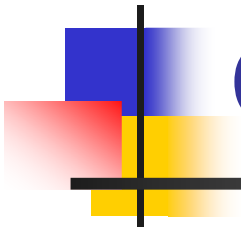
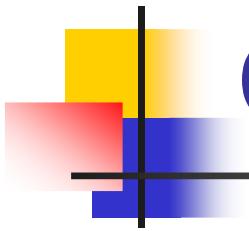


Ownership and Immutability in Generic Java (OIGJ)

Yoav Zibin[+], Alex Potanin[*]
Paley Li[*], Mahmood Ali[^], and Michael Ernst[$]

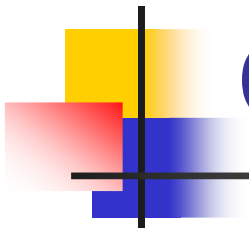
Presenter: Yossi Gil[+]

[+]IBM [*]Victoria,NZ [^]MIT [$]Washington



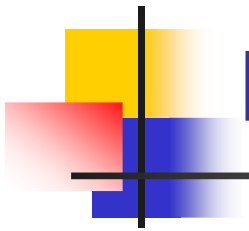
Ownership + Immutability

- Our previous work
  - OGJ: added Ownership to Java
  - IGJ: added Immutability to Java
- This work
  - OIGJ: combine Ownership + Immutability
  - The sum is greater than its parts
    - IGJ could not type-check *existing code* for creating **immutable cyclic data-structures** (e.g., lists, trees)
    - We found a non-trivial connection between ownership and immutability



Contributions

- No refactoring of existing code
  - Prototype implementation
    - No syntax changes (uses type-annotations in Java 7)
    - No runtime overhead
    - Backward compatible
  - Verified that Java's collection classes are properly encapsulated (using few annotations)
- Flexibility
  - OIGJ can type-check more code than previous work: cyclic structures, the factory and visitor design patterns
- Formalization
  - Formalized the concepts of raw/cooked immutable objects and wildcards as owner parameters
  - Proved soundness



Problem 1: Representation exposure

- Internal representation leaks to the outside
  - **`private`** doesn't offer real protection!

```
class Class {
  private List signers;
  public List getSigners() {
    return this.signers;
  }
}
```

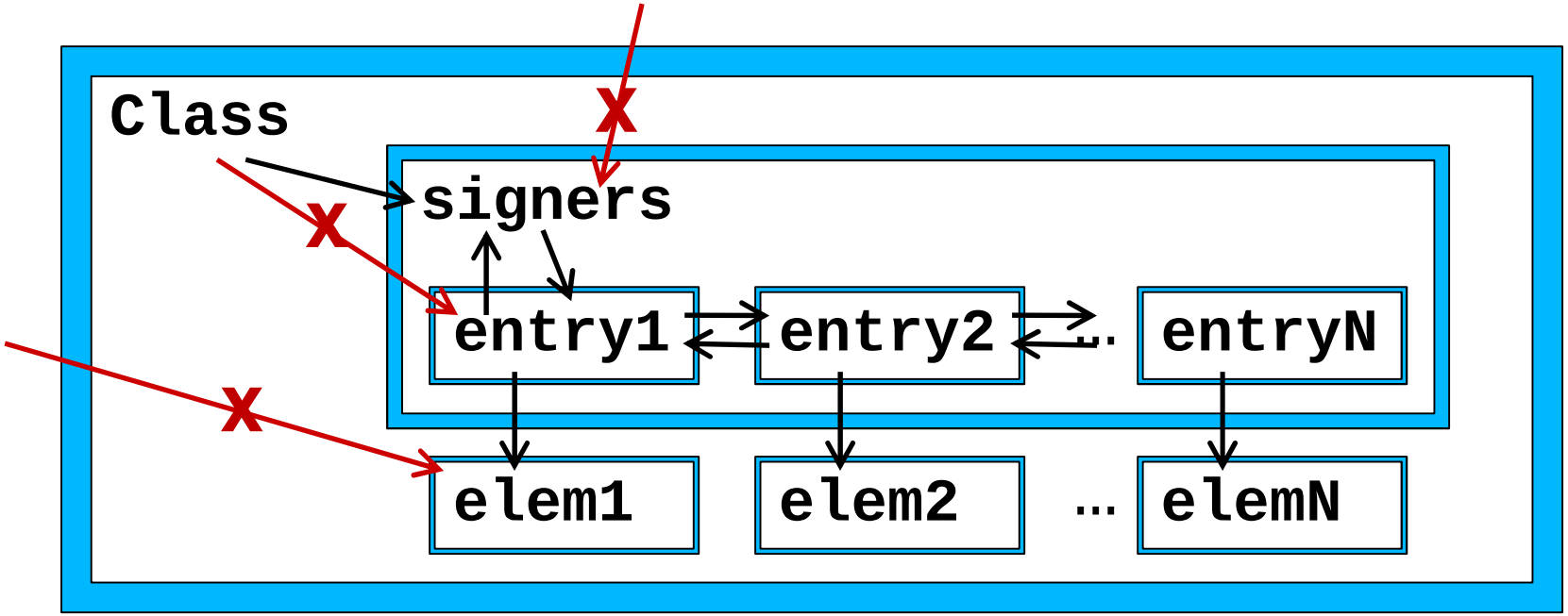
Real life example!

