Controlling Mutation and Aliases with Fractional Permissions

John Boyland

University of Wisconsin-Milwaukee

ECOOP '12
Outline of Session

I. Fractional Permissions
II. Applications
III. Problems
I. Fractional Permissions
Structured Aliasing?
My Thoughts
Structured Aliasing?
My Thoughts

- Aliasing is relevant ONLY if \( \exists \) mutable state.
Structured Aliasing?
My Thoughts

- Aliasing is relevant ONLY if $\exists$ mutable state.
- For isolation/parallelism, we want tracking.
Structured Aliasing?
My Thoughts

- Aliasing is relevant ONLY if ∃ mutable state.
- For isolation/parallelism, we want tracking.
- Fractions: don’t copy tokens, split tokens.
Structured Aliasing? My Thoughts

• Aliasing is relevant ONLY if \( \exists \) mutable state.
• For isolation/parallelism, we want tracking.
• Fractions: don’t copy tokens, \textit{split} tokens.
• Unify separation logic & ownership.
Fractional Permissions
Fractional Permissions

• Rough recipe:
Fractional Permissions

- Rough recipe:
- Separation Logic
Fractional Permissions

- Rough recipe:
  - Separation Logic
    - Scalars
Fractional Permissions

- Rough recipe:
  - Separation Logic
    - Scalars
    + Fractions
Fractional Permissions

- Rough recipe:
  - Separation Logic
    - Scalars
  + Fractions
  + Nesting (AKA Adoption)
Separation Logic

- Linear heap predicates
- flow-sensitive state information
- Nonlinear scalar predicates
- Hoare logic, and e.g., \( y = x + z \)
- Linear and nonlinear connectives
- and, or, implication, existentials, ...
- [Reynolds, O’Hearn, Brookes, Parkinson...]
Minus Scalars

• Separation Logic intends (full) verification.
• We do verification of limited properties:
  • aliasing, mutation, concurrency
  • object structure, design patterns
• Thus we drop integer and boolean values: only interested in pointer values.
Plus Fractions

• We distinguish write (one) from read (less than one) permission to mutable state.

• Important for concurrency correctness.

• Some variants of SL have fractions in some limited situations.

• Fractions are endemic for us.
Plus Nesting

- Motivation: nonlinear (flow-insensitive) analysis is simpler.
- Add a way to express flow-insensitive ownership along with flow-sensitive uniqueness
- aka “adoption” [Fähndrich&DeLine 2002]
- “Interesting” interactions with fractions.
I. Outline

• Fractions
• Fractional Heaps
• Fractional Permission Logic
Fractions
Fractions

- $\mathbb{Q}^+$ positive rational numbers ($\mathbb{R}^+$ OK).
Fractions

- $\mathbb{Q}^+$ positive rational numbers ($\mathbb{R}^+$ OK).
- No zero (0 permission = no permission).
Fractions

- $\mathbb{Q}^+$ positive rational numbers ($\mathbb{R}^+$ OK).
- No zero (0 permission = no permission).
- Both addition and multiplication are total and permit cancellation.
Fractions

- $\mathbb{Q}^+$ positive rational numbers ($\mathbb{R}^+$ OK).
- No zero (0 permission = no permission).
- Both addition and multiplication are total and permit cancellation.
- Fractions can be arbitrarily small.
Fractions

• \( \mathbb{Q}^+ \) positive rational numbers (\( \mathbb{R}^+ \) OK).

• No zero (0 permission = no permission).

• Both addition and multiplication are total and permit cancellation.

• Fractions can be arbitrarily small.

