# Glacier: Usable Enforcement of Transitive Immutability

Michael Coblenz, Whitney Nelson, **Jonathan Aldrich**,
Brad Myers, and Joshua Sunshine

17-396/17-696/17-960: Language Design and Prototyping

Carnegie Mellon University

institute for SOFTWARE RESEARCH

Carnegie Mellon University
School of Computer Science

# Motivation: vulnerability from Java 1.1.1

```
public class Class {
  private Object[] signers;

  public Object[] getSigners() {
    return signers;
  }
}
class Evil {
  public void evil() {
    getClass().getSigners()[0] = "com.google";
  }
}
```

Tracks which principals have signed the code represented by this class.

Returns the internal array used for storage

An attacker can mutate the array, allowing arbitrary code to be treated as trusted.

Note: example simplified for presentation purposes

# Patching the vulnerability: make a copy

```
public class Class {
    private Object[] signers;

    public Object[] getSigners() {
        return signers.clone();
    }
}
```

Tracks which principals have signed the code represented by this class.

Patches the vulnerability, but far from ideal – makes a costly copy on each call.

Note: example simplified for presentation purposes

# A better solution: immutability

```
public class Class {
    private @Immutable Object[] signers;

    public @Immutable Object[] getSigners() {
        return signers;
    }
}
```

Returns an immutable array – one that attackers cannot write to. No performance cost unless we need to change the list of signers (unlikely here).

A common problem:

- Provide access to read-mostly data
- Protect integrity

Note: example simplified for presentation purposes

# Immutability: not solved already?

- Java's **final**, C++'s **const** restrict assignment
  - But **const** is unsound, both are too weak to be useful
  - What properties to programmers actually need?
  - Can we enforce mathematical properties that provide value?

- Some languages, type systems have stronger semantics
  - Haskell (immutable by default), IGJ (immutability in Java)
    - These have not caught on
    - We found serious usability problems with IGJ (more later)
  - Can we leverage the science of usability to do better?

- Message: better immutability types can improve security, correctness
- Meta-message: significant benefit from combining type theory and usability science

# Many Design Decisions

- Immutability vs. read-only references
  - Can the data structure be changed through other pointers?
- Scope
  - Enforce immutability for all uses of a type, or case-by-case?
- Transitivity
  - Is this object immutable, or all reachable data?
- Initialization
  - Relax immutability during initialization?
- Abstraction
  - Protect only abstract state, e.g. allowing caching? Or all state?
- Polymorphism
  - Collections polymorphic over the immutability of contents?

# What do programmers need?

Semi-structured interviews with 8 experienced [mean 15 years] developers found:

- Significant usage of immutable or access-restricted APIs
- State change is a major source of bugs
  - Q: "Are bugs frequently caused by unintended state change?"
    A: "Oh God, most of them!
- Existing language constructs did not meet perceived needs
  - Viral nature of C++'s **const** caused usability problems
  - Need to protect an entire class from mutation
  - Guarantees too weak to be useful
- Takeaways: some evidence that:
  - Immutability matters to practitioners
  - Need better usability, stronger semantics than Java, C++

# What guarantees would help?

- Immutability > read only references
  - Read-only references restrict mutation only through one reference
    - Mutation through other references can still cause problems
  - **Immutability** means data that cannot be changed at all
    - Powerful mathematical properties: equational reasoning, guarantees no race conditions, prohibits an attacker from violating data integrity

- Transitive > non-transitive
  - **Transitive immutability** protects an entire reachable data structure from mutation
    - Lifts the guarantees provided by immutability to the units that matter architecturally

```
@Immutable Person p = …;
p.getAddress().setCity(city); // transitive immutability error
```

# Are existing research systems usable?

- Some research systems provide the guarantees we want. Are they usable enough?
- Pilot study with 3 programmers using the IGJ immutability type system [Zibin *et al.* 2007] showed difficulties:
  - Enforcing transitive immutability
  - Understanding error messages
- Root problems may include complexity, high syntactic overhead
  - Issues may be shared with other systems
    - C++: what is constant here?

```
int * const x
```

# Are current industrial systems usable?

Study of 10 developers carrying out immutability-related tasks using `final` in Java

**Results**

- 0/10 developers correctly expressed immutability
  - Even with a "cheat sheet" of steps recommended by Bloch
  - Too many details can go wrong, e.g. transitivity, defensive copies…

```
public class User { …
    final String[] authorizedFiles; // Files the user is authorized to access
    public User(…, String[] authorizedFiles) {
        // implement me
        this.authorizedFiles = authorizedFiles;
    }
```

# Specifying immutability in immutable designs

- With **final** (Bloch):
  - Don’t provide any methods that modify the object’s state.
  - Ensure that the class can’t be extended.
  - Make all fields **final**.
  - Make all fields **private**.
  - Ensure exclusive access to any mutable components.