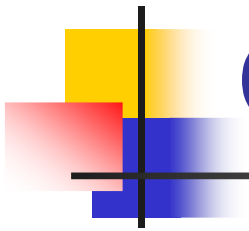
Forgot to copy signers!

http://java.sun.com/security/getSigners.html
Bug: the system thinks that code signed by one identity is signed by a different identity

Solution for Representation Exposure

- Ownership!
  - **`Class`** should own the list **`signers`**
  - No **outside** alias can exist
  - Ownership can be nested: note the tree structure





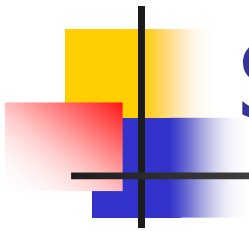
Ownership: Owner-as-dominator

- Dominators in graph theory
  - Given: a directed rooted graph
  - X *dominates* Y if any path from the root to Y passes X
- Owner-as-*dominator*
  - Object graph; roots are the static variables
  - An object cannot leak outside its owner, i.e.,
  - Any path from a root to an object passes its owner
  - Conclusion: No aliases to internal state

Problem 2: Unintended Modification

- Modification is not explicit in the language
  - can **`Map.get()`** modify the map?
  - **`for (Object key : map.keySet()) { map.get(key); }`**
    throws **`ConcurrentModificationException`**
    for the following map
    `new LinkedHashMap(100, 1, true)`

Reorders elements according to last-accessed (like a cache)



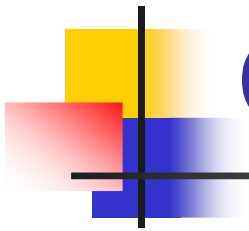
Solution: Immutability

- Varieties of Immutability
  - Class immutability (like String or Integer in Java)
  - Object immutability
    - The same class may have both mutable and immutable instances
  - Reference immutability
    - A particular reference cannot be used to mutate its referent (but other aliases might cause mutations)

Example in IGJ syntax

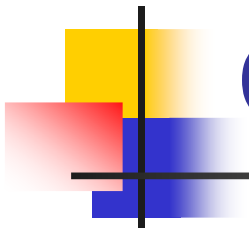
```
class Student {
  @Immutable Date dateOfBirth; …
  void setTutor(@ReadOnly Student tutor) @Mutable { … }
}
```

Method may modify the **this** object



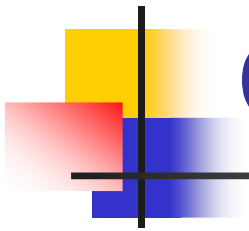
Objects vs. References

- Objects
  - mutable or immutable
  - Creation of an immutable object
    - **Raw** state: Fields can be assigned
    - **Cooked** state: Fields cannot be assigned
- References
  - mutable, immutable, or **readonly**



Challenge: Cyclic Immutability

- Cooking a cyclic data-structure is complicated
  - Many objects must be raw simultaneously to manipulate backward pointers
  - Then everything must become immutable simultaneously
- OIGJ's novel idea:
  - Prolong the cooking phase by using **ownership** information
  - Enables creation of **immutable** cyclic structures



Cooking immutable objects

- Previous work
  - An object becomes cooked when **its** constructor finishes
- OIGJ's observation
  - An object becomes cooked when **its owner's** constructor finishes
  - The outside world will not see this cooking phase
  - The complex object with its representation becomes immutable simultenously