• Fractions greater than one permitted in intermediate heaps.
Fractional Heaps

- A partial map from locations $L$ to pairs of values $V$ and fractions $\mathbb{Q}^+$
- $L = O \times F$, pointer and field
- $V = O$, pointers to objects.
- A memory has only fractions of one.
Fractional Heaps
Scaling Fractional Heaps

\[ \frac{1}{2} \]
Some Heaps Add
Some Heaps Add
Some Heaps Don’t Add
Some Heaps Don’t Add
Some Heaps Don’t Add

Different Values Clash
Memory
Memory

whole fraction
Memory

not defined
Memory
Permission Logic

- $o.f \rightarrow o'$

- A “unit permission” (Wrigstad “token”)
Permission Logic

- $\Pi_1 + \Pi_2$

Heap for $\Pi_1$

Heap for $\Pi_2$
Permission Logic

- $\Pi_1 + \Pi_2$
Permission Logic

\[ \Pi_1 \rightarrow \Pi_2 \]
Permission Logic

- $\Pi_1 \rightarrow \Pi_2$
Permission Logic

- $0$

- $q\Pi$

Heap for $\Pi$
Permission Logic

- $0$
- $q\prod$ with $q = \frac{3}{2}$
Permission Logic

- **Existential:** \( \exists r \cdot (o.f \rightarrow r) + \Pi \)
  - obligatory unit permission yields witness

- **Conditional:** \( \Gamma \uparrow \Pi_1 : \Pi_2 \)
  - Here \( \Gamma \) is a boolean formula
    - true or false
Boolean Formulae

- true: $\top$
- conjunction: $\Gamma_1 \land \Gamma_2$
- negation: $\neg \Gamma$
- comparison: $o = o'$

- All these can be true, false or unknown
Boolean Formulae

- existential  \( \exists x \cdot \Gamma \)
- unrestricted
- predicates calls  \( p(\ldots) \)
- can be recursive

- Either can be true or unknown (not false).
Boolean Formulae

- nesting
  \[ \Pi \prec o.f \]
- ownership
- indicates that \( \Pi \) is available in \( o.f \rightarrow o' \)
- by itself, as with all formulae, has no heap.
- "Known" nesting may be less that "actual."
- \( \Pi_1 \leq \Pi_2 : \text{there is } \Pi_0, \Pi_0 + \Pi_1 = \Pi_2 \)
Unit Permission Again

- \( o.f \rightarrow o' \)

\[ \Pi \prec o.f \]
Unit Permission Again

- \( o.f \rightarrow o' \)

\[ \Pi \prec o.f \]

\[ \Pi_1 \prec o.f \]

where

\[ \Pi_1 \leq \Pi \]
Example: Point

- permission to write o’s x and y fields:
  \[(\exists r \cdot o.x \rightarrow r) + (\exists r \cdot o.y \rightarrow r)\]

- preceding permission nested in o’s “All” field
  \[(\exists r \cdot o.x \rightarrow r) + (\exists r \cdot o.y \rightarrow r) \prec o.All\]
Example: Point

- permission to write o’s \( x \) and \( y \) fields:

\[
(\exists r \cdot o.x \rightarrow r) + (\exists r \cdot o.y \rightarrow r)
\]

- preceding permission nested in o’s “All” field

\[
(\exists r \cdot o.x \rightarrow r) + (\exists r \cdot o.y \rightarrow r) \prec o.All
\]
Example: Point

- A point is an object we know has x and y:

\[
\text{Point}(r) = (\exists r' \cdot r.x \rightarrow r') + (\exists r' \cdot r.y \rightarrow r') < r.\text{All}
\]

- \(o\) is a Point

- \(o\) is a Point and we can read its state:

\[
\text{Point}(o) + q(o.\text{All} \rightarrow 0)
\]
Nesting

\[ o.x \rightarrow o_x + \]

\[ o.y \rightarrow o_y + \]

\[ o.\text{All} \rightarrow 0 \]
Nesting

\[ o.x \rightarrow o_x + \]

\[ o.y \rightarrow o_y + \]

\[ o.\text{All} \rightarrow 0 \]

\[ \text{Point}(o) + o.\text{All} \rightarrow 0 \]
Carving

\[ \text{Point}(o) + q(o.\text{All} \rightarrow 0) \]
Carving

Point(o) + q(o.All → 0)
Carving

Point(o) + q(o.All → 0)
Carving

Point(o) + q(o.All → 0)

q(o.x → r_x) + (q(o.x → r_x) → q(o.All → 0))

