Locking discipline inference and checking

Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, Javier Thaine

University of Washington, USA
Università di Verona, Italy
Julia Srl, Italy

ICSE 2016

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;
  int balance;
  @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 aquire 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 (allows data races)
- Similar problems with other definitions

Contributions

- Formal semantics for locking disciplines
  - value-based
  - unambiguous
  - prevents data races
- 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.

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

Exiting the statement or method unlocks the monitor.

```
Date d = new Date();

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

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

Our implementations handle explicit locks too

Defining a locking discipline

Informally:
“If program element *x* is annotated by @GuardedBy(*L*),
a thread may only use *x*
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

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? **← our def**

```
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

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

No data race

Locking discipline semantics providing value protection

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

A *use* is a dereference

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.

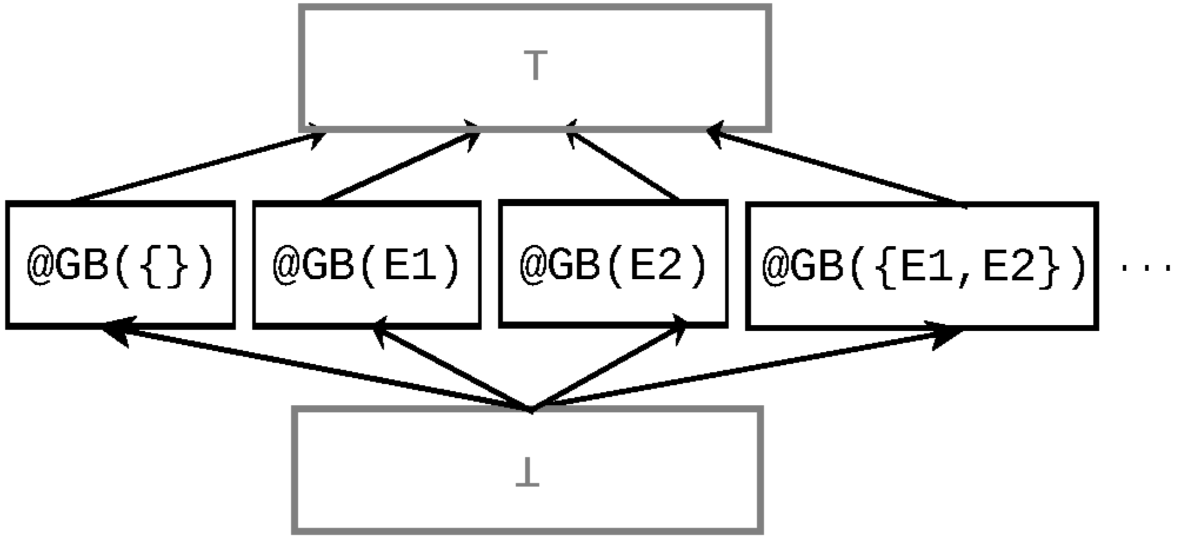
No reassignment of guard expression.
Side effects permitted (do not affect the monitor).

Formal semantics + proof of correctness [Ernst NFM 2016]

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 via type-checking



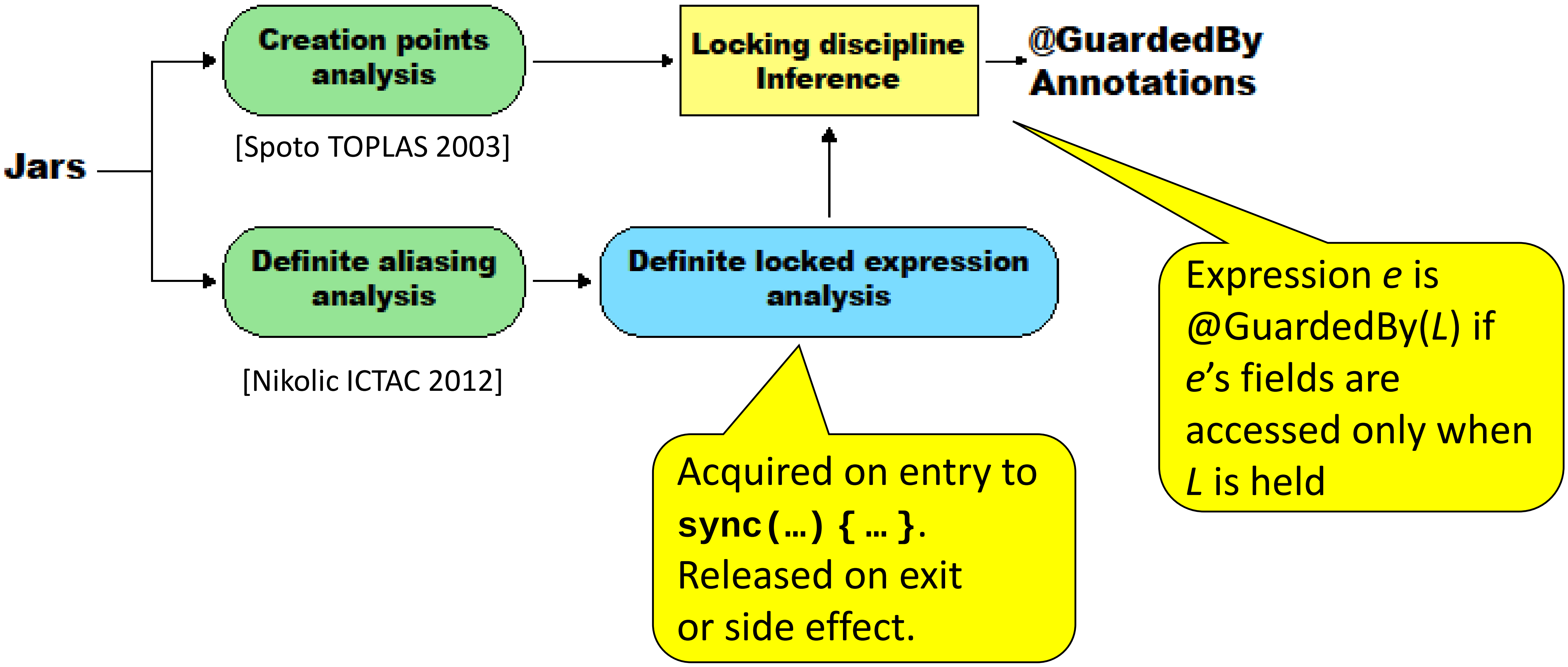
- No two **`@GuardedBy`** annotations are related by subtyping
- Why not **@GB**(*L1*) <: **@GB**(*L1*, *L2*)?
  - Side effects and aliasing

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

Type system also supports

- method pre/postconditions (**`@Holding`** annotations)
- side effect annotations
- type qualifier polymorphism
- reflection
- flow-sensitive type inference

Inference via abstract interpretation



Jars

Creation points analysis

[Spoto TOPLAS 2003]

Definite aliasing analysis

[Nikolic ICTAC 2012]

Definite locked expression analysis

Locking discipline Inference

@GuardedBy Annotations

Acquired on entry to **sync (…) { … }**. Released on exit or side effect.

Expression *e* is @GuardedBy(*L*) if *e*’s fields are accessed only when *L* is held

Experimental evaluation

- 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 (manual analysis)
- Results:
  - Inference: precision 100%, recall 83%
  - Type-checking: precision 100%, recall 99%
  - Programmers: precision 50%, recall 42%

Programmer mistakes

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

- Creating external aliases
- Lock writes but not reads
- Syntax errors
- Omitted annotations

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/

Contributions

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

Related work

- Name-based semantics: JML, JCIP, many others
- 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, detect deadlocks too
- Ownership types