Ownership Types for Safe Programming: Preventing Data Races and Deadlocks

Chandrasekhar Boyapati
Robert Lee
Martin Rinard

presented by Matthias Niklaus

ACM (OOPSLA) November 2002
Problem

- Problem
  - multi-threaded program
  - hard to debug
  - not reproducible faults

- Goals
  - Safety properties
    - No data races
  - Liveness properties
    - No deadlock
Two extreme approaches:

**Safety-first**
- Characteristic of topdown OO Design
- Can result in slow, deadlock-prone code

**Liveness-first**
- Characteristic of multithreaded systems programming
- Can result in buggy code full of races
Outline

- Safety-Property
  - Preventing data races
  - Ownership types
- Liveness property
  - Preventing deadlocks
- Extensions and other properties
  - Soundness, completeness
  - Type inference
  - Tree based partial order
  - Condition variables
- Experience
Data Races

Thread 1:

```java
o.b = o.b + 4;
```

Thread 2:

```java
o.b = o.b - 2;
```

Object `o`

- `a = 0`
- `b = 2`
- `c = 4`

What is the result?
How To Avoid Data Races?

Thread 1:

Object o

- a = 0
- b = 2
- c = 4

Thread 2:

- o.b = o.b - 2;

What is the result?
- T1: 2 + 4 => 6, 6 - 2 => 4
- T2: 2 - 2 => 0, 0 + 4 => 4

Are these all the possible solutions?
How To Avoid Data Races?

Thread 1:

Object o

- $a = 0$
- $b = 2$
- $c = 4$

Thread 2:

- $o.b = o.b - 2$

What is the result?

- $T1 \, 2 + 4 \Rightarrow 6, \, T2 \, 6 - 2 \Rightarrow 4$
- $T2 \, 2 - 2 \Rightarrow 0, \, T1 \, 0 + 4 \Rightarrow 4$

No But that should be all!

- $T1 | T2 \, 2 + 4 \Rightarrow 6, \, 2 - 2 \Rightarrow 0$

These results are not legal!
How To Avoid Data Races?

Thread 1:
```java
lock(o);
o.b = o.b + 4;
unlock(o);
```

Thread 2:
```java
lock(o);
o.b = o.b - 2;
unlock(o);
```

Now the result is 4.

**mutual exclusion**

Do we have to lock every object before accessing?
Mutual Exclusion

- Mutual exclusion lock

- No synchronisation needed if:
  - Immutable object
  - Accessible to single thread: `thisThread`
  - Unique pointer to object
Object Ownership

Key Idea

- Every object is protected by its root owner
- For race-free access to an object
  - A thread must lock its root owner
- A thread always holds the lock on thisThread

---

Diagram:

- ThisThread
  - Thread1 object
  - Thread2 object
  - Potentially shared objects
Ownership Properties

O1. The owner of an object does not change over time.

O2. The ownership relation forms a forest of rooted trees, where the roots can have self loops.

O3. The necessary and sufficient condition for a thread to access to an object is that the thread must hold the lock on the root of the ownership tree that the object belongs to.

O4. Every thread implicitly holds the lock on the corresponding thisThread owner. A thread can therefore access any object owned by its corresponding thisThread owner without any synchronization.
A Data Race Free Program

class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head = null;
    T.Owner> pop() access(this) {
        if (head == null) return null;
        T.Owner> value = head.value();
        head = head.next();
        return value;
    }
    ...
}

class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T.Owner> value;
    T.Owner> value() access(this) {
        return value;
    }
    TNode<nodeOwner, TOwner> next() {
        access(this) {
            return next;
        }
    }
    ...
}

