Mostly-functional behavior in Java programs

William C. Benton
Red Hat Emerging Technologies and
University of Wisconsin

Charles N. Fischer
University of Wisconsin

Motivation

We'd like to do aggressive code transformations, specification checking and analysis of large object-oriented programs.

The problem

Java programs are difficult to analyze, transform, and reason about (in part) due to mutable state.

Our contribution

Type-and-effect system and type-based analysis

Our contribution

Type-and-effect system and type-based analysis

Initialization effects

Our contribution

Type-and-effect system and type-based analysis

Initialization effects

Quiescing field inference

Our contribution

Type-and-effect system and type-based analysis

Initialization effects	Quiescing field inference	Degrees of method purity

Our contribution

Type-and-effect system and type-based analysis

Initialization effects	Quiescing field inference	Degrees of method purity

Our contribution

Type-and-effect system and type-based analysis

Initialization effects

Quiescing field inference

Degrees of method purity

Surprising result: substantial *mostly-functional behavior* in Java!

Forecast

- A simple object-oriented effects system
- Initializers and initialization effects
- Final fields and eventual immutability
- Inferring quiescing fields
- Evaluating quiescing field inference

Effect systems: motivation

```
// method1: List<T> → int
int method1(List<T> l) {
  return l.size();
}
```

```
// method2: List<T> → int
int method2(List<T> l) {
  int i = 0;
  while (! l.isEmpty() ) {
    l.remove(0);  i++;
  }
  return i;
}
```

Effect systems: motivation

```
// method1: List<T> → int
int method1(List<T> l) {
  return l.size();
}
```

READS state of *l*

```
// method2: List<T> → int
int method2(List<T> l) {
  int i = 0;
  while (! l.isEmpty() ) {
    l.remove(0);  i++;
  }
  return i;
}
```

Effect systems: motivation

```
// method1: List<T> → int
int method1(List<T> l) {
   return l.size();
}
```

READS state of *l*

READS, WRITES state of *l*

```
// method2: List<T> → int
int method2(List<T> l) {
   int i = 0;
   while (! l.isEmpty() ) {
      l.remove(0);  i++;
   }
   return i;
}
```

Effect systems: motivation

```
// method1: List<T> → int
int method1(List<T> l) {
  return l.size();
}
```

READS state of *l*

READS, WRITES state of *l*

```
// method2: List<T> → int
int method2(List<T> l) {
  int i = 0;
  while (! l.isEmpty() ) {
    l.remove(0);  i++;
  }
  return i;
}
```

Type systems: "what?"

Effect systems: "how?"

Effect systems: motivation

```
// method1: List<T> → int
int method1(List<T> l) {
  return l.size();
}
```

READS state of *l*

READS, WRITES state of *l*

```
// method2: List<T> → int
int method2(List<T> l) {
  int i = 0;
  while (! l.isEmpty() ) {
    l.remove(0);  i++;
  }
  return i;
}
```

"How" consists of an *effect* (READ or WRITE) in some *region*.

Object-oriented effect systems

- Classic effect systems typically feature lexically-scoped regions
- *Object-oriented effect systems* better support classes, fields, &c.
- See Greenhouse & Boyland (ECOOP 99) or Bierman & Parkinson (WOOD 03)

Inferring effects for bytecodes

$l = l_h.\nu$

$l_h.\nu = l$

method invocations

Inferring effects for bytecodes

$$l = l_h.v$$

Inferring effects for bytecodes

$$\frac{\mathsf{load}(s, l, l_h, \kappa.\nu) \qquad \mathsf{rpt}(l_h, \rho)}{\varphi(s) \sqsupseteq \textsc{read} : \{\langle \rho, \kappa.\nu \rangle\}\rangle} \textsc{Read}$$

Inferring effects for bytecodes

$$\frac{\mathsf{load}(s, l, l_h, \kappa.\nu) \qquad \mathsf{rpt}(l_h, \rho)}{\varphi(s) \sqsupseteq \textsc{read} : \{\langle \rho, \kappa.\nu \rangle\}\rangle} \; \textsc{read}$$

Regions we consider: ρ_{this}, $\rho_{0..n}$, and $\top$.

