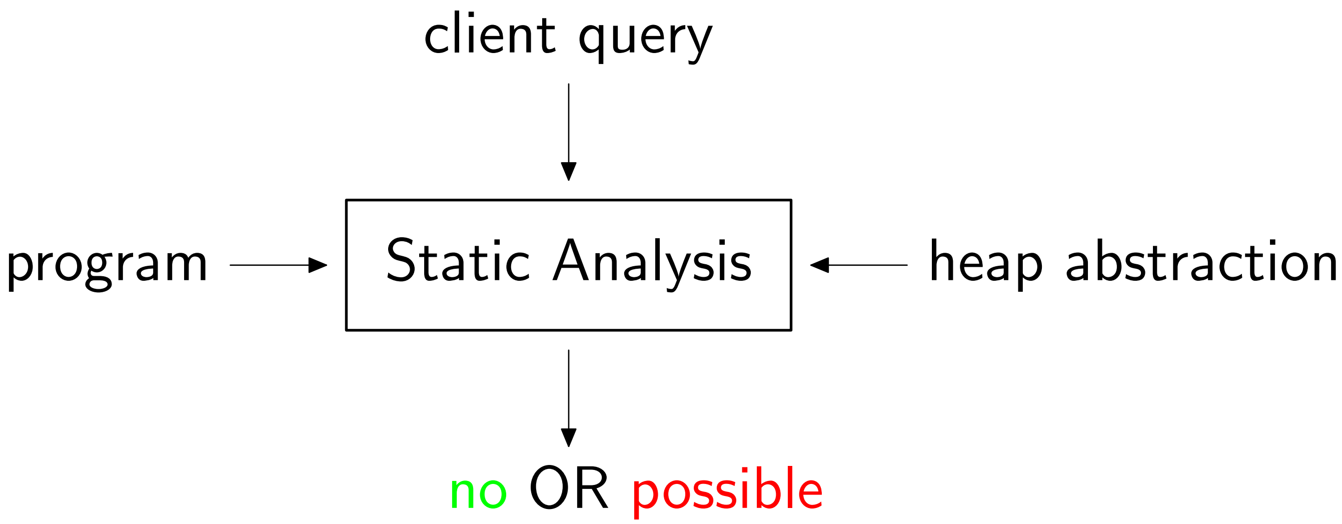
no OR possible

(false positives ⇒ imprecision!)

Introduction

Broad goal: verify correctness properties of software

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



client query

program → Static Analysis ← heap abstraction

no OR possible

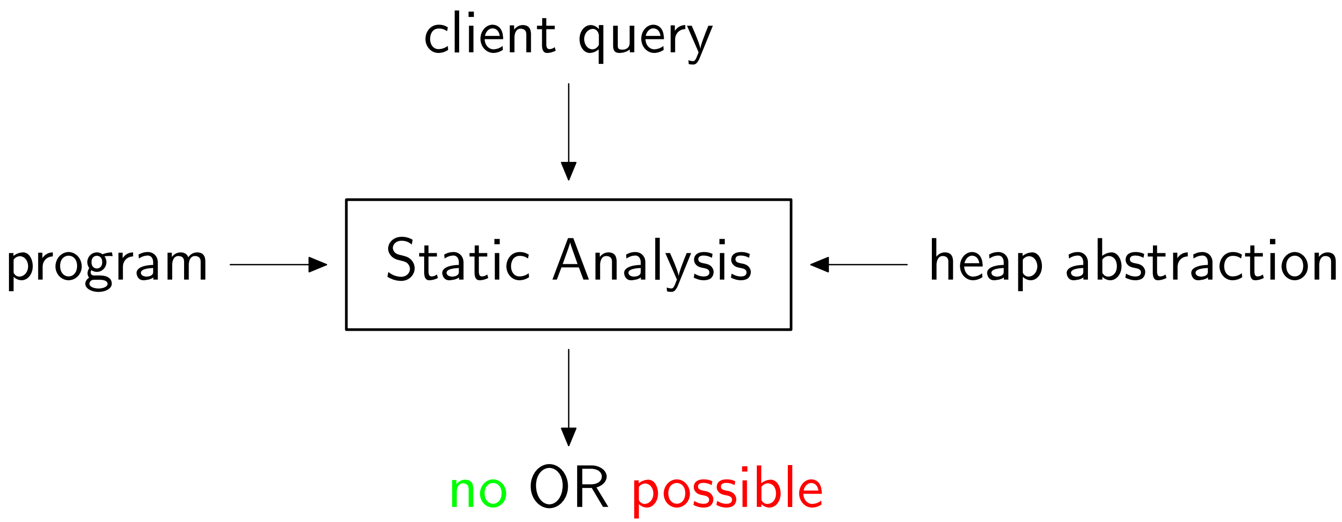
(false positives ⇒ imprecision!)