class T<Owner> {
}

TStack<thisThread, thisThread> s1;
TStack<thisThread, world> s2;
TStack<world, world> s3;
A Data Race Free Program

```
s1 (TStack)       s2 (TStack)
  s1.head (TNode)  s1.head.next (TNode)
     s1.head.value (T)    s1.head.next.next.value (T)
  s1.head.next (TNode)  ``
Deadlock

1. mutual exclusion: at least one resource must be held in a non-shareable mode.

2. hold and wait: there must be a process holding one resource and waiting for another.

3. no preemption: resources cannot be preempted.

4. circular wait: there must exist a set of processes \([p_1, p_2, ..., p_n]\) such that \(p_1\) is waiting for \(p_2\), \(p_2\) for \(p_3\), and so on and \(p_n\) waits for \(p_1\)....
Deadlock scenario

- Cycle of the form
  - Thread 2 holds Lock 3 and waits for Lock 2
  - Thread 1 holds Lock 2 and waits for Lock 1...
  - Thread n holds Lock 1 and waits for Lock n
Avoiding deadlocks

- Impose a partial order among locks
- Acquire locks in descending order
- The type checker statically verifies the order
Lock Level Properties

L1. The lock levels form a partial order.

L2. Objects that own themselves are locks. Every lock belongs to some lock level. The lock level of a lock does not change over time.

L3. The necessary and sufficient condition for a thread to acquire a new lock \( l \) is that the levels of all the locks that the thread currently holds are greater than the level of \( l \).

L4. A thread may also acquire a lock that it already holds. The lock acquire operation is redundant in that case.
Rules

- $P$: the program being checked
- $E$: environment providing types for the FV(e)
- $ls$: set of locks held before $e$ is evaluated
- $l_{\text{min}}$: the minimum lock level of all locks in $ls$
- $t$: type of $e$

$P;E;ls;l_{\text{min}} \vdash e: t$

judgement $\rightarrow e$ evaluates to type $t$
under provided environment
Rule: EXP_SYNC

Rule for acquiring an new lock

[EXP SYNC]

\[ P; E \vdash_{\text{final}} e_1 : \text{cn}'(\text{self:cn.l} \ldots) \quad P \vdash \text{cn.l} < l_{\text{min}} \]

\((E = E_1, \text{locks(... l), } E_2) \implies (P; E \vdash \text{cn.l} < \text{level}(l)) \lor (l = e_1)\]

\[ P; E; l_s, e_1; \text{cn.l} \vdash e_2 : t_2 \]

\[ P; E; l_s; l_{\text{min}} \vdash \text{synchronized } e_1 \text{ in } e_2 : t_2 \]
Rule: EXP_REF

Rule for accessing a field

\[ [\text{EXP REF}] \]

\[ P; E; ls; l_{\text{min}} \vdash e : cn\langle o_1..n \rangle \quad P; E \vdash \text{RootOwner}(e) = r \]

\[ (P \vdash (t \textit{ fd}) \in cn\langle f_1..n \rangle) \wedge (r \in ls) \]
\[ \vee (P \vdash (\text{final} \ t \textit{ fd}) \in cn\langle f_1..n \rangle) \]

\[ P; E; ls; l_{\text{min}} \vdash e.\textit{fd} : t[e/\text{this}][o_1/f_1]..[o_n/f_n] \]
A Deadlock free Program

```java
1  class Vector<world:Vector.l, elementOwner> { 
2     LockLevel l = new;
3     int elementCount = 0;
4     ...
5     int size() locks (this) {
6         synchronized (this) {
7             return elementCount;
8         }
9     }
10 
11     boolean isEmpty() locks (this) {
12         synchronized (this) {
13             return (size() == 0);
14         }
15     }
```
Some Properties

- Soundness of the type System
- Completeness
- Runtime overhead
- Type inference
  - Intra-procedural type inference
  - Default types
- Lock level polymorphism
- Tree-based partial orders
- Condition variables
Can prevent deadlock in transactional methods via resource-ordering based on Java hash codes (among other solutions)

class Cell {
    long value;

    void swapValue(Cell other) {
        if (other == this) return; // alias check
        Cell fst = this; // order via hash codes
        Cell snd = other;
        if (fst.hashCode() > snd.hashCode()) {
            fst = other; snd = this;
        }
        synchronized(fst) {
            synchronized (snd) {
                long t = fst.value;
                fst.value = snd.value;
                snd.value = t;
            }
        }
    }
}
Some Properties

- Soundness of the type System
- Completeness
- Runtime overhead
- Type inference
  - Intra-procedural type interference
  - Default types
- Lock level polymorphism
- Tree-based partial orders
- Condition variables
Rule: EXP_WAIT

Rule for condition variable

[EXP WAIT]

\[
E = E_1, \text{locks}(\infty [e]_{opt}), E_2 \\
P; E \vdash_{\text{final}} e \quad ls = \{e\} \\
P; E; ls; l_{\text{min}} \vdash e.\text{wait} : \text{int}
\]

[EXP NOTIFY]

\[
P; E \vdash_{\text{final}} e \quad e \in ls \\
P; E; ls; l_{\text{min}} \vdash e.\text{notify} : \text{int}
\]
## Experience with annotations

<table>
<thead>
<tr>
<th>Program</th>
<th># Lines of Code</th>
<th># Lines annotated</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMTP Server (Apache)</td>
<td>2105</td>
<td>46</td>
</tr>
<tr>
<td>POP3 Mail Server (Apache)</td>
<td>1364</td>
<td>31</td>
</tr>
<tr>
<td>Discrete Event Simulator</td>
<td>523</td>
<td>15</td>
</tr>
<tr>
<td>HTTP Server</td>
<td>563</td>
<td>26</td>
</tr>
<tr>
<td>Chat Server</td>
<td>308</td>
<td>22</td>
</tr>
<tr>
<td>Stock Quote Server</td>
<td>242</td>
<td>12</td>
</tr>
<tr>
<td>Game Server</td>
<td>87</td>
<td>11</td>
</tr>
<tr>
<td>Database Server</td>
<td>302</td>
<td>10</td>
</tr>
</tbody>
</table>
Conclusions

- The proposed approach eliminates important errors in concurrent programs
  - Data races
  - Deadlocks
- Checking is fast and scalable.
- Requires *some* programming overhead.

Comment

- Everyone is able to write a race free and live program.
- The type system is very complex.
- You always have to specify a static order among the objects.