Inferring effects for bytecodes

$$\frac{\text{load}(s, l, l_h, \kappa.\nu) \qquad \text{rpt}(l_h, \rho)}{\varphi(s) \sqsupseteq \textsc{read} : \{\langle \rho, \kappa.\nu \rangle\}\rangle} \;\textsc{read}$$

$l_h.\nu = l$

Regions we consider: ρ_{this}, $\rho_{0..n}$, and $\top$.

Inferring effects for bytecodes

$$\frac{\text{load}(s, l, l_h, \kappa.\nu) \qquad \text{rpt}(l_h, \rho)}{\varphi(s) \sqsupseteq \textsc{read} : \{\langle \rho, \kappa.\nu \rangle\}\}\rangle} \; \textsc{read}$$

$$\frac{\text{store}(s, l_h, \kappa.\nu, l) \qquad \text{rpt}(l_h, \rho)}{\varphi(s) \sqsupseteq \textsc{write} : \{\langle \rho, \kappa.\nu \rangle\}\}\rangle} \; \textsc{write}$$

Regions we consider: ρ_{this}, $\rho_{0..n}$, and $\top$.

Inferring effects for bytecodes

$$\frac{\overset{\textsc{read}}{\mathsf{load}(s, l, l_h, \kappa.\nu) \qquad \mathsf{rpt}(l_h, \rho)}}{\varphi(s) \sqsupseteq \textsc{read} : \{\langle \rho, \kappa.\nu \rangle\}\rangle}$$

$$\frac{\overset{\textsc{write}}{\mathsf{store}(s, l_h, \kappa.\nu, l) \qquad \mathsf{rpt}(l_h, \rho)}}{\varphi(s) \sqsupseteq \textsc{write} : \{\langle \rho, \kappa.\nu \rangle\}\rangle}$$

method invocations

Regions we consider: ρ_{this}, $\rho_{0..n}$, and $\top$.

Inferring effects for bytecodes

$$\textsc{Read}\quad \frac{\mathsf{load}(s, l, l_h, \kappa.\nu) \qquad \mathsf{rpt}(l_h, \rho)}{\varphi(s) \sqsupseteq \textsc{read} : \{\langle \rho, \kappa.\nu \rangle\}\rangle}$$

$$\textsc{Write}\quad \frac{\mathsf{store}(s, l_h, \kappa.\nu, l) \qquad \mathsf{rpt}(l_h, \rho)}{\varphi(s) \sqsupseteq \textsc{write} : \{\langle \rho, \kappa.\nu \rangle\}\rangle}$$

$$\textsc{Summary}\quad \frac{s_0, \cdots, s_n \in m}{\varphi(m) \sqsupseteq \varphi(s_0) \sqcup \cdots \sqcup \varphi(s_n)}$$

$$\textsc{Call}\quad \frac{s \rightarrow m'}{\varphi(s) \sqsupseteq \mathsf{pmap}(s, \varphi(m'))}$$

Regions we consider: ρ_{this}, $\rho_{0..n}$, and $\top$.

Extending the simple system

Type-and-effect system and type-based analysis

Initialization effects

Quiescing field inference

Degrees of method purity

Extending the simple system

Type-and-effect system and type-based analysis

- Initialization effects
- Quiescing field inference
- Degrees of method purity

Extending the simple system

Type-and-effect system and type-based analysis

Initialization effects	Quiescing field inference	Degrees of method purity

Forecast

- A simple object-oriented effects system
- Initializers and initialization effects
- Final fields and eventual immutability
- Inferring quiescing fields
- Evaluating quiescing field inference

Motivating initialization effects

```
class IntBox {
  private int i;
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

Motivating initialization effects

class IntBox {
  private int *i*;
  IntBox(**int** *j*) {
    this.*i* = *j*;   **writes** IntBox.i to **this**
  }

  int get() {
    return *i*;   **reads** IntBox.i from **this**
  }
}

Motivating initialization effects

```
class IntBox {
  private int i;
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

writes IntBox.i to **this**

These two effects cannot interfere!

