Disciplined locking: No more concurrency errors

http://CheckerFramework.org/

Werner Dietl, University of Waterloo
Michael Ernst, University of Washington

Concurrency: essential but error-prone

+ Essential for performance (exploit multiple cores)

+ Design component of GUIs

– Data races: concurrent access to shared data
  - easy mistake to make
  - leads to corrupted data structures
  - difficult to reproduce and diagnose

Thread-unsafe code

```
class BankAccount {

  int balance;

  void withdraw(int amount) {
    int oldBal = this.balance;
    int newBal = oldBal - amount;
    this.balance = newBal;
  }

  ...
```

Data race example

Shared account
Initial balance = 500

Thread 1:

sharedAccount.withdraw(50)

```
int oldBal = this 500 ance;
int newBal = o 500 l - ar 50 nt;
this.balance = n 450 l;
```

Thread 2:

sharedAccount.withdraw(100)

```
int oldBal = this 500 ance;
int newBal = o 500 l - a 100 nt;
this.balance = n 400 l;
```

Withdrawals = 150
Final balance = 450

Solution: locking

```
class BankAccount {

  Object acctLock;
  @GuardedBy(“acctLock”) int balance;

  void withdraw(int amount) {
   synchronized (acctLock) {
    int oldBal = this.balance;
    int newBal = oldBal - amount;
    this.balance = newBal;
   }
  }
```

Locking:

- Only one thread can acquire the lock
- No concurrent access to data
- Which lock to hold?

Key issues:

- Names vs. values
- Aliasing

Locking discipline = which locks to hold when accessing what data

```
@GuardedBy("lock1") int w;
@GuardedBy("lock2") int x;
@GuardedBy("lock2") int y;
                    int z;
```

- Write locking discipline as documentation and for use by tools
- `@GuardedBy` [Goetz 2006] is a de-facto standard
  - On GitHub, 35,000 uses in 7,000 files
- Its semantics is informal, ambiguous, and incorrect
  - It allows data races
- Similar problems with other definitions

Outline

- Formal semantics for locking disciplines
  - unambiguous
  - prevents data races (other concurrency errors exist)
  - two variants: value-based, name-based
- Two implementations:
  - type-checker that validates use of locking
  - inference tool that infers locking discipline
- Experiments: programmer-written `@GuardedBy`:
  - are often inconsistent with informal semantics
  - permit data races even when consistent

Concurrency background

Each object is associated with a *monitor* or *intrinsic lock*.

specification of locking discipline

synchronized statement or method locks the monitor.

Exiting the statement or method unlocks the monitor.

```
Date d = new Date();

@GuardedBy("d") List lst = ...;

synchronized (d) {
  lst.add(...)
  lst.remove(...)
  otherList = lst;
}
```

guard expression; arbitrary, e.g. **`a.b().f`**

Our implementations handle explicit locks too.

Defining a locking discipline

Informally:

"If program <u>element *x*</u> is annotated by @GuardedBy(<u>*L*</u>), a thread may only <u>use *x*</u> while holding the lock *L*."

```
MyObject lock;
@GuardedBy("lock.field") Pair shared;
@GuardedBy("lock.field") Pair alias;

synchronized (lock.field) {
  shared.a = 22;
  alias = shared;
}
```

Guard expression:
- Aliases? Yes
- Reassignment? No
- Side effects? Yes
- Scoping? Def site

Value protection answers

Element being guarded:
- Name or value? **Value**
- Aliases? Yes
- Reassignments? Yes
- Side effects? Yes

What is a use?
- Occurrence of name? ← current
- Dereference of name? (**`x.f`**)
- Dereference of value? ← **better**

(Dereference = field read/write)

```
MyObject lock;
@GuardedBy("lock") Pair shared;
Pair alias;
```

Name protection

… not value protection

```
synchronized (lock) {
  alias = shared;
}
alias.a = ...
```

Suffers a data race

Value protection

… not name protection

```
alias = shared;
synchronized (lock) {
  shared.a = ...
}
```

No data race

`alias.a = ...` is forbidden without holding **`lock`**

Locking discipline semantics providing value protection

type qualifier

```
@GuardedBy("lock") Pair shared;
```

type　　variable

Suppose expression *x* has type: @GuardedBy(*L*) C

A *use* is a dereference　　Type system constraint; may lock an alias

When the program dereferences a value that has ever been bound to *x*, the program holds the lock on the value of expression *L*.