Heap abstraction affects **precision** and **scalabilty**

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!)

Heap abstraction affects **precision** and **scalabilty**

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: ThreadEscape

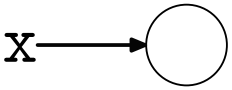
Query: Does a variable point to a thread-escaping object at a program point?

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

Example client: THREADESCAPE

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

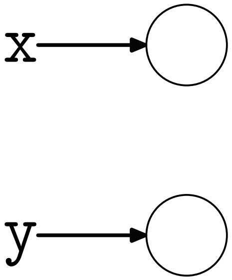
```
    getnew() {
      return new
    }
    x = getnew()
    y = getnew()
    y.f = new
    z = new
    spawn y
 p: ... ? ...
```



Example client: THREADESCAPE

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

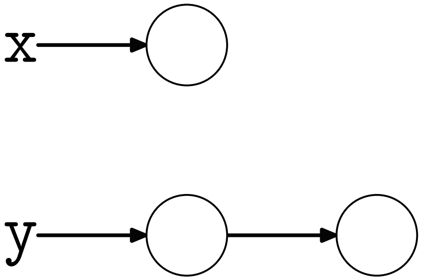
```
        getnew() {
          return new
        }
        x = getnew()
        y = getnew()
        y.f = new
        z = new
        spawn y
     p: ... ? ...
```



Example client: THREADESCAPE

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

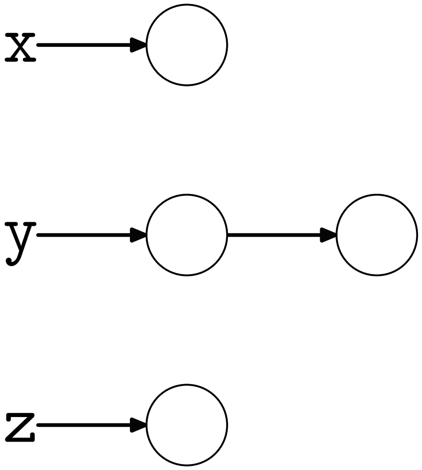
```
    getnew() {
      return new
    }
    x = getnew()
    y = getnew()
    y.f = new
    z = new
    spawn y
p:  ... ? ...
```



Example client: THREADESCAPE

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

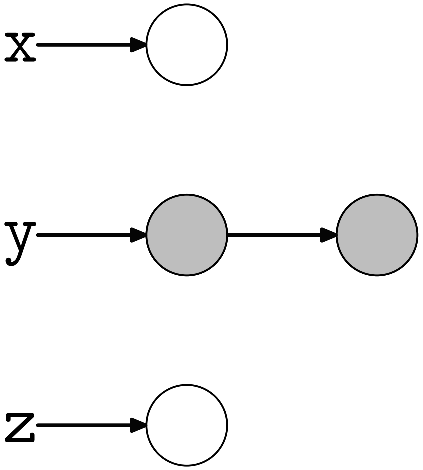
```
    getnew() {
      return new
    }
    x = getnew()
    y = getnew()
    y.f = new
    z = new
    spawn y
 p: ... ? ...
```



Example client: THREADESCAPE

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

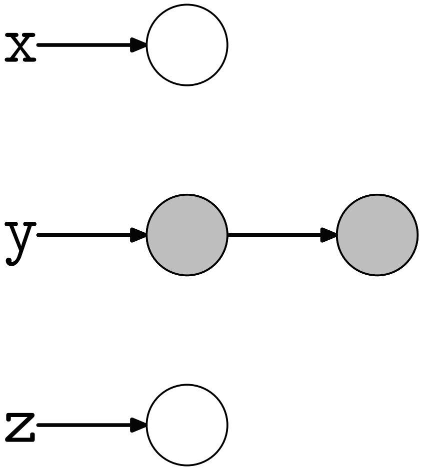
```
    getnew() {
      return new
    }
    x = getnew()
    y = getnew()
    y.f = new
    z = new
    spawn y
p: ... ? ...
```



Example client: 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:  ... ? ...
```

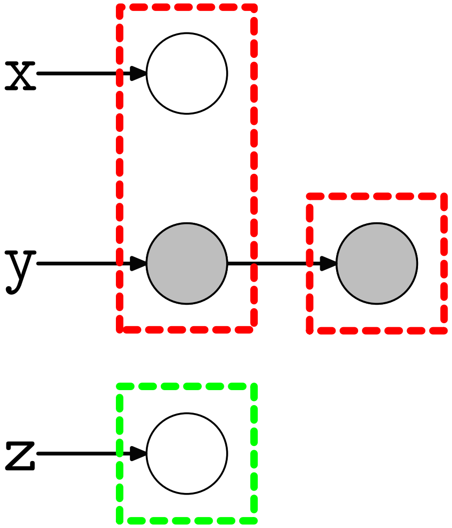


	x	y	z
concrete answer	no	yes	no

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:  ... ? ...
```