reads IntBox.i from **this**

Motivating initialization effects

class IntBox {
  private int *i*;
  IntBox(**int** *j*) {
    this.*i* = *j*;
  }

  int get() {
    return *i*;
  }
}

initializes IntBox.i of **this**

These two effects cannot interfere!

reads IntBox.i from **this**

Motivating initialization effects

```
class IntBox {
  private int i;
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

```
IntBox factory(int j) {
  IntBox r =
    new IntBox(j);
  return r;
}
```

Motivating initialization effects

```
class IntBox {
  private int i;
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

```
IntBox factory(int j) {
  IntBox r =
    new IntBox(j);
  return r;
}
```

"Pure" methods can modify newly-allocated objects (Leavens et al.; Rugina and Cherem)

Defining initialization effects

```
class IntBox {
  private int i;
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

initializes IntBox.i field of **this**

Defining initialization effects

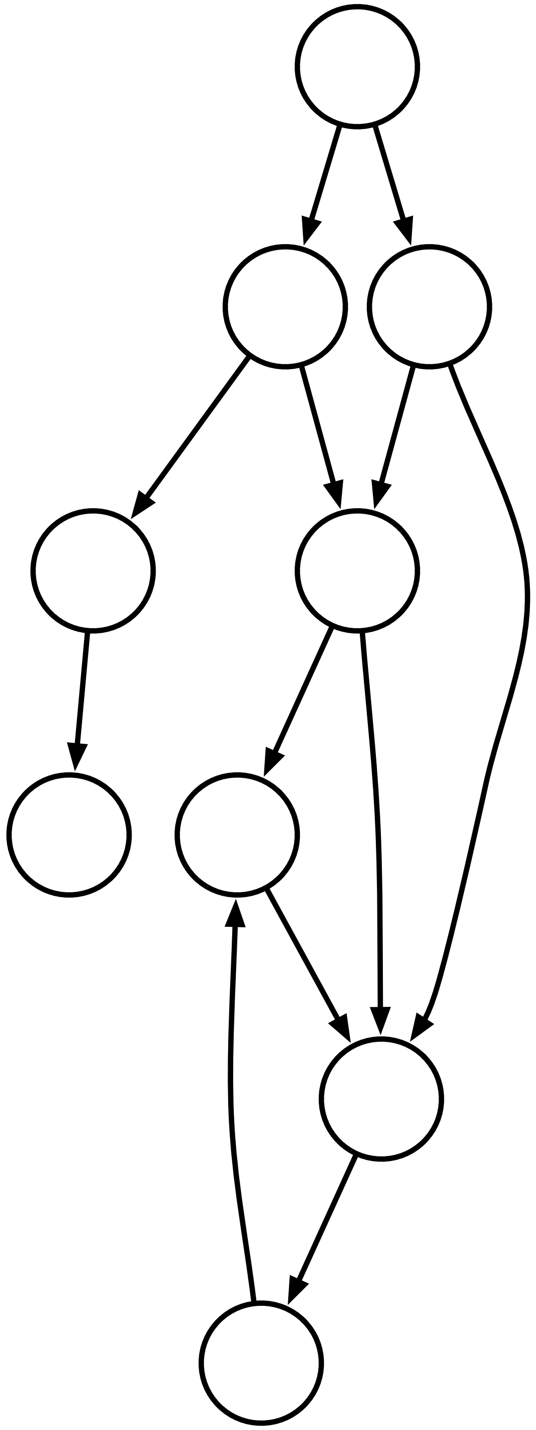
```
class IntBox {
  private int i;
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

An *initialization effect* is a WRITE to the state of an object during its creation.