The referent of *L* must not change while the thread holds the lock.

No reassignment of guard expression.

Side effects permitted (do not affect the monitor).

Locking discipline semantics providing name protection

```
@GuardedBy("lock") Pair shared;
```

variable annotation — type — variable

Suppose variable *v* is declared as @GuardedBy(*L*)

A *use* is a variable read or write — No aliasing permitted

When the program accesses *v*, which must not be aliased, the program holds the lock on the value of expression *L*.

L may only be "itself" or "this".

Guarantees L always evaluates to the same value

Sound name protection is restrictive

Demo

Key theoretical contributions

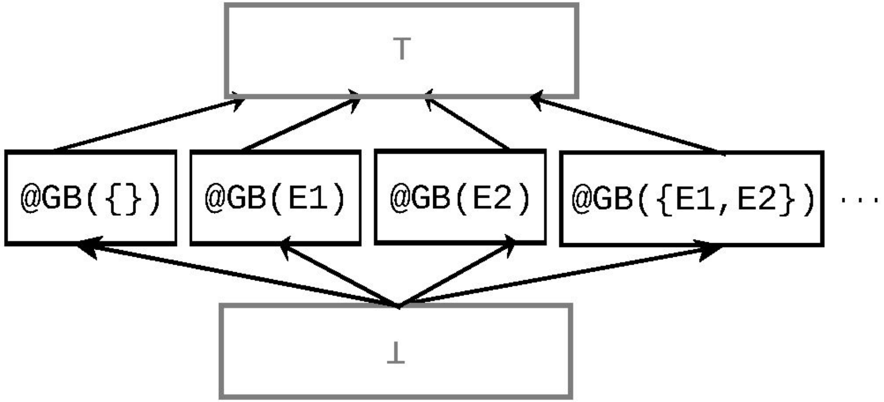
- Two formal semantics (name-based and value-based)
  - Core calculus based on RaceFreeJava [Abadi TOPLAS 2006]
  - Structural Operational Semantics
  - Definitions of accessed variables and dereferenced locations
- Proofs of correctness
  - By contradiction:
    - assume data race
    - show locking discipline must have been violated

Static analysis of a locking discipline

- Goal is to determine facts about values
  - Program is written in terms of facts about variables
- Analysis computes an approximation (an abstraction)
  - of values each expression may evaluate to
  - of locks currently held by the program

Both abstractions are sound

Enforcement of value semantics via type-checking



Type rule:
If x : `@GB`(*L*) ,
then *L* must be held
when x is dereferenced

- No two `@GuardedBy` annotations are related by subtyping
- Why not `@GB`(*L1*) <: `@GB`(*L1*, *L2*)?
  - Side effects and aliasing

More type system features

- Method pre/postconditions (`@Holding` annotations)
- Side effect annotations
- Type qualifier polymorphism
- Reflection
- Flow-sensitive type inference

Type-checking is a modular analysis

Modular = one procedure at a time

When type-checking a method, examine *only*:

- its body
- the signatures of its callees

Advantages: fast; clear error messages

Disadvantage: need library annotations

Is this code safe?

```
@GuardedBy("myMonitor") List<String> myList;
System.identityHashCode(myList);
```

Is this code safe?

```
@GuardedBy("myMonitor") List<String> myList;
System.identityHashCode(myList);

myList.clear();
```

clear() requires synchronization

```
@GuardedBy("myMonitor") List<String> myList;
System.identityHashCode(myList);
synchronized(myMonitor) {
  myList.clear();
}
```