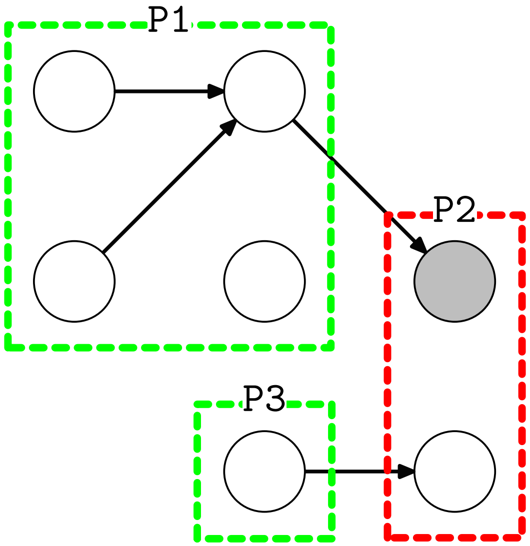
	x	y	z
concrete answer	no	yes	no
abstract answer	yes	yes	no

Heap abstractions

Heap abstraction: partitioning of concrete objects

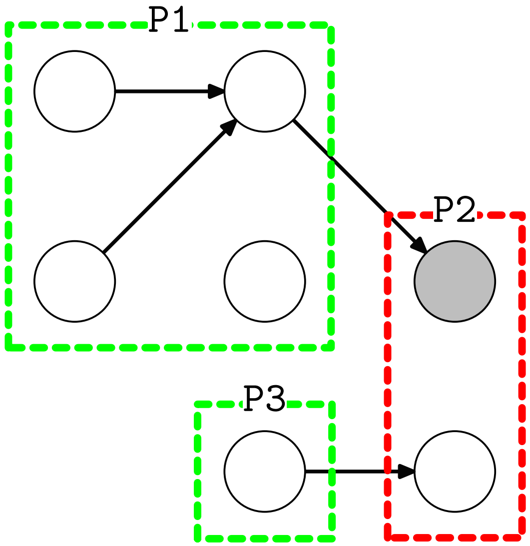
Heap abstractions

Heap abstraction: partitioning of concrete objects



Heap abstractions

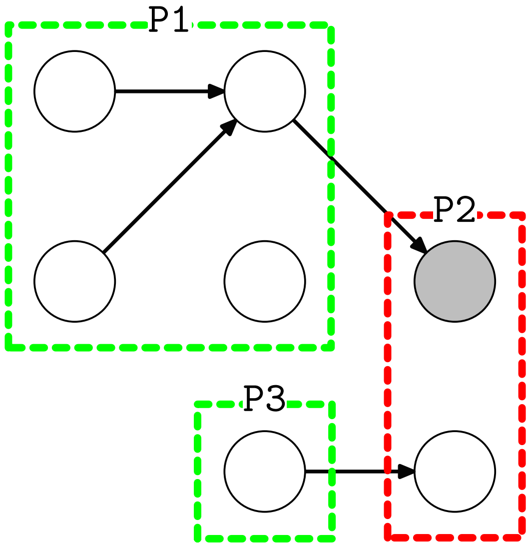
Heap abstraction: partitioning of concrete objects



Property holds of partition $\Leftrightarrow \exists o \in$ partition such that property holds of o

Heap abstractions

Heap abstraction: partitioning of concrete objects



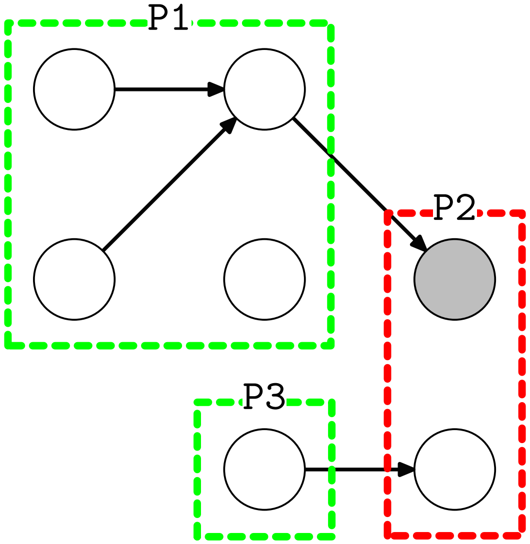
Property holds of partition $\Leftrightarrow \exists o \in$ partition such that property holds of o

Formally: heap abstraction is function α

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

Heap abstractions

Heap abstraction: partitioning of concrete objects



Property holds of partition $\Leftrightarrow \exists o \in$ partition such that property holds of o