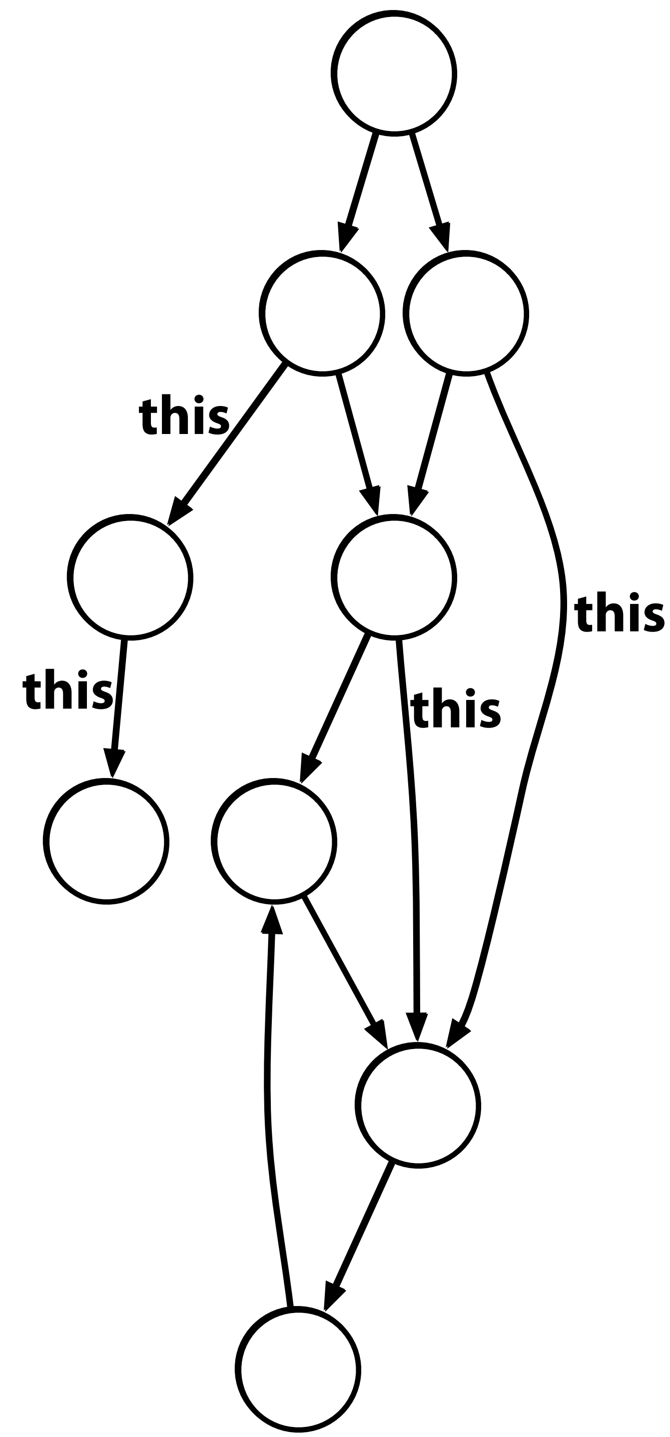
An *initializer* is a method that executes on an object during the dynamic lifetime of its constructor.