Cooking LinkedList (1 of 2)

Sun's code is similar

```
1 : LinkedList(Collection<E> c) {
2 :  this();
3 :  Entry<E> succ = this.header, pred = succ.prev;
4 :  for (E e : c) {
5 :    Entry<E> entry =
6 :      new Entry<E>(e,succ,pred);
7 :    // An entry is modified after it's constructor finished
8 :    pred.next = entry; pred = entry;
9 :  }
10:  succ.prev = pred;
11: }
```

- No refactoring – the original code must compile in OIGJ

Cooking LinkedList (2 of 2)

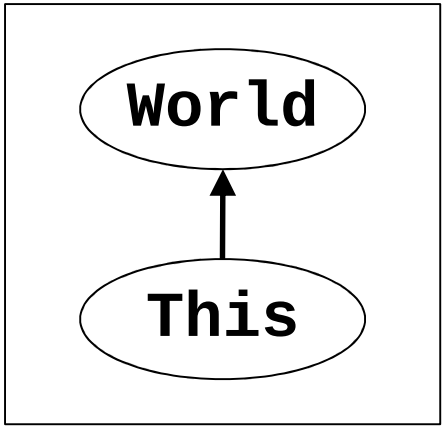
Sun's code is similar

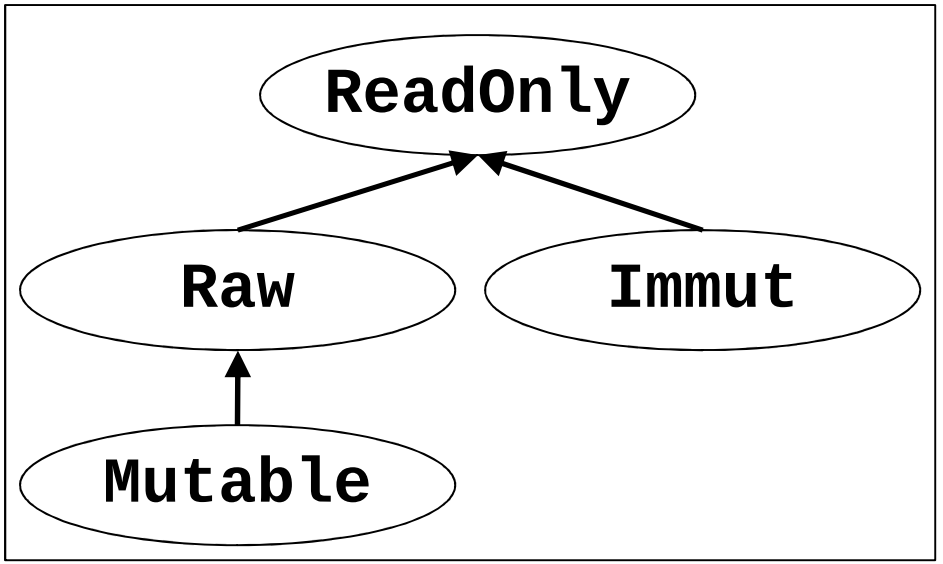
```
1 : LinkedList(@ReadOnly Collection<E> c) @Raw {
2 :  this();
3 :  @This @I Entry<E> succ = this.header, pred = succ.prev;
4 :  for (E e : c) {
5 :    @This @I Entry<E> entry =
6 :      new @This @I Entry<E>(e,succ,pred);
7 :    // An entry is modified after it's constructor finished
8 :    pred.next = entry; pred = entry;
9 :  }
10:  succ.prev = pred;
11: }
```

Code in OIGJ; Annotations next slide.

- The list owns its entries
- Therefore, it can mutate them, even after their constructor finished

Hierarchies in OIGJ





Ownership hierarchy

`World` – anyone can access

`This` – **`this`** owns the object

Immutability hierarchy

`ReadOnly` – no modification

`Raw` – object under construction

OIGJ syntax: fields (1 of 2)

