Universe Types for Race Safety

Dave Cunningham
(PhD student of Susan Eisenbach, Sophia Drossopoulou)
Imperial College London

VAMP
03/09/2007
Race Conditions

Race conditions:

▶ are bugs in shared memory concurrent software.
▶ are caused by incorrect synchronisation.
▶ are hard to reproduce.
▶ can corrupt program state.
▶ can lead to strange program behaviour.

Testing hard... ⇒
Race Conditions

Race conditions:

- are bugs in shared memory concurrent software.
- are caused by incorrect synchronisation.
- are hard to reproduce.
- can corrupt program state.
- can lead to strange program behaviour.

Testing hard... $\implies$ Static type system?
Race Conditions

Race conditions:

- are bugs in shared memory concurrent software.
- are caused by incorrect synchronisation.
- are hard to reproduce.
- can corrupt program state.
- can lead to strange program behaviour.

Testing hard... ⇒ Static type system?

Can prove it correct...
A state where two threads can access the same object:

We prove such states never arise during execution.
Object Accesses and Locks

If we know that:

► No two threads simultaneously hold the same lock.
► Threads only access objects for which they hold the lock.

Then: Threads will never simultaneously access an object.
Enforcing synchronisation is the key:

```java
sync (e') {
    ...
    e.f = 10;
    ...
}
```
General Approach

Enforcing synchronisation is the key:

```
sync (e’) {
    ...
    e.f = 10;
    ...
}
```

Require that $e’$ is **guarded by** the same lock $l$:

$$
\not\vdash_{gb} e’ : l
\not\vdash_{gb} e : l
$$
Example

\[
\begin{align*}
\emptyset & \vdash y \\
\{1\} & \vdash y \\
\{1\} & \vdash y.f = 10 \\
\{1\} & \vdash \ldots \ y.f = 10 \ldots \\
\emptyset & \vdash \text{sync} (x) \{\ldots \ y.f = 10 \ldots \}
\end{align*}
\]
Type System (for illustrative purposes only!)

\[ \emptyset \vdash \text{this} \quad \text{(Var)} \]
\[ \vdash_{gb} e : l \]
\[ l \in \mathbb{L} \]
\[ \mathbb{L} \vdash e.f \]

\[ \mathbb{L}' \vdash e \quad \text{(Sub)} \]
\[ \mathbb{L}' \subseteq \mathbb{L} \]
\[ \mathbb{L} \vdash e \]

\[ \mathbb{L} \vdash e' \quad \text{(Field)} \]
\[ \vdash_{gb} e' : l \]
\[ \mathbb{L} \cup \{1\} \vdash e \]
\[ \mathbb{L} \vdash \text{sync } e' \quad e \]

\[ \mathbb{L} \vdash e' \quad \text{(Sync)} \]
\[ \vdash_{gb} e' : l \]
\[ \mathbb{L} \vdash \text{sync } e' \quad e \]

Now we need only define $\vdash_{gb}$ (the hard bit).
A first attempt at defining $\vdash_{gb}$

Paths are sequences of field accesses starting from a variable e.g.

- `x.f.g`
- `this.first.next.next`

We use them to statically characterise objects.
A first attempt at defining $\vdash_{gb}$

Paths are sequences of field accesses starting from a variable e.g.

- $x.f.g$
- `this.first.next.next`

We use them to statically characterise objects.

If we let $\vdash_{gb} p : p$
(i.e. the set of all locks $\equiv$ the set of all paths)
A first attempt at defining $\vdash_{gb}$

Paths are sequences of field accesses starting from a variable e.g.

- $x.f.g$
- `this.first.next.next`

We use them to statically characterise objects.

If we let $\vdash_{gb} p : p$
(i.e. the set of all locks $=$ the set of all paths)

Then we allow: `sync (p) { ... p.f=20 }`
Derivation tree with paths

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]

\[
\begin{align*}
&\hline
&\null \quad \hline
&\hline
&\hline
&\hline
&\hline
\end{align*}
\]
A problem

\[ \emptyset \vdash \text{sync (x) \{ x=y ; x.f=20 \} } \]
A problem

\[
\emptyset \vdash \text{sync (x) } \{ \ x = y \ ; \ x.f = 20 \ \} \\
\uparrow \text{ accesses the object } y
\]
A problem

∅ ⊢ sync (x) \{ x=y ; x.f=20 \}
\uparrow \text{ accesses the object } y

similarly...

\{x\} ⊢ sync (x.f) \{ x.f=y ; x.f.g=20 \}
\uparrow \text{ accesses the object } y
A problem

\[
\emptyset \vdash \text{sync } (x) \{ \ x = y \ ; \ x.f = 20 \ \}
\]
\[\uparrow \text{accesses the object } y\]

similarly...

\[
\{x\} \vdash \text{sync } (x.f) \{ \ x.f = y \ ; \ x.f.g = 20 \ \}
\]
\[\uparrow \text{accesses the object } y\]

Solution – restrict such assignments.