Formally: heap abstraction is function α

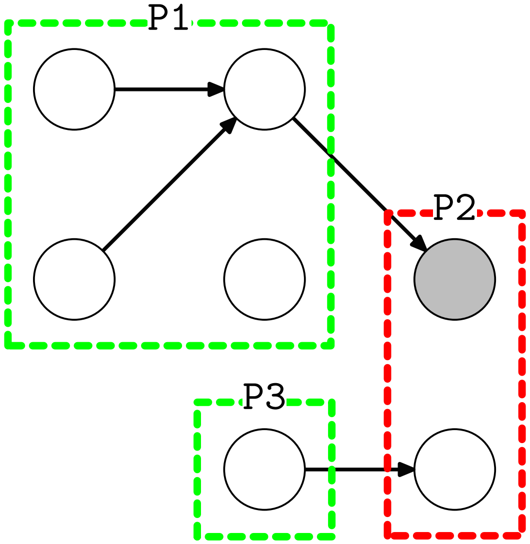
$$\text{concrete object } o \longrightarrow \text{abstract object } \alpha(o)$$

Example:

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

Heap abstractions

Heap abstraction: partitioning of concrete objects



Property holds of partition $\Leftrightarrow \exists o \in$ partition such that property holds of o

Formally: heap abstraction is function α

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

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

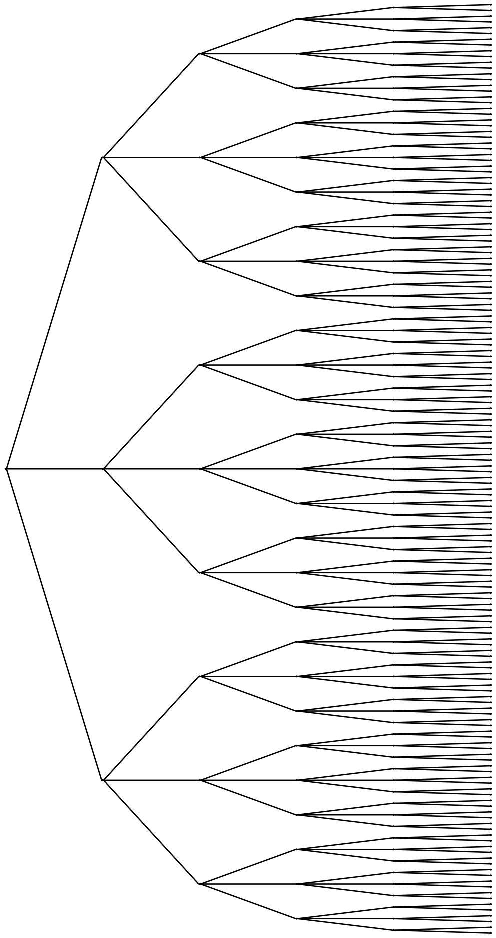
Goal: get an idea of the utility of these abstractions
without implementing expensive static analyses

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

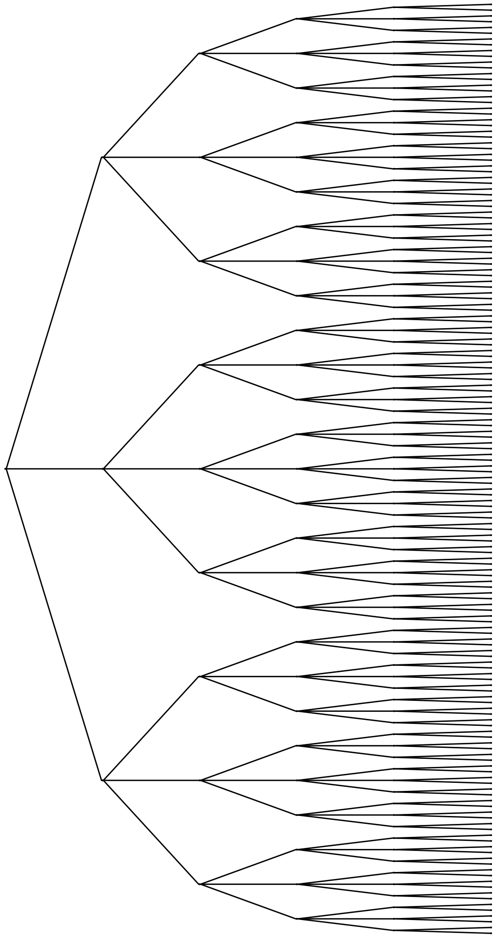


Static: all traces (expensive)

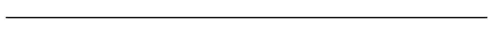
Main idea

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: ω_1 ω_2 ω_3 ω_4 ω_5

Methodology

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

Concrete trace:	ω_1	ω_2	ω_3	ω_4	ω_5
Abstract trace:	ω_1^α	ω_2^α	ω_3^α	ω_4^α	ω_5^α

Methodology

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

Concrete trace:	ω_1	ω_2	ω_3	ω_4	ω_5
Abstract trace:	ω_1^α	ω_2^α	ω_3^α	ω_4^α	ω_5^α
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:	ω_1	ω_2	ω_3	ω_4	ω_5	
Abstract trace:	ω_1^α	ω_2^α	ω_3^α	ω_4^α	ω_5^α	
Abstract query answer:	no	yes	no	yes	no	⇒ 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?