# Are current industrial systems usable?

Study of 10 developers carrying out immutability-related tasks using **final** in Java

**Results**

- 0/10 developers correctly expressed immutability
- 7/10 developers implemented put() mutably for an immutable HashBucket

```
HashBucket put(Object k, Object v) {
   // replace or merge
   for (int i = 0; i < keys.length; i++) {
     if (k.equals(keys[i])) {
  →    values[i] = v;
       …
     }
   }
…
```

Based on a real bug in BaseX

# Are current industrial systems usable?

Study of 10 developers carrying out immutability-related tasks using **final** in Java

**Results**

- 0/10 developers correctly expressed immutability
- 7/10 developers implemented put() mutably for an immutable HashBucket
- 4/10 developers introduced a getSigners()-like vulnerability

```
public String[] getAuthorizedFiles() {
    // TODO; returning null is bogus
    return authorizedFiles;
}
```

# GLACIER

Great
Languages
Allow
Class
Immutability
Enforced
Readily

# Glacier: simple transitive immutability

- We set out to design a type system that is
  - **Simple** – to avoid the usability problems in earlier systems
  - **Strong** – enforcing transitive immutability
    - not just final fields or read-only references
  - **Sound** – always enforces the claimed mathematical properties

# A Glacier example

```
@Immutable class Person
{
        String name;
        Address address;
}

class Address { … }

Person p = …
p.name = "Alex"
```

Every Person instance is @Immutable

OK, String is @Immutable

Error: Address is not @Immutable

Error: name is (implicitly) final

# Glacier's Design Decisions

As simple as possible, given strong and sound semantics:

- Immutability vs. read-only references
  - Immutability [Strong semantics]
- Scope
  - Class immutability [Simplicity, usability]
- Transitivity
  - All reachable data is immutable [Strong semantics]
- Initialization
  - No relaxation [Simplicity]
- Abstraction
  - Protect all state, no exceptions for caching [Simplicity]
- Polymorphism
  - Not supported [Simplicity]

Is it too simple? Maybe, but we wanted **an existence proof for a usable, useful immutability type system**

# Informal evidence: simplicity reasonable

- Observation: most Java classes are naturally either mutable or immutable
  - Advice from Josh Bloch on making classes immutable

    "Classes should be immutable unless there's a very good reason to make them mutable."

  - Immutable collections libraries are designed differently

    add() returns a new collection, vs. side-effecting in a mutable library

- Suggests we might be able to live with class-level immutability, lack of polymorphism

# Glacier: simple transitive immutability

- Glacier is an annotation system and checker for Java
  - @Immutable marks a class immutable
  - All fields of an @Immutable object are **final** and must point to other @Immutable objects
  - Sound handling of inheritance, parametric polymorphism, arrays
    - @Immutability inherited
    - Type parameters of an @Immutable class must be @Immutable
    - @ReadOnly necessary for standard library treatment of arrays

# Theoretical Evaluation: is Glacier sound?

- Does @Immutable enforce transitive immutability?
  - Key design decisions based on (multiple) formal models of immutability type systems and proofs of soundness

$$\boxed{\Gamma \vdash s \dashv \Gamma'} \quad \boxed{\Gamma \vdash Seq \dashv \Gamma'} \quad \boxed{C \vdash M}$$

$$\frac{\begin{array}{cc}\texttt{fieldType}(\Gamma(x), f) = \tau & \texttt{freeze}(\Gamma(y)) <: \tau \\ \texttt{isImmutable}(\Gamma(x)) \Rightarrow \Gamma(x) \in T_l & \Gamma(y) \in T_l \Rightarrow \Gamma(x) \in T_l\end{array}}{\Gamma \vdash x.f = y \dashv \Gamma} \text{T-To-Field}$$