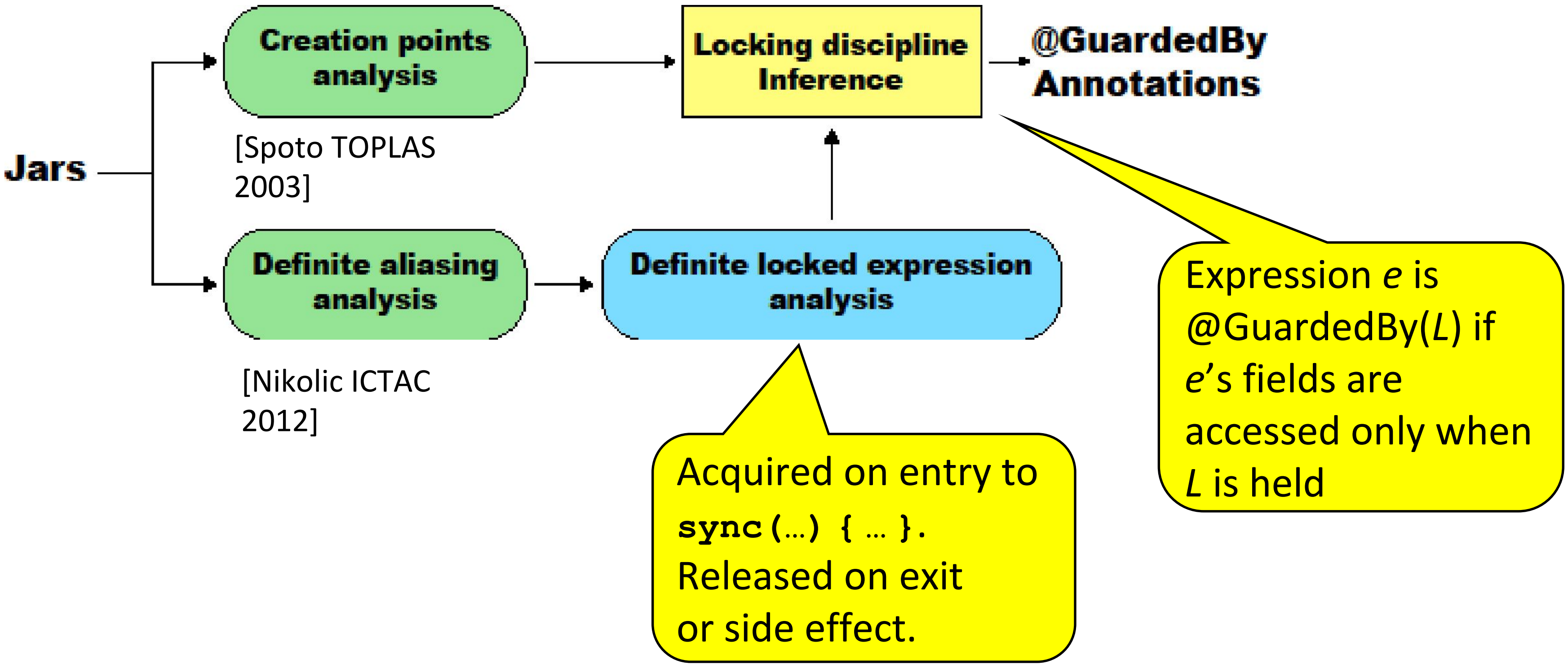
Library annotations = constraints on callers

```
@GuardedBy("myMonitor") List<String> myList;
System.identityHashCode(myList);
synchronized(myMonitor) {
  myList.clear();
}
```

```
class System {
  @ReleasesNoLocks
  static int
  identityHashCode(Object x);
}
```

```
interface List<E> {
  @ReleasesNoLocks
  boolean
  clear(@GuardSatisfied List<E> this);
}
```

Inference of both semantics via abstract interpretation



Inference implementation

1. Where is the guarded element used?
   - Name protection: syntactic uses of variable
   - Value protection: estimate via creation points analysis
2. What expressions are locked at those points?
   - Definite aliasing analysis
   - Side effect analysis
   - Viewpoint adaptation (contextualization)

Whole-program analysis

- Makes closed-world assumption
- Type-checking is modular, incremental

Experimental evaluation of value semantics

- 15 programs, 1.3 MLOC
  - BitcoinJ, Daikon, Derby, Eclipse, Guava, Jetty, Velicity, Zookeeper, Tomcat, …
  - 5 contain programmer-written **`@GuardedBy`** annotations
- 661 correct annotations
  - Candidates: annotations written by the programmer or inferred by our tool
  - Correct: program never suffers a data race on the element
    - Determined by manual analysis

Experimental results

Are the annotations correct?