1
In General: Invariants

- Invariant(o,...) nests state in o.All
- Invariant(o,...) + o.All → 0 gives one access
- Carving permits breaking the invariant:
  
  \[ o.x \rightarrow r_x + ((o.x \rightarrow r_x) + (o.All \rightarrow 0)) \]
- Until o.All → 0 is needed again.
Linked-List Node

- Field “data” is a “unique” Point;
- Field “next” is a “unique” Node:

\[
\text{Node}(r) = \left( \begin{array}{c}
\exists r_p \cdot r.\text{data} \rightarrow r_p + r_p.\text{All} \rightarrow 0 + \text{Point}(r_p) + \\
\exists r_n \cdot r.\text{next} \rightarrow r_n + r_n.\text{All} \rightarrow 0 + \text{Node}(r_n) \\
\end{array} \right) < r.\text{All}
\]
Linked-List Node

- Field “data” is a “unique” Point;
- Field “next” is a “unique” Node:

\[
\text{Node}(r) = \left( \exists r_p \cdot r.\text{data} \rightarrow r_p + r_p.\text{All} \rightarrow 0 + \text{Point}(r_p) + \right) \left( \exists r_n \cdot r.\text{next} \rightarrow r_n + r_n.\text{All} \rightarrow 0 + \text{Node}(r_n) \right) < r.\text{All}
\]

Can you spot the problem?
Linked-List Node

- Field “data” is a “unique” Point;
- Field “next” is a “unique” Node:

\[
\text{Node}(r) = \langle \exists r_p \cdot r.\text{data} \rightarrow r_p + r_p.\text{All} \rightarrow 0 + \text{Point}(r_p) + \exists r_n \cdot r.\text{next} \rightarrow r_n + r_n.\text{All} \rightarrow 0 + \text{Node}(r_n) \rangle \approx r.\text{All}
\]
Linked-List Node

- Field “data” is a “unique” Point;
- Field “next” is a “unique” Node:

\[
\text{Node}(r) = \left( \exists r_p \cdot r.\text{data} \rightarrow r_p + r_p.\text{All} \rightarrow 0 + \text{Point}(r_p) + \exists r_n \cdot r.\text{next} \rightarrow r_n + r_n.\text{All} \rightarrow 0 + \text{Node}(r_n) \right) < r.\text{All}
\]

Every node points to a new next node!
Linked-List Node

\[ \text{Node}(r) = \left( \exists r_p \cdot r.\text{data} \rightarrow r_p + \right. \]
\[ \left. r_p.\text{All} \rightarrow 0 + \text{Point}(r_p) + \right. \]
\[ \exists r_n \cdot r.\text{next} \rightarrow r_n + \]
\[ r_n = 0 ? 0 : \]
\[ r_n.\text{All} \rightarrow 0 + \text{Node}(r_n) \]
\[ \prec r.\text{All} \]
Linked-List Node

\[ \text{Node}(r) = \begin{cases} \exists r_p \cdot r.\text{data} \to r_p + r_p.\text{All} \to 0 + \text{Point}(r_p) + \\
\exists r_n \cdot r.\text{next} \to r_n + r_n = 0 ? 0 : \\
\quad r_n.\text{All} \to 0 + \text{Node}(r_n) \\
\prec r.\text{All} \end{cases} \]

“not null” unique
Linked-List Node

\[
\text{Node}(r) = \begin{cases} 
\exists r_p \cdot r.\text{data} \to r_p + \\
r_p.\text{All} \to 0 + \text{Point}(r_p) + \\
\exists r_n \cdot r.\text{next} \to r_n + \\
r_n = 0 ? 0 : \\
r_n.\text{All} \to 0 + \text{Node}(r_n) \\
< r.\text{All}
\end{cases}
\]