$$\frac{x \notin \texttt{dom}(\Gamma) \quad \texttt{fieldType}(\Gamma(y), f) = \tau \quad \Gamma(\mathrm{y}) \notin T_l}{\Gamma \vdash x = y.f \dashv \Gamma,\ x : \tau} \text{T-From-Field}$$

$$\frac{\begin{array}{c}x \notin \texttt{dom}(\Gamma) \\ \texttt{isImmutable}(C) \Rightarrow \tau = \mathsf{liquid}\ C \quad \neg\texttt{isImmutable}(C) \Rightarrow \tau = C\end{array}}{\Gamma \vdash \mathrm{x} = \mathsf{new}\ \mathrm{C} \dashv \Gamma, \mathrm{x} : \tau} \text{T-New}$$

$$\frac{\begin{array}{c}x \notin \texttt{dom}(\Gamma) \quad \texttt{methodLookup}(\Gamma(y), m) = \tau\ m(\overline{\tau\ x})\ Q \\ \Gamma(y) \in T_l \Leftrightarrow Q = \mathsf{liquid} \quad \forall i. \Gamma(z_i) <: \tau_i\end{array}}{\Gamma \vdash x = y.m(\overline{z}) \dashv \Gamma, x : \tau} \text{T-Method-Call}$$

**Theorem 1 (Soundness).** *For some program consisting of Seq and a set of class declarations* $\overline{CL}$*,*
*if* $\varnothing \vdash Seq \dashv \Gamma$ *and* $(\varnothing, \langle \varnothing, Seq \rangle \cdot \mathsf{top}) \to^* (\sigma, \mathrm{S})$*, then* $\texttt{wf}(\sigma, \mathrm{S})$.

# Empirical Evaluation

- A user study:
  - Usability: can people specify immutability with the system? Better than Java's `final`?
  - Usefulness: Does using Glacier prevent bugs and security vulnerabilities?
- Two case studies: is Glacier applicable to real-world projects?

# Participants (N=20)

- Mean programming experience: 9.5 years (range: 4-19 years)
- Mean Java experience: 3 years (range: 1-8 years)
- 90% had used **final** before
- Pre-test on **final**; mean score 3.45 correct (of 5)
  - 9 of 20 thought that it is forbidden to call setters on objects referenced by **final** fields
  - On reading **final** documentation: "I've only used **final** on integers before, so this will be instructive."

# User Study Methodology

| final (N=10) | Glacier (N=10) |
| --- | --- |
| Questionnaire | Questionnaire |
| 3 pages of documentation on final | 2-page paper tutorial |
| 2 annotation tasks | 2 annotation tasks |
| Instructions on immutability [Bloch] | |
| Revised annotation tasks | |
| 2 programming tasks | 2 programming tasks |

# Specifying immutability in immutable designs

- With **final** (Bloch):
  - Don't provide any methods that modify the object's state.
  - Ensure that the class can't be extended.
  - Make all fields **final**.
  - Make all fields **private**.
  - Ensure exclusive access to any mutable components.
- With Glacier:
  - Add @Immutable where required

# Evaluation: does Glacier help?

User experiment carrying out immutability-related tasks using **final** in Java vs. @Immutable in Glacier

**Results**

Ensuring Person, Accounts data structures are transitively immutable

| | final | Glacier |
|---|---|---|
| Correctly enforced immutability in class Person | 0/10 | 10/10 |
| Correctly enforced immutability in class Accounts | 0/10 | 9/10 |

# Evaluation: does Glacier help?

User experiment carrying out immutability-related tasks using **final** in Java vs. @Immutable in Glacier

**Results**

Implementing put() in an immutable Hashtable (based on a real bug in BaseX)

| | final | Glacier |
|---|---|---|
| Claimed task completion | 10/10 | 7/10 |
| Task correct (avoided mutating array in place) | 3/10 | 7/7 |

# Evaluation: does Glacier help?

User experiment carrying out immutability-related tasks using **final** in Java vs. @Immutable in Glacier