Are the annotations complete?

	Precision	Recall
Inference	100%	83%
Type-checking	100%	99%
Programmers	50%	42%

Tell the whole truth and nothing but the truth.

Programmer mistakes

Errors in every program that programmers annotated
with respect to both value and name semantics

- Wrong lock
  - Guava: `@GuardedBy("Segment.this")` should be `@GuardedBy("this")`
- Lock writes but not reads
  - Guava: `SerializingExecutor.isThreadScheduled`
- Omitted annotations
  - BitcoinJ: `PaymentChannelServer` ; Apache Velocity: several
- Creating external aliases
  - BitcoinJ: `PaymentChannelClient.conn`

Implementations

- Type checker:
  - Lock Checker, distributed with the Checker Framework
  - http://CheckerFramework.org/
  - Live demo: http://eisop.uwaterloo.ca/live
- Inference:
  - Julia abstract interpretation
  - http://juliasoft.com/

The Checker Framework

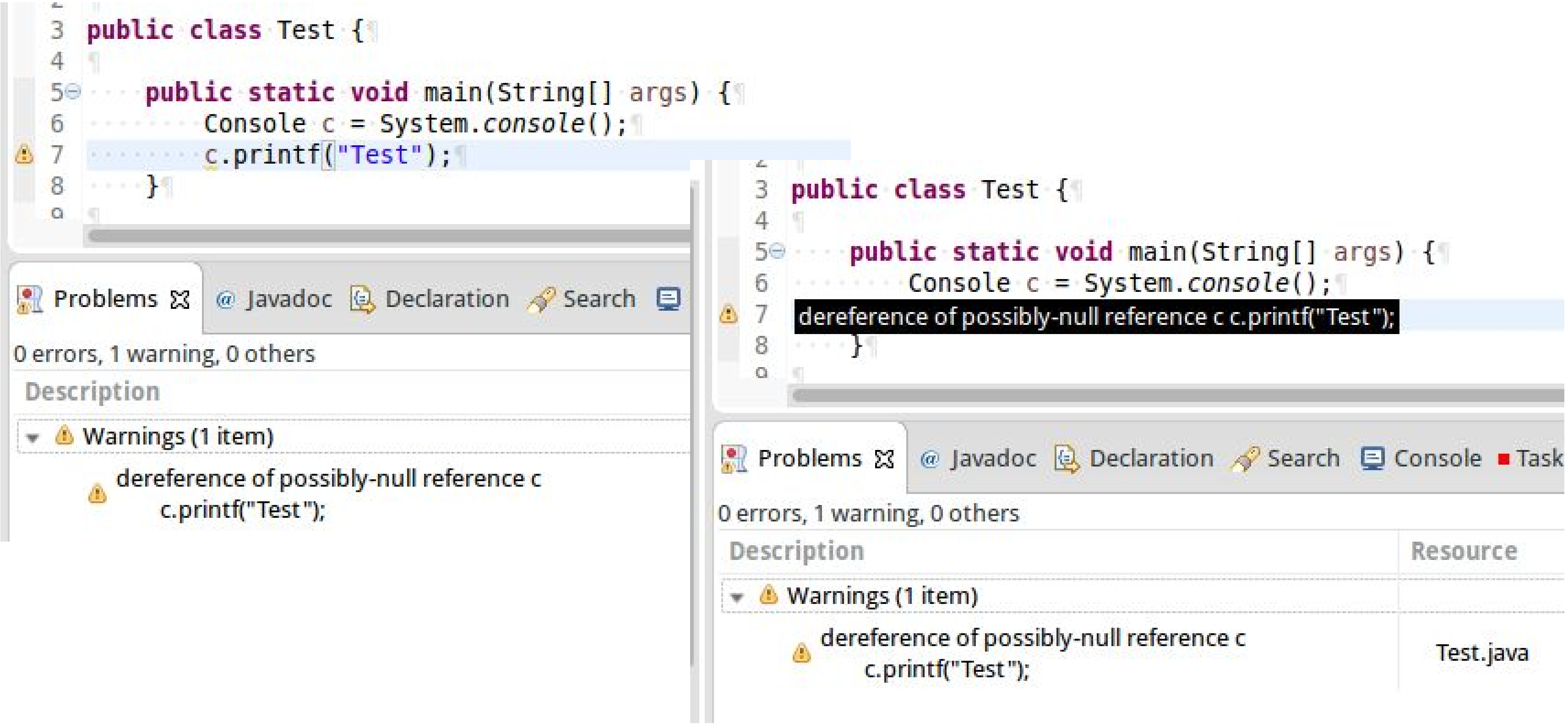
A framework for pluggable type checkers

“Plugs” into the OpenJDK or OracleJDK compiler

```
javac -processor MyChecker …
```

Standard error format allows tool integration

Eclipse plug-in



Ant and Maven integration

```
<presetdef name="jsr308.javac">
  <javac fork="yes"
    executable="${checkerframework}/checker/bin/${cfJavac}" >
    <!-- JSR-308-related compiler arguments -->
    <compilerarg value="-version"/>
    <compilerarg value="-implicit:class"/>
  </javac>
</presetdef>
```

```
<dependencies>
  ... existing <dependency> items ...
  <!-- annotations from the Checker Framework:
         nullness, interning, locking, ... -->
    <dependency>
      <groupId>org.checkerframework</groupId>
      <artifactId>checker-qual</artifactId>
      <version>1.9.7</version>
    </dependency>
</dependencies>
```

Live demo `http://eisop.uwaterloo.ca/live/`

Checker Framework Live Demo

Write Java code here:

```java
import org.checkerframework.checker.nullness.qual.Nullable;
class YourClassNameHere {
    void foo(Object nn, @Nullable Object nbl) {
        nn.toString(); // OK
        nbl.toString(); // Error
    }
}
```

Choose a type system: Nullness Checker ▼

Check

Examples:

Nullness: NullnessExample | NullnessExampleWithWarnings