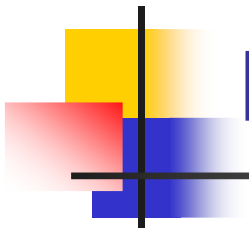
```
1:class Foo {
2:  // An immutable reference to an immutable date.
    @O @Immut Date imD = new @O @Immut Date ();
3:  // A mutable reference to a mutable date.
    @O @Mutable Date mutD = new @O @Mutable Date();
4:  // A readonly reference to any date. Both roD and imD cannot mutate
    // their referent, however the referent of roD might be mutated by an
    // alias, whereas the referent of imD is immutable.
    @O @ReadOnly Date roD = ... ? imD : mutD;
5:  // A date with the same owner and immutability as this
    @O @I Date sameD;
6:  // A date owned by this; it cannot leak.
    @This @I Date ownedD;
7:  // Anyone can access this date.
    @World @I Date publicD;
```

- Two annotations per type

OIGJ syntax: methods (2 of 2)

```
8 : // Can be called on any receiver; cannot mutate this.
    int readonlyMethod() @ReadOnly {...}
9 : // Can be called only on mutable receivers; can mutate this.
    void mutatingMethod() @Mutable {...}
10: // Constructor that can create (im)mutable objects.
    Foo(@O @I Date d) @Raw {
11:   this.sameD = d;
12:   this.ownedD = new @This @I Date ();
13:   // Illegal, because sameD came from the outside.
      // this.sameD.setTime(...);
14:   // OK, because Raw is transitive for owned fields.
      this.ownedD.setTime(...);
15: }
```

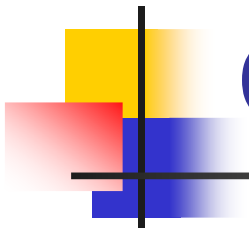
- Method receiver's annotation has a dual purpose:
  - Determines if the method is applicable.
  - Inside the method, the bound of **@I** is the annotation.



Formalization: Featherweight OIGJ

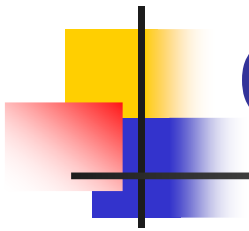
- Novel idea: **Cookers**
  - Every object o in the heap is of the form:

    `o→Foo<o',Mutable>` or `o→Foo<o',Immut`$_{o''}$`>`
  - **o'** is the owner of **o**
  - **o''** is the **cooker** of **o**, i.e., when the constructor of **o''** finishes then **o** becomes **cooked**
  - We keep track of the set of ongoing constructors
  - Subtyping rules connect cookers and owners
- Proved soundness and type preservation



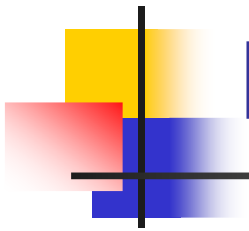
Case studies

- Implementation uses the checkers framework
  - Only 1600 lines of code (but still a prototype)
  - Requires type annotations available in Java 7
- Java's Collections case study
  - 77 classes, 33K lines of code
  - 85 ownership-related annotations
  - 46 immutability-related annotations



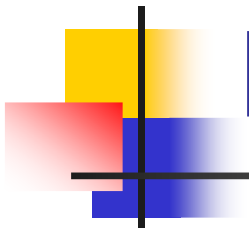
Case studies conclusions

- Verified that collections own their representation
- Method **`clone`** is problematic
  - **`clone`** makes a shallow copy that breaks ownership
  - Our suggestion: compiler-generated **`clone`** that nullifies fields, and then calls a copy-constructor



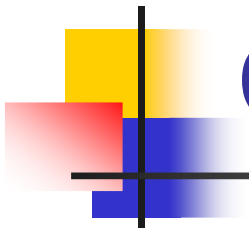
Previous Work

- Universes
  - Relaxed owner-as-**dominator** to owner-as-**modifier**
    - ReadOnly references can be freely shared
    - Constrains modification instead of aliasing, i.e., only the owner can modify an object
- Reference immutability:
  - C++'s **`const`**
  - Javari



Future work

- Inferring ownership and immutability annotations
- Bigger case study
- Extending OIGJ
  - owner-as-modifier
  - uniqueness or external uniqueness



Conclusions

- Ownership Immutability Generic Java (OIGJ)
  - Simple, intuitive, small
  - Static – no runtime penalties (like generics)
  - Backward compatible, no JVM changes
- Case study proving usefulness
- Formal proof of soundness
- Paper submitted to OOPSLA. Links:
  - **`http://ecs.victoria.ac.nz/twiki/pub/Main/TechnicalReportSeries/`**
  - **`http://code.google.com/p/checker-framework/`**
  - **`http://code.google.com/p/ownership-immutability/`**