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

$\Rightarrow$ performing the most precise analysis using a given heap abstraction α

What does this tell us?

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

$\Rightarrow$ performing the most precise analysis using a given heap abstraction α

$\Rightarrow$ provides **upper bound** on precision of any static analysis using α

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]

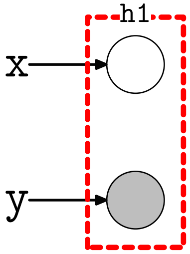
Common pattern: factory constructor methods

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

Abstraction: call stack [Shivers, 1988]

Common pattern: factory constructor methods

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



ALLOC

✗ Allocation sites are too weak

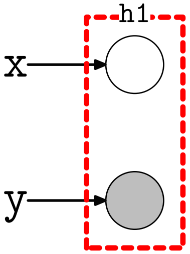
Abstraction: call stack [Shivers, 1988]

Abstraction $\textsc{Alloc}_k$ (k is call stack depth):

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

Common pattern: factory constructor methods

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



$\textsc{Alloc}$

✗ Allocation sites are too weak

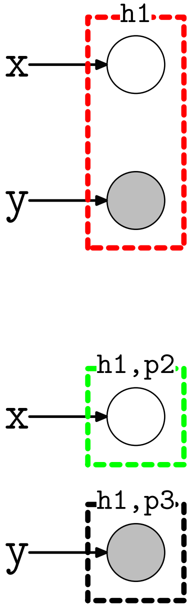
Abstraction: call stack [Shivers, 1988]

Abstraction $\textsc{Alloc}_k$ (k is call stack depth):

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

Common pattern: factory constructor methods

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



$\textsc{Alloc}$

$\textsc{Alloc}_{k=1}$

- ✗ Allocation sites are too weak
- ✓ Adding one level of calling context is sufficient

Abstraction: object recency [Balakrishnan & Reps, 2006]

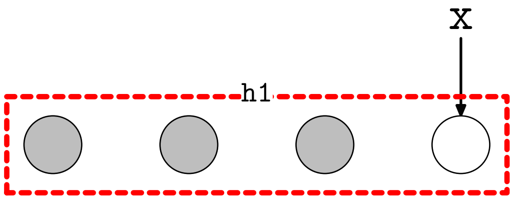
Common pattern: server programs construct data, release to new thread

```
    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
    }
```



$\text{A}\textsc{lloc}_{k=\infty}$

✗ No amount of calling context helps

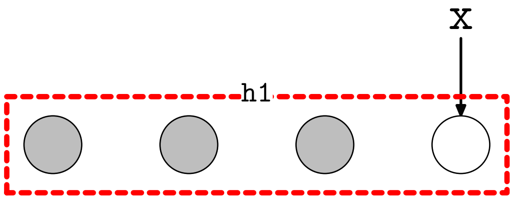
Abstraction: object recency [Balakrishnan & Reps, 2006]

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

recency-bit(o)

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

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



$\textsc{Alloc}_{k=\infty}$

- ✗ No amount of calling context helps

Abstraction: object recency [Balakrishnan & Reps, 2006]

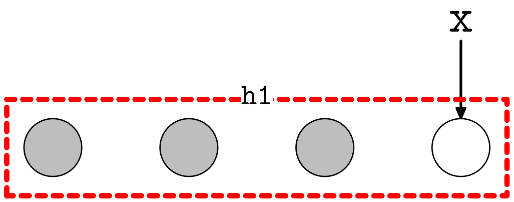
Abstraction $\textsc{Recency}^r_k$ (r is recency depth); for $r = 1$:

recency-bit(o)

Objects allocated:	o1	o2	o3	o4	o5
$\textsc{Alloc}_k$:	h2	h4	h4	h2	h4

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

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



$\textsc{Alloc}_{k=\infty}$

✗ No amount of calling context helps

Abstraction: object recency [Balakrishnan & Reps, 2006]

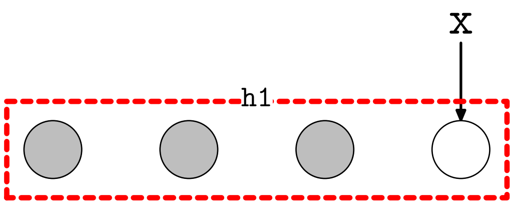
Abstraction $\textsc{Recency}_k^r$ (r is recency depth); for $r = 1$:

recency-bit(o)