“maybe null” unique
Two Element List

Node($n_1$) + $n_1$.All $\rightarrow$ 0
Two Element List

\[
\text{Node}(n_1) + n_1.\text{All} \rightarrow 0 = \frac{1}{2}(n_1.\text{All} \rightarrow 0) + \frac{1}{2}(n_1.\text{All} \rightarrow 0)
\]
Effective Uniqueness

• Read permission does not guarantee unique:

\[ n_1.data = n_2.data \]

is possible.
Effective Uniqueness

- Read permission does not guarantee unique:

\[ n_1.data = n_2.data \]

is possible
Iteration

Node(l) +
l.All → 0

Node n = l;

while (n != null) {
    Point p = n.data;
    p.x += 3;
    n = n.next;
}

Node(\(l\)) +
\(l.\text{All} \rightarrow 0\)

Node \(n = l;\)

while (\(n \neq \text{null}\)) {
    Point \(p = n.\text{data};\)
    \(p.x += 3;\)
    \(n = n.\text{next};\)
}
Node(l) + l.All → 0
Node n = l;
while (n != null) {
    Point p = n.data;
    p.x += 3;
    n = n.next;
}

Node(l) + (n = l) + l.All → 0
Node(n) + n.All → 0 → l.All → 0
Node\( (l) \) + \\
l.All \rightarrow 0 \\
Node n = l; \\
while (n != null) { \\
    Point p = n.data; \\
    p.x += 3; \\
    n = n.next; \\
}
Node(l) + l.All → 0

Node n = l;

while (n != null) {
    Point p = n.data;
    p.x += 3;
    n = n.next;
}

p.x → r_x +

p.x → r_x → p.All → 0 +

p.All → 0 → n.All → 0 +

n.All → 0 → l.All → 0
Iteration

Node($l$) + $l$.All $\rightarrow$ 0

Node $n = l$;

while ($n$ != null) {
  Point $p = n$.data;
  $p$.x += 3;
  $n = n$.next;
}

$n = 0 ? 0 : n$.All $\rightarrow$ 0 + Node($n$) +
($n = 0 ? 0 : n$.All $\rightarrow$ 0 + Node($n$)) $\rightarrow$ $l$.All $\rightarrow$ 0
II. Applications
Annotations as Sugar

- (maybe null) “unique”
- field’s value nested with \( o.\text{All} \rightarrow 0 \)
- method effects are lent permissions
  - passed in on entry; passed out on exit
  - \( \text{writes}(x.f) \equiv x.f \rightarrow r \)
  - \( \text{reads}(x.f) \equiv z(x.f \rightarrow r) \)
Annotations as Sugar

• (maybe null) “unique”
• field’s value nested with $o.\text{All} \rightarrow 0$
• method effects are lent permissions
• passed in on entry; passed out on exit
• $\text{writes}(x.f) \equiv x.f \rightarrow r$
• $\text{reads}(x.f) \equiv z(x.f \rightarrow r)$

Fraction variable
Annotations as Sugar

- Shorthand: $r.\text{All} \equiv r.\text{All} \rightarrow 0$
- Ownership: one object’s state in another:
  - $\text{owned}(o)r \equiv r.\text{All} \prec o.\text{All}$
Independent Iterators

interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}

interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}

Independent Iterators

```
interface Iterator {
    reads(All) boolean hasNext();
    writes(All) Object next();
    writes(All) void remove();
}
```
Why Independent?

1. Utility methods can use iterator alone.
2. Iterators can be concatenated
3. Iteration creation can be delegated.
class Util {
    writes(it.All) static
    int count(Iterator it, Object x) {
        int c = 0;
        while (it.hasNext()) {
            if (x == it.next()) ++c;
        }
        return c;
    }
}

UTILITY
class AppendIterator implements Iterator {
    unique Iterator it1;
    unique Iterator it2;
    AppendIterator(unique Iterator i1, 
                   unique Iterator i2) {
        it1 = i1; it2 = i2;
    }
    reads(All) boolean hasNext() {
        return it1.hasNext() || it2.hasNext();
    }
    ...
class SpecializedCollection {
    Object[] data;
    ...
    Iterator iterator() {
        return Arrays.asList(data).iterator();
    }
    ...
}
Can Iterators be Independent?