Inferring initializer methods

Inferring initializer methods

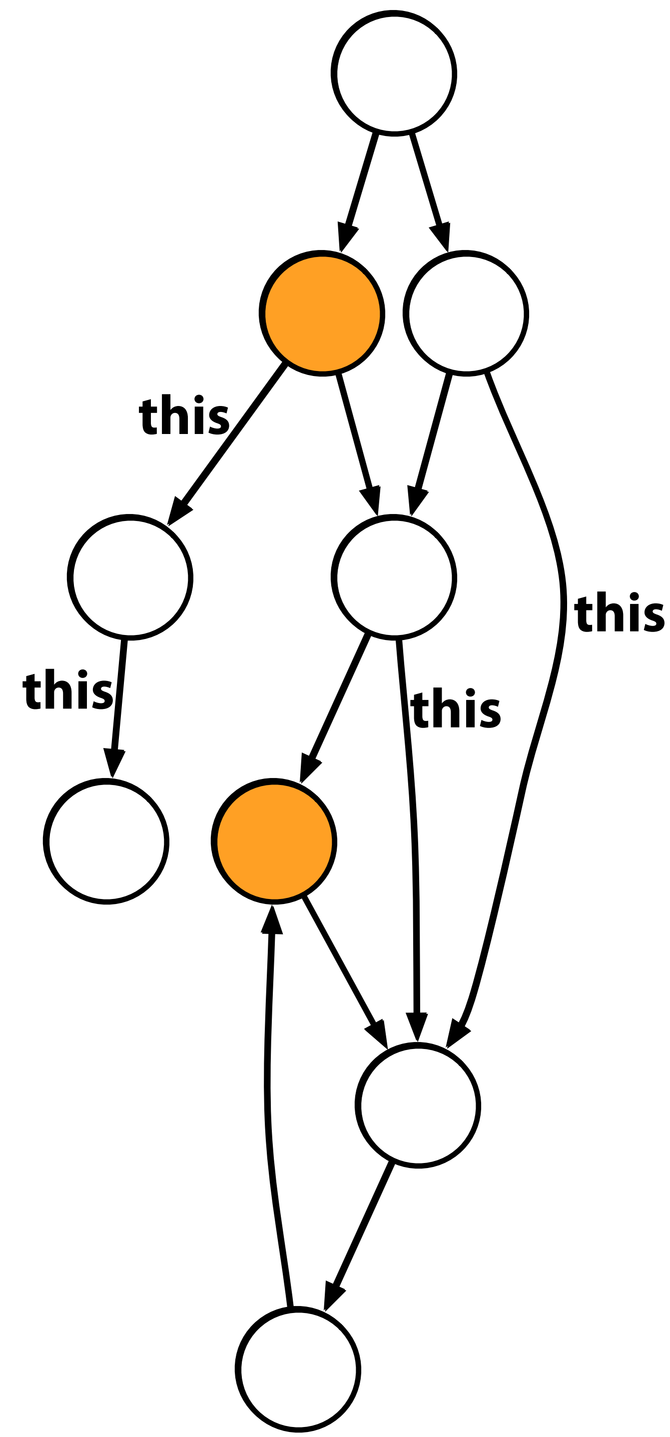


Inferring initializer methods



Inferring initializer methods

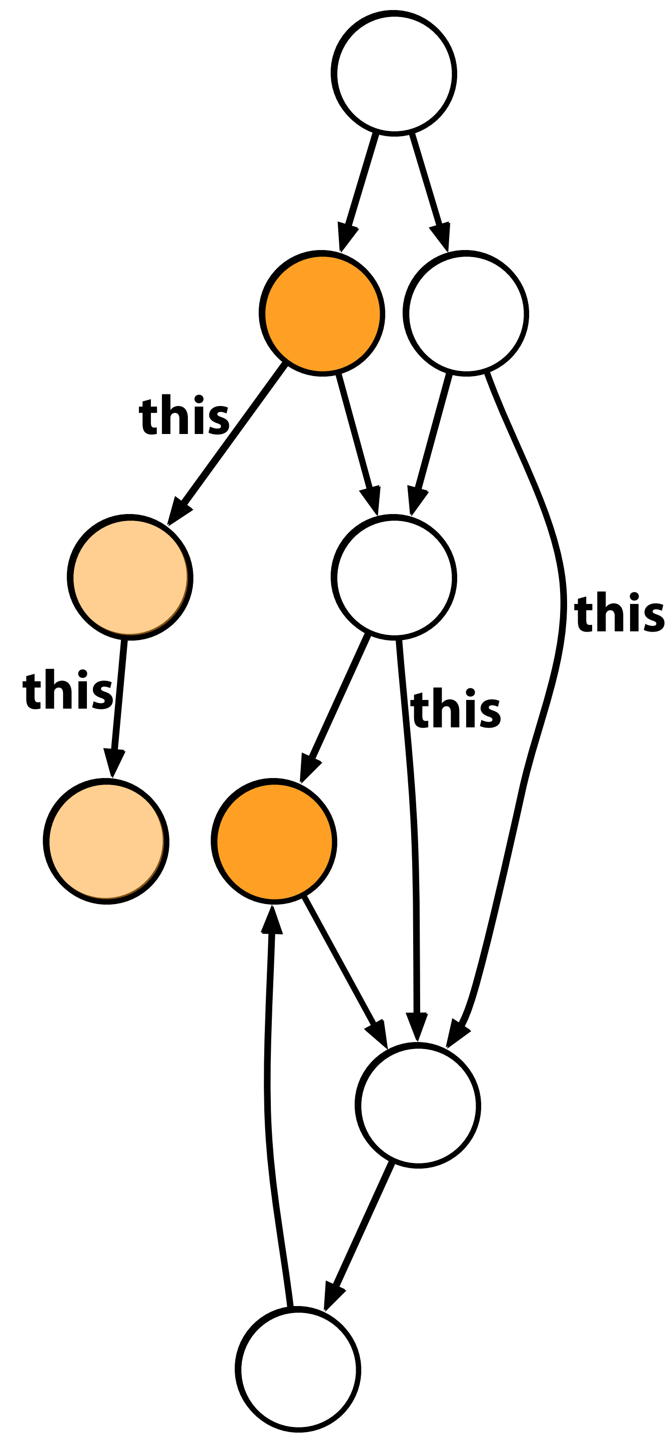
A constructor is an initializer on its receiver object.