**Results**

Implementing pieces of a server with user accounts

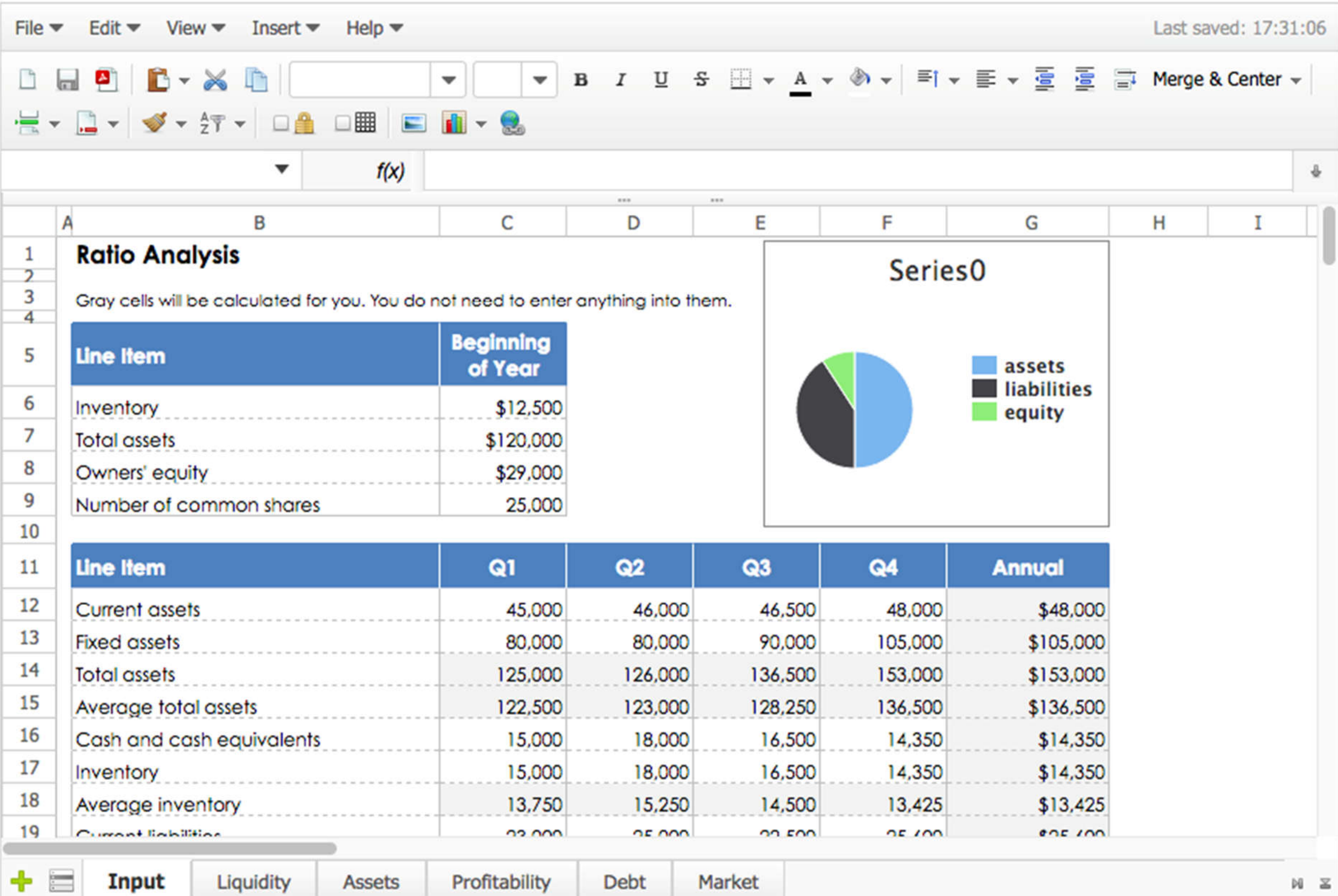
| | final | Glacier |
|---|---|---|
| Claimed task completion | 8/10 | 7/10 |
| Task correct (avoided security vulnerability) | 4/8 | 7/7 |

# Results, Limitations

- Glacier
  - enabled more users to finish tasks without bugs/vulnerabilities
  - only slightly decreased task completion

- Limitation: Small lab study
  - But if people insert bugs in small, simple projects, they are likely to in large, complex projects
- Limitation: Graduate student participants
  - But they had at least some experience in Java