• Collections are affected by iterators:
  • Can “remove” using iterator.

• Iterators are affected by collections:
  • If collection modified, iterators invalid.
Aliasing

- Iterator and collection share mutable state:
Interference

- R-W; W-R; W-W
  1. access through iterator
  2. access through collection
     R-W: iterator becomes invalid
     W-R: collection results different
     W-W: both
Avoiding Interference 1

- R-R is OK
- If we can prevent interference:
  - we can pretend no state overlap
Avoiding Interference 2

prevent collection writes while iterators in use

prevent collection reads while remove iterator in use
Avoiding Interference 2

- Prevent collection writes while iterators in use
- Prevent collection reads while remove iterator in use
- Creating iterator requires read access
- Creating remove iterator requires write access
Avoiding Interference 3

prevent collection writes while iterators in use

prevent collection reads while remove iterator in use

creating iterator encumbers read access

creating remove iterator encumbers write access
List Representation

class LinkedList<T, Object o> {
    class Node {
        T data;
        Node next;
    }
}
List Representation

class LinkedList<T, Object o> {
    class Node {
        owned(o) T data;
        unique Node next;
    }
}
List Representation

class LinkedList<T, Object o> {
    class Node {
        T data;
        Node next;
    }

    Node(r, r.o) =

    ∃r_d · r.data → r_d +
    (r_d = 0 ? 0 : r_d.All < r_o.All) +

    ∃r_n · r.next → r_n +
    (r_n = 0 ? 0 : r_n.All + Node(r_n, r_o))

    ≺ r.All
class LinkedList<T, Object o> {
    class Node {
        T data;
        Node next;
    }
    Node head;
}
List Representation

class LinkedList<T, Object o> { 
    class Node { 
        int data;
        Node next;
    } 
    Node head;

    LinkedList(r, r_o) = 

    \exists r_h \cdot r.\text{head} \rightarrow r_h +
    (r_h = 0 \ ? 0 : r_h.\text{All} + \text{Node}(r_h, r_o)) < r.\text{All}
Iterator Representation

class LinkedList<T, Object o> {
    class Iterator<fraction q> {
        Node pre;
    }
}
Iterator Representation

class LinkedList<T, Object o> {
    class Iterator<fraction q> {
        Node pre;
    }

    Iterator(r, r_o, z, r_l) =

    \exists r_p \cdot r.pre \rightarrow r_p +

    r_p = 0 ? 0 : zr_p.All + Node(r_p, r_o) +

    (r_p = 0 ? 0 : zr_p.All + Node(r_p, r_o)) \rightarrow zr_l.All

    \prec r.All
Iterator Creation

class LinkedList<T, Object o> {
    Iterator<z> iterator() {
        return new Iterator();
    }
}
class LinkedList<T, Object o> {
    Iterator<z> iterator() {
        return new Iterator();
    }
    “this”
In: $z^R_t$. All
class LinkedList<T, Object o> { 
    Iterator<T> iterator() { 
        return new Iterator(); 
    } 
}

In: \( zr_t \cdot \text{All} \)

Out: \( r_r \cdot \text{All} + (r_r \cdot \text{All} \rightarrow zr_t \cdot \text{All} + \nu) \)

result
Iterator Creation

class LinkedList<T, Object o> {
    Iterator<z> iterator() {
        return new Iterator();
    }
}