Objects allocated:	o1	o2	o3	o4	o5
$\textsc{Alloc}_k$:	h2	h4	h4	h2	h4
recency-bit:	0	0	0	1	1

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

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



$\textsc{Alloc}_{k=\infty}$

✗ No amount of calling context helps

Abstraction: object recency [Balakrishnan & Reps, 2006]

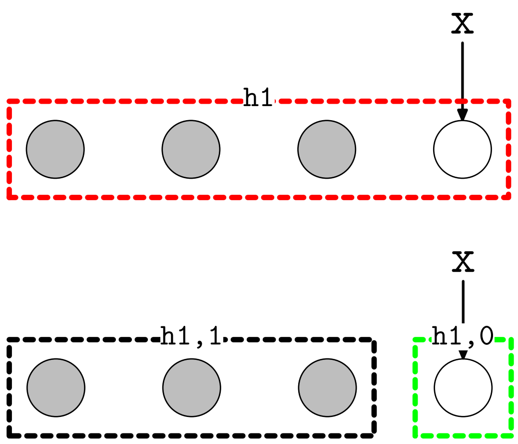
Abstraction $\textsc{Recency}_k^r$ (r is recency depth); for $r = 1$:

recency-bit(o)

Objects allocated:	o1	o2	o3	o4	o5
$\textsc{Alloc}_k$:	h2	h4	h4	h2	h4
recency-bit:	0	0	0	1	1

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

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



$\textsc{Alloc}_{k=\infty}$

$\textsc{Recency}^{r=1}$

✗ No amount of calling context helps

✓ Recency makes the proper distinctions

Abstraction: heap connectivity [Sagiv et al., 2002]

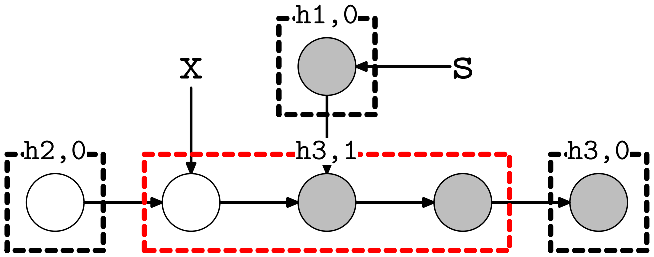
Common pattern: build linked list data structures

```
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

```
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 ...
```



$\textsc{Recency}^{r=\infty}$

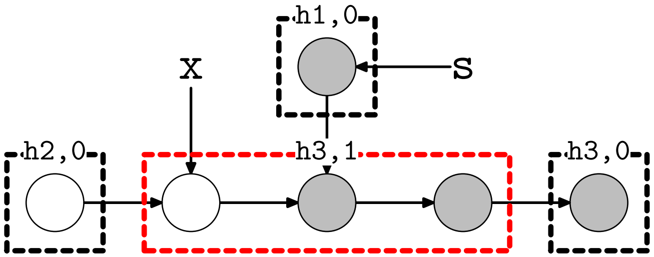
✗ No amount of recency helps

Abstraction: heap connectivity [Sagiv et al., 2002]

$\textsc{ReachFrom}_k$: set of alloc. sites reaching $\textsc{Alloc}_k(o)$

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 ...
```



$\textsc{Recency}^{r=\infty}$

✗ No amount of recency helps

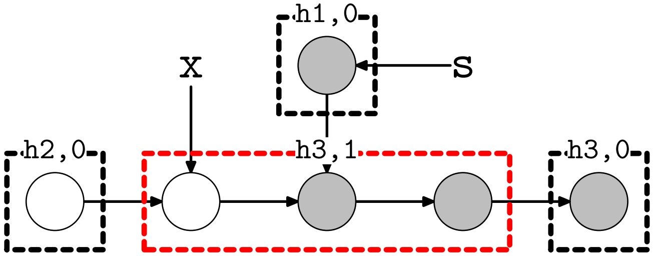
Abstraction: heap connectivity [Sagiv et al., 2002]