# CASE STUDY 1: ZK SPREADSHEET




**Ratio Analysis**

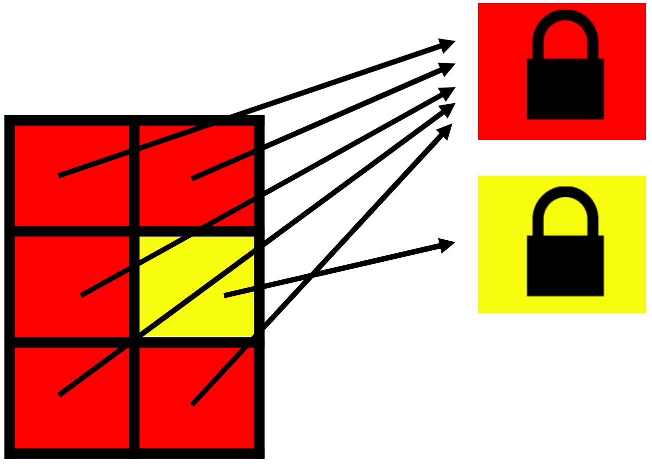
Gray cells will be calculated for you. You do not need to enter anything into them.

| Line Item | Beginning of Year |
|---|---|
| Inventory | $12,500 |
| Total assets | $120,000 |
| Owners' equity | $29,000 |
| Number of common shares | 25,000 |

| Line Item | Q1 | Q2 | Q3 | Q4 | Annual |
|---|---|---|---|---|---|
| Current assets | 45,000 | 46,000 | 46,500 | 48,000 | $48,000 |
| Fixed assets | 80,000 | 80,000 | 90,000 | 105,000 | $105,000 |
| Total assets | 125,000 | 126,000 | 136,500 | 153,000 | $153,000 |
| Average total assets | 122,500 | 123,000 | 128,250 | 136,500 | $136,500 |
| Cash and cash equivalents | 15,000 | 18,000 | 16,500 | 14,350 | $14,350 |
| Inventory | 15,000 | 18,000 | 16,500 | 14,350 | $14,350 |
| Average inventory | 13,750 | 15,250 | 14,500 | 13,425 | $13,425 |
| Current liabilities | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |

# Case study: ZK Spreadsheet

- Authors didn't use immutability (performance concerns)
- Refactored model (36 KLOC) to make cell styles immutable
- Updated calls in spreadsheet module (21 KLOC) to use modified model
- 20 person-hours
- Found two previously-unknown bugs



# Case study: Guava ImmutableList

- Goal: see how Glacier works in reusable library code
- Refactored:
  - @Immutable ImmutableList
  - @Immutable ImmutableCollection
  - and subclasses (as required)
- Success, but with some limitations
  - No polymorphism → one method duplicated
  - Could not leverage a cache used to convert collections to lists

# Future Work

- Can we add expressiveness while retaining usability?
  - Lazy initialization of caches
  - Allow mutation temporarily (circular data structures)
  - Polymorphism
- Which structures should be designed to be immutable? What is the current practice?

# Immutability Types – based on math and science

- Glacier is a new immutability type system for Java
  - **Simple** enough to be usable by programmers
  - **Soundly** enforces a strong mathematical property: **transitive immutability**
  - **Applies to real code** with little overhead and only minor code changes
  - **Helps users write correct code and prevent security vulnerabilities**
    - First user study on immutability!

- Glacier illustrates an effective approach to improving languages
  - Use **mathematical models** to ensure correctness and power of tools
  - Leverage **usability science** to ensure benefit from that power in practice

# Backup Slides

# Sample Errors

| Error | # users |
|---|---|
| Provided mutating methods | 0 |
| `Person` not `final` | 6 |
| `Address` not `final` | 10 |
| `Accounts` not `final` | 2 |
| `User` not `final` | 9 |
| Fields of `Person` not `final` | 2 |
| Fields of `Address` not `final` | 6 |
| `Accounts.users` not `final` | 1 |
| Fields of `User` not `final` | 4 |
| Fields of `Person` not `private` | 4 |
| Fields of `Address` not `private` | 8 |
| `Accounts.users` not `private` | 2 |
| Fields of `User` not `private` | 7 |
| Omitted copying `users` in `Accounts` constructor | 4 |
| Omitted copying `users` in `Accounts.getUsers()` | 2 |
| Omitted copying `authorizedFiles` in `User` constructor | 8 |