Inferring initializer methods

A constructor is an initializer on its receiver object.

A method that is *only* invoked via this-edges from an initializer is also an initializer on its receiver object.



Inferring initialization effects

Initialization effects are writes to fields of this that occur within an initializer.

Forecast

- A simple object-oriented effects system
- Initializers and initialization effects
- Final fields and eventual immutability
- Inferring quiescing fields
- Evaluating quiescing field inference

Final fields

```
class IntBox {
  private int i;
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

Final fields

```
class IntBox {
  private final int i;
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

Final fields

```
class IntBox {
  private final int i;        i is a run-time constant
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

Final fields

```
class IntBox {
  private final int i;
  IntBox(int j) {
    this.i = j;
  }

  int get() {
    return i;
  }
}
```

Final fields

```
class IntBox {
  private final int i;
  IntBox(int j) {
    init(i);
  }

  int get() {
    return i;
  }
```

```
  private void init(int j) {
    this.i = j;
  }
}
```

Final fields

```
class IntBox {
  private final int i;
  IntBox(int j) {
    init(i);
  }

  int get() {
    return i;
  }

  private void init(int j) {
    this.i = j;
  }
}
```

Final fields *must be* assigned exactly once on every path through each constructor and *may only be* assigned in the constructor.

Final fields and immutability

- Java definition of **final** is restrictive, designed for simple verification
- Many fields that represent run-time constants are not declared **final**
- Several groups have developed analyses to find such fields

One example: stationary fields

- Unkel and Lam (2008): *stationary fields* are never written after they are read
- About 50% of fields in open-source Java programs can be inferred stationary; a much smaller percentage are **final**
- Their analysis is based on flow- and context-sensitive points-to analysis

Forecast

- A simple object-oriented effects system
- Initializers and initialization effects
- Final fields and eventual immutability
- Inferring quiescing fields
- Evaluating quiescing field inference

Quiescing fields

- A field is *quiescing* if it is *initialized* but never *written*; all final fields are quiescing
- Inference algorithm for these is straightforward: consider only fields that aren't implicated in a WRITE effect

Comparing kinds of fields

- All final fields are quiescing fields
- Some quiescing fields are not stationary