$\textsc{ReachFrom}_k$: set of alloc. sites reaching $\textsc{Alloc}_k(o)$

$\textsc{PointedToBy}_k$: set of alloc. sites reaching $\textsc{Alloc}_k(o)$ 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 ...
```



$\textsc{Recency}^{r=\infty}$

✗ No amount of recency helps

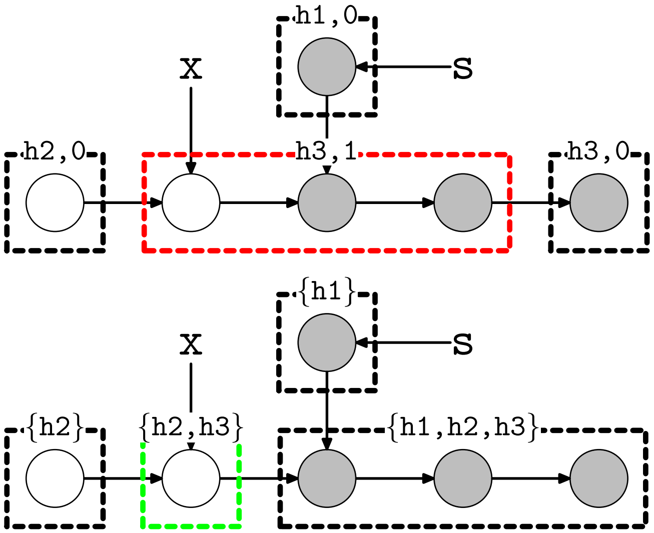
Abstraction: heap connectivity [Sagiv et al., 2002]

$\textsc{ReachFrom}_k$: set of alloc. sites reaching $\textsc{Alloc}_k(o)$

$\textsc{PointedToBy}_k$: set of alloc. sites reaching $\textsc{Alloc}_k(o)$ 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 ...
```



$\textsc{Recency}^{r=\infty}$

$\textsc{ReachFrom}_{k=0}$

- ✗ 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 `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

benchmark	ALLOC	$\text{ALLOC}_{k=5}$	RECENCY	REACHFROM
`antlr`	48.6	85.0	81.0	**100.0**
`avrora`	54.7	62.3	69.2	**77.8**
`batik`	13.5	15.1	**20.9**	**20.6**
`fop`	36.3	**99.3**	42.8	41.3
`hsqldb`	62.6	69.0	**94.3**	?
`luindex`	6.3	**97.2**	6.8	6.8
`lusearch`	14.3	**90.0**	19.0	19.6
`pmd`	12.4	**87.1**	14.9	14.6
`xalan`	64.0	**78.9**	**78.7**	76.6
average	34.8	**76.0**	47.5	44.7

Main points:

- ALLOC can be very imprecise
- $\text{ALLOC}_{k=5}$ works best most of the time

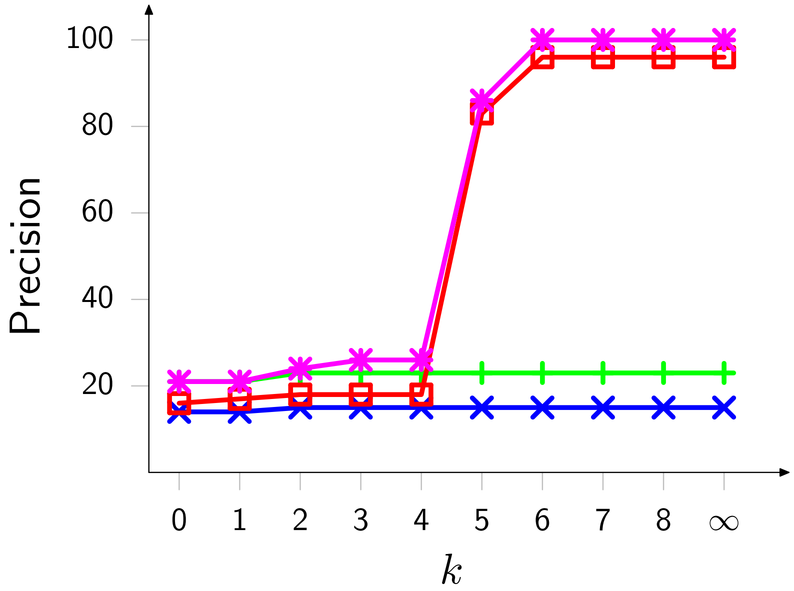
General results: NonStationaryField

benchmark	Alloc	$\text{Alloc}_{k=5}$	Recency	ReachFrom
`antlr`	59.1	60.1	**91.0**	78.3
`avrora`	33.2	33.6	**93.6**	77.2
`batik`	35.8	36.1	**99.5**	65.3
`fop`	42.0	44.9	**90.9**	68.2
`hsqldb`	45.4	49.5	**94.6**	?
`luindex`	78.0	84.2	**94.8**	**94.8**
`lusearch`	38.2	38.2	**64.9**	56.5
`pmd`	37.8	39.9	**96.4**	69.4
`xalan`	44.0	44.5	**90.4**	74.2
average	45.9	47.9	**90.7**	73.0