MapKey: MapKeyExampleWithWarnings

Interning: InterningExample | InterningExampleWithWarnings

Lock: GuardedByExampleWithWarnings | HoldingExampleWithWarnings | EnsuresLockHeldExample | Locl

Example type systems

Null dereferences (`@NonNull`)

>200 errors in Google Collections, javac, ...

Equality tests (`@Interned`)

>200 problems in Xerces, Lucene, ...

Concurrency / locking (`@GuardedBy`)

>500 errors in BitcoinJ, Derby, Guava, Tomcat, ...

Fake enumerations / typedefs (`@Fenum`)

problems in Swing, JabRef

Checkers are usable

- Type-checking is familiar to programmers
- Modular: fast, incremental, partial programs
- Annotations are not too verbose
  - **`@NonNull`**: 1 per 75 lines
  - **`@Interned`**: 124 annotations in 220 KLOC revealed 11 bugs
  - **`@Format`**: 107 annotations in 2.8 MLOC revealed 104 bugs
  - Possible to annotate part of program
  - Fewer annotations in new code
- Few false positives
- First-year CS majors preferred using checkers to not
- **Practical**: in daily use at Google, on Wall Street, etc.

Related work

- Name-based semantics: JML, JCIP, rccjava [Abadi TOPLAS 2006], …
- Heuristic checking tools: Warlock, ESC/Modula-3, ESC/Java
- Unsound inference: [Naik PLDI 2006] uses may-alias, [Rose CSJP 2004] is dynamic
- Sound inference for part of Java [Flanagan SAS 2004]
- Type-and-effect type systems: heavier-weight, detects deadlocks too
- Ownership types

Learning more

Using Type Annotations to Improve Your Code

BoF3427, tonight, 19:00 to 19:45

Continental Ballroom 4

Technical papers:

- “Locking discipline inference and checking”, ICSE 2016
- “Semantics for locking specifications”, NFM 2016

Try the Lock Checker: http://CheckerFramework.org/

How to prevent (some) concurrency errors

- Formal semantics for locking disciplines
  - unambiguous, prevents data races
- Experiments: programmer-written @GuardedBy:
  - are often inconsistent with informal semantics
  - permit data races even when consistent with informal semantics
- Type-checker validates use of locking discipline (@GuardedBy)
  - Download from http://CheckerFramework.org/