- Some stationary fields are not quiescing
- The inference algorithm for quiescing fields runs in seconds on substantial Java programs

Forecast

- A simple object-oriented effects system
- Initializers and initialization effects
- Final fields and eventual immutability
- Inferring quiescing fields
- Evaluating quiescing field inference

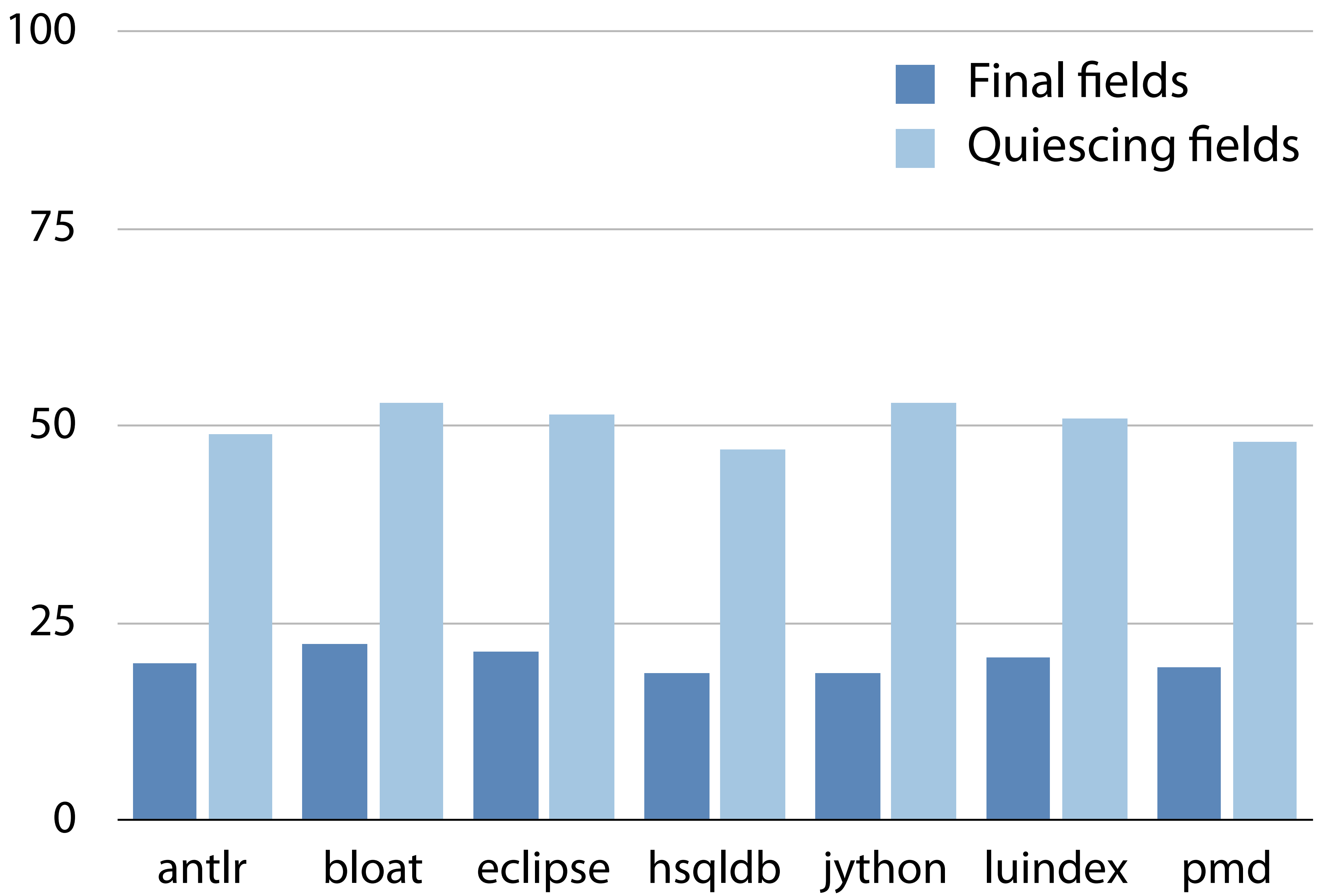
Evaluation

- Medium-sized Java programs from the DaCapo benchmark suite
- Soot and DIMPLE for bytecode analysis, performed on workstation hardware
- Executed benchmarks under instrumented Jikes RVM

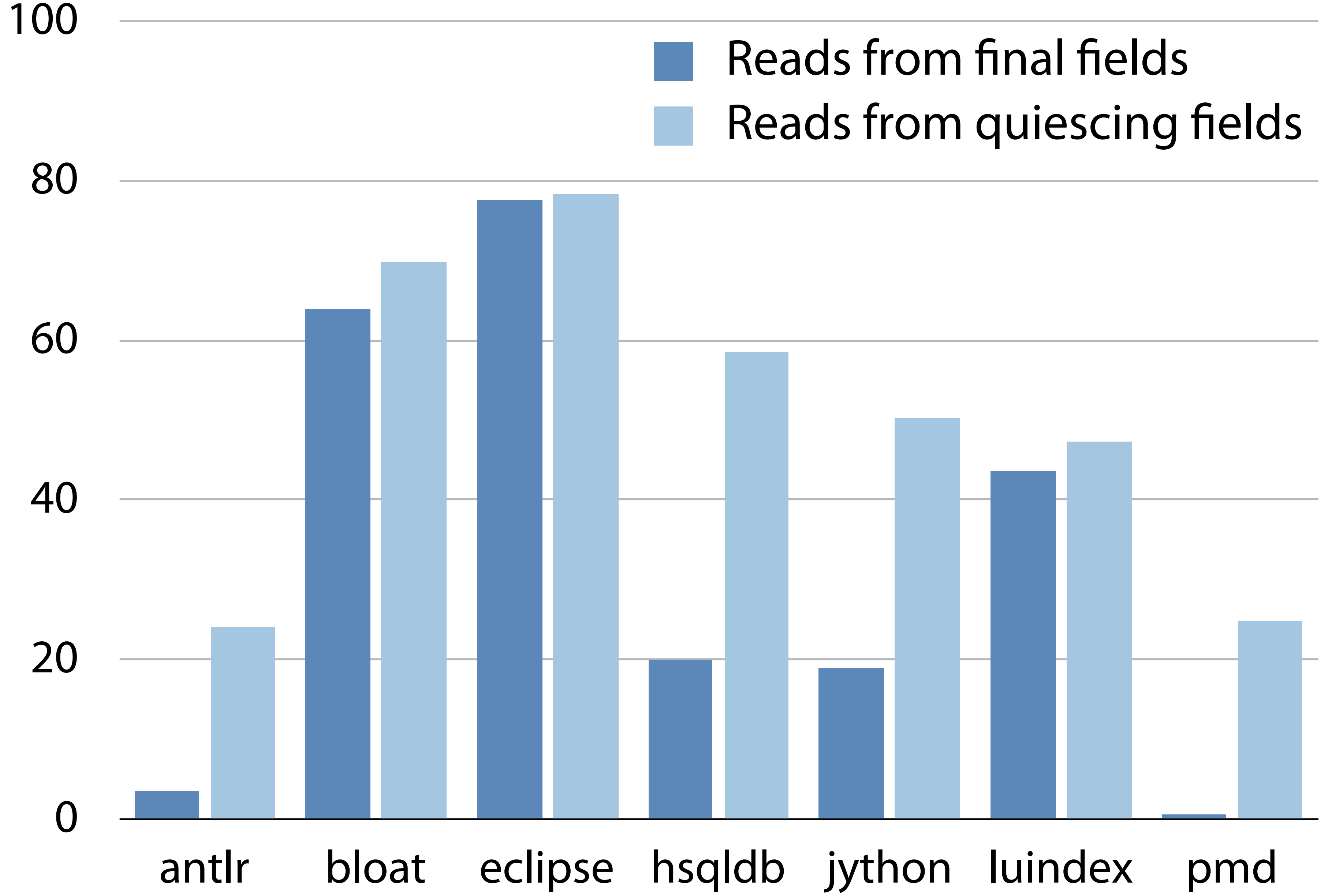
Benchmarks

	statements	classes	fields	methods
antlr	1.39 M	3729	14082	37209
bloat	1.41 M	3827	14524	33609
eclipse	1.38 M	3895	15161	33408
hsqldb	1.59 M	4190	17566	38504
jython	1.45 M	4058	14737	35604
luindex	1.35 M	3903	14511	32759
pmd	1.51 M	4265	15489	36393

Static prevalence of final and quiescing fields



Final and quiescing fields as a percentage of all dynamic reads



Conclusion

- Three novel features improve the precision of type-and-effect systems
- A significant portion of Java field reads are from fields with unchanging values
- It is possible to efficiently infer quiescing fields with type-based analyses

Thanks!

willb@acm.org

http://www.cs.wisc.edu/~willb/