In: $z_{r_t}.All$

Out: $r_r.All + (r_r.All \rightarrow z_{r_t}.All + \bigcirc)$

some leftover permissions
Iterator Heaps

\[
\text{LinkedList}(l, o) + \frac{1}{2} l.\text{All} \rightarrow 0
\]
Iterator Heaps

\[
\text{LinkedList}(l, o) + \frac{1}{2} l.\text{All} \rightarrow 0
\]

\[
\text{Iterator}(i, o, \frac{1}{2}, l) + i.\text{All} + (i.\text{All} \rightarrow \frac{1}{2} l.\text{All} + v)
\]
Iterator Heaps

\[
\text{LinkedList}(l, o) + \frac{1}{2} l.\text{All} \rightarrow 0
\]

\[
\text{Iterator}(i, o, \frac{1}{2}, l) + i.\text{All} + (i.\text{All} \rightarrow \frac{1}{2} l.\text{All} + v)
\]
Iterator Heaps

$$\text{LinkedList}(l, o) + \frac{1}{2} l.\text{All} \rightarrow 0$$

$$\text{Iterator}(i, o, \frac{1}{2}, l) + (i.\text{All} \rightarrow \frac{1}{2} l.\text{All} + v)$$

Before

$$\begin{align*}
(l,\text{All}) & \quad (l,\text{head}) \\
(n_1,\text{All}) & \quad (n_1,\text{data}) \\
(n_1,\text{next}) & \quad (n_2,\text{All}) \\
(n_2,\text{data}) & \quad (n_2,\text{next})
\end{align*}$$

After

$$\begin{align*}
&(i,\text{All}) \\
&(i,\text{pre})
\end{align*}$$
Iterator Heaps

\[
\text{LinkedList}(l, o) + \frac{1}{2} l.\text{All} \rightarrow 0
\]

\[
\text{Iterator}(i, o, \frac{1}{2}, l) + i.\text{All} + (i.\text{All} \rightarrow \frac{1}{2} l.\text{All} + v)
\]
Concurrency

- Non-interacting parallelism:
- Divide permissions between threads.
- Mutual exclusion locks ("synchronized"):
  - Protected permissions nested in lock.
- "Volatile" fields, can be accessed anytime:
  - State of value nested in known location.
Noninteracting Parallelism
Noninteracting Parallelism
Noninteracting Parallelism
Synchronization
Synchronization

- Objects serving as locks are owned by Lock:
  - $l.\text{All} \preceq G.\text{Lock}$
Synchronization

- Objects serving as locks are owned by `Lock`:
  - `l.All < G.Lock`
- State protected by a lock is nested in it:
  - e.g. `l.val < l.All`
Synchronization

• Objects serving as locks are owned by Lock:
  • \( l.\text{All} \prec G.\text{Lock} \)
• State protected by a lock is nested in it:
  • e.g. \( l.\text{val} \prec l.\text{All} \)
• Synchronization provides \( l.\text{All} \):
  • requires \( l.\text{All} \) or \( l \) “after” last locked.
Synchronization

synchronized (p) {
    p.x += 1;
}

Synchronization

Point(r_p) \land (r_p.\text{All} \prec G.\text{Lock}) \land (l \prec r_p)

synchronized (p) {
    p.x += 1;
}

Synchronization

Point($r_p$) \land (r_p.\text{All} \prec G.\text{Lock}) \land (l < r_p)

synchronized (p) {
    p.x += 1;
}

New formula that compares lock order. Here $l$ is the previously locked location.
Synchronization

\[ \text{Point}(r_p) \land (r_p.\text{All} \prec G.\text{Lock}) \land (l < r_p) \]

synchronized (p) {
    \[ r_p.\text{All} \]
    p.x += 1;
}

Synchronization

Point($r_p$) $\land$ ($r_p$.All $\prec$ G.Lock) $\land$ ($l < r_p$)

synchronized (p) {
    $r_p$.All
    p.x += 1;
    $r_p$.x + ($r_p$.x $\rightarrow$ $r_p$.All)
}

Point($r_p$) $\land$ ($r_p$.All $\prec$ G.Lock) $\land$ ($l < r_p$)
Synchronization

synchronized (p) {
    p.x += 1;
}
synchronized (p) {
    p.x += 1;
}

1
synchronized (p) {
    p.x += 1;
}

Synchronization
Synchronization

synchronized (p) {
    p.x += 1;
}
Synchronization

synchronized (p) {

    p.x += 1;