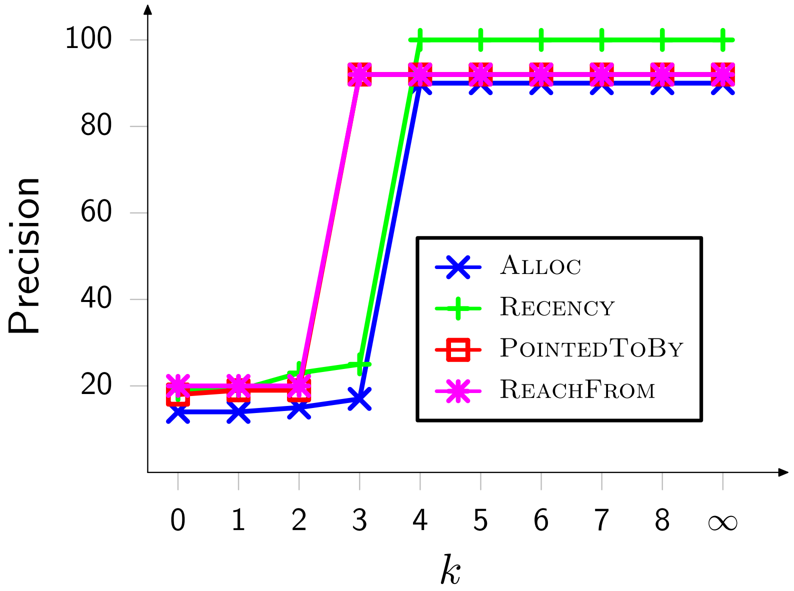
Main points:

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

Effect of call stack depth k



(a) (ThreadEscape, `batik`)



(b) (ThreadEscape, `lusearch`)

Main points:

- Phase transition: sharp increase in precision beyond $k \approx 5$
- Synergy of information: ReachFrom requires high k to be precise

Effect of recency depth

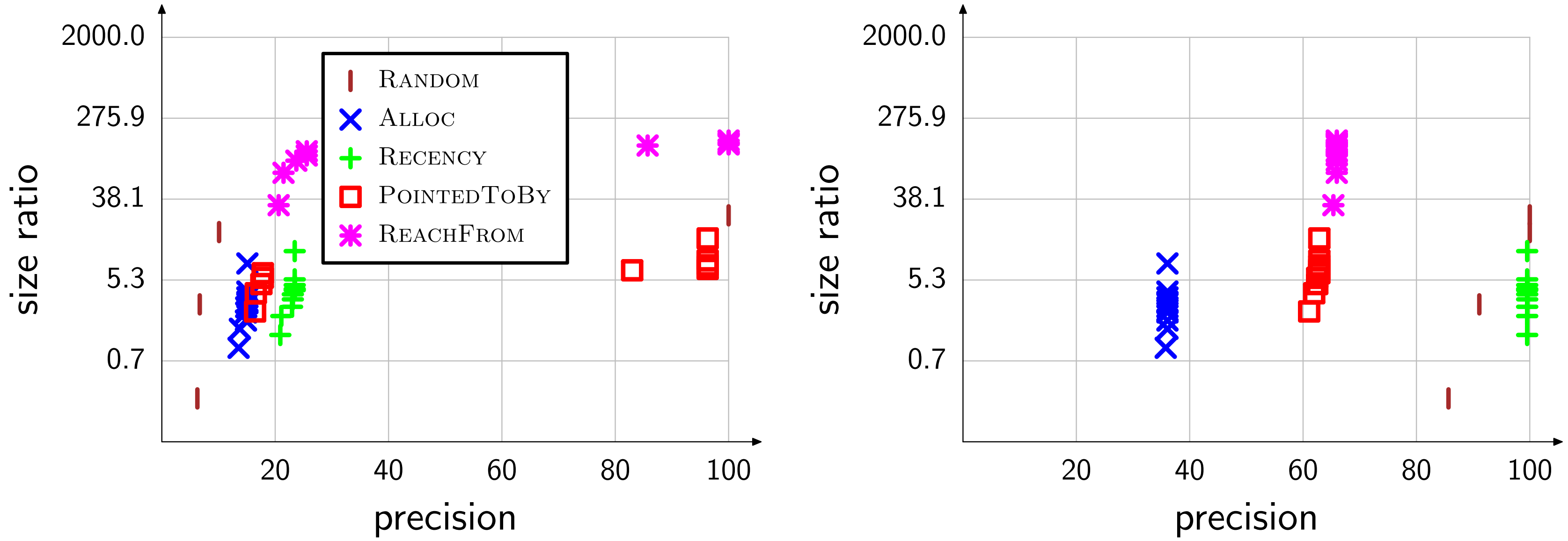
THREADESCAPE on `batik`:

	$r = 0$	$r = 1$	$r = 2$	$r = 3$	$r = 4$	$r = 5$
$k = 0$	13.5	20.9	21.4	**22.1**	**22.5**	**22.6**
$k = \infty$	15.1	23.4	**99.0**	**99.0**	**99.0**	**99.0**

Main points:

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

Tradeoff between precision and size



(a) (ThreadEscape, `batik`) (b) (NonStationaryField, `batik`)

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!