}
Synchronization

\[ r_p.\text{All} \]

synchronized (p) {
    p.x += 1;
}

REENTRANCY
Synchronization

\[
synchronized (p) \{ \\
p.x += 1; \\
\}
\]

\[
\text{REENTRANCY} \\
r_p.\text{All}
\]

\[
\text{All} \\
r_p.\text{All}
\]
Synchronization

```java
synchronized (p) {
    p.x += 1;
    r_p.All
}
```
Synchronization

```
synchronized (p) {
    p.x += 1;
}
```

**REENTRANCY**

\[
\]
Volatile Fields

• Can be assigned at any time.

• Value assigned should have state nested...
  1. In known owner, or
  2. In G.Lock, or
  3. Fractionally in G.Immutable.
p = new Point(1,2);

// nest p in thread2

x.v = p;
Volatile Fields (1)

```java
p = new Point(1,2);
// nest p in thread2
x.v = p;
```

THREAD₁

THREAD₂
Volatile Fields (1)

```
p = new Point(1,2);

// nest p in thread2
x.v = p;
```
Volatile Fields (1)

p = new Point(1,2);

// nest p in thread2

x.v = p;
Volatile Fields (2)

p = new Point(1,2);

// make p a lock

x.v = p;
Volatile Fields (2)

```
p = new Point(1,2);
// make p a lock
x.v = p;
```
Volatile Fields (2)

```java
p = new Point(1,2);
// make p a lock
x.v = p;
```
p = new Point(1,2);

// make p a lock

x.v = p;
Immutability

• Nest any fraction of r.All in G.Immutable:

• $\exists z \cdot (z r.\text{All} \prec G.\text{Immutable})$

• Implicitly:

• Every thread nests a fraction of $G.\text{Immutable}$

• Every method gets a fraction of $G.\text{Immutable}$

• Fractions can get arbitrarily small.
Volatile Fields (3)

**THREAD\textsubscript{1}**

\[
p = \text{new Point(1,2)};
\]

// make p immutable

\[
x.v = p;
\]

**THREAD\textsubscript{2}**

\[
\]

(G,Immutable)
Volatile Fields (3)

THREAD\textsubscript{1}

\texttt{p = new Point(1,2);}

// make p immutable

\texttt{x.v = p; (G,Immutable)}

THREAD\textsubscript{2}

\texttt{(G,Immutable)}

1

1
Volatile Fields (3)

p = new Point(1,2);

// make p immutable

x.v = p; (G, Immutable)
Volatile Fields (3)

```java
p = new Point(1,2);
// make p immutable
x.v = p;
```

THREAD₁

THREAD₂

(G,Immutable)

(G,Immutable)
III. Problems
AWT/Swing

- (J)Component state can only be accessed from Swing Event Threads
- If an Event Thread blocks on input (e.g. using a JDialog), new thread is created/reused.
- Runnable instances can be passed to Swing.
Java Concurrency

- `wait()`
  - releases lock temporarily
- `Thread.start()`, `Thread.join()`
- Anyone can “join” a thread, multiple times
- `java.util.concurrent.ReentrantLock`
- can lock/unlock independently
Related Work

Ownership

Reads

Linearity
Related Work

Ownership
[Clarke 01]

Linearity

Reads
Related Work

Ownership

Reads

[Clarke & Drossopoulou 02]

Linearity
Related Work

Ownership

[ Müller & Rudich 07 ]

Reads

Linearity
Related Work

Ownership

Linearity

Reads
Related Work

Ownership

Reads

Linearity

[Ishtiaq & O’Hearn 01]
Related Work

Ownership

Reads

[Parkinson & Bierman 12]
Linearity
Related Work

Ownership

Linearity

Reads

[Fähndrich & Deline 02]
Related Work

Ownership

Reads

Linearity

[Bornat et al 05/Brookes 06]
Related Work

Ownership

Reads

Linearity
Related Work

Ownership

[Reads]

[Bierhoff & Aldrich 07]

Linearity