How does this affect expressiveness?
Iteration

class Node { Node next; int cargo }

Node i = ...;
sync(i) {
    while (i!=null) {
        i.cargo = 20;
        i = i.next;
    }
}
class Node { Node next; int cargo }

Node i = ...;
sync(i) {
    while (i!=null) {
        i.cargo = 20;
        i = i.next;
    }
}

Here, assigning to i conflicts with the locking of i
What does this tell us?

This demonstrates that:

- 1:1 locking ($\vdash_{gb} p : p$) is unfeasable.
- E.g. many nodes should be guarded by 1 lock.
- This allows granularity control, and iteration.
- ($\vdash_{gb}$) is now a many-to-1 relationship:
  - $\vdash_{gb} i : l$
  - $\vdash_{gb} i.next : l$
  - $\vdash_{gb} i.next.next : l$
- Need to be careful with assignment.
Carving the Heap

Artist’s impression of a heap:
Carving the Heap

Artist’s impression of a heap:
Other work has used a programmer-supplied set, e.g. \{\texttt{RED}, \texttt{BLUE}\}

The source code looks like:

```java
RED Object r = new RED Object();
BLUE Object b = new BLUE Object();

r = b; //not allowed

void m(RED Object x, RED Object y) {
    x = y
}

m(r,b); //not allowed
```
Regions as Locks

Suppose we already have a region type system:

$$\Gamma \vdash e : R$$
Suppose we already have a region type system:

\[ \Gamma \vdash e : R \]

\[ \Gamma \vdash_{gb} e : R \]
Suppose we already have a region type system:

\[ \Gamma \vdash e : R \]

Note we now need a \( \Gamma \) in the race type system too:

\[ \mathbb{L}, \Gamma \vdash e : F \]
Regions as Locks

Suppose we already have a region type system:

\[
\begin{align*}
\Gamma & \vdash e : R \\
\Gamma & \vdash_{gb} e : R \\
\end{align*}
\]

Note we now need a $\Gamma$ in the race type system too:

$\lll, \Gamma \vdash e : F$

RED Object $r_1, r_2 = \ldots$
BLUE Object $b = \ldots$

\[
\text{sync}(r_1) \{
    \quad b.f = 10; // not allowed \\
    \quad r_2.f = 10; // OK \\
\}
\]
Iteration Example

```java
class Node { RED Node next; int cargo }

RED Node i = ...;

sync (i) {
    while (i!=null) {
        i.cargo = 20;
        i = i.next;
    }
}
```
Carving up the heap helps us verify safe locking:

- `x.f = y ; x.f.g = 10`  
  (must lock `l` where `Γ ⊢ gb y : l`)
Summary

Carving up the heap helps us verify safe locking:

- \( x.f = y ; x.f.g = 10 \)  
  (must lock \( l \) where \( \Gamma \vdash_{gb} y : l \))
- Regions restrict assignment only where the lock changes.
- \( x.f = y \) ensures \( \Gamma \vdash_{gb} x.f : l \)
Summary 2

Advantages of carving with regions:
- Simple
- Inference is easy

Disadvantages of regions:
- Lock count does not scale with object count

Regions used by:
- **Guava** – D. Bacon, R. Strom, A. Tarafdar (OOPSLA’00)
- **Sync... with data** – M. Vaziri, F. Tip, J. Dolby (POPL’06)
- **Locksmith** – P. Pratikakis, J. Foster, M. Hicks (PLDI’06)
Ownership types impose a heap hierarchy:
Ownership types impose a heap hierarchy:

Can use the “owner” of an object as its lock.
Universes form this hierarchy with 3 keywords

The keywords indicate the relative position of the referenced object.
class C {
    peer Object m(peer Object x) {
        peer Object y = new peer Object();
        rep Object z = new rep Object();
        x = y;
        x = z;  // not allowed
        any Object a = z;
        z = a;  // not allowed
        return y;
    }
}

rep Object o = new rep C().m(new rep Object());
Background of Universes

Universes

- are an ownership type system (see Peter Müller’s thesis).
- have the `any` type (unique to universes).
- are simple.
- are used in the JML (verification) tools.
Synchronisation

Let’s assume have a sound universe type system $\Gamma \vdash e : u$

(\text{where } u \in \{\text{rep}, \text{peer}, \text{any}\})

We can use this to define:

$$
\frac{
\Gamma \vdash e : u
}{
\Gamma \vdash_{gb} e : u
}
$$

peer Object x = new peer Object();
peer Object y = new peer Object();
rep Object z = new rep Object();
sync (x)
{
y.f = 20
}
// OK
sync (x)
{
z.f = 20
}
// error!
Synchronisation

Let’s assume have a sound universe type system $\Gamma \vdash e : u$

(\text{where } u \in \{\text{rep, peer, any}\})

We can use this to define: $\frac{\Gamma \vdash e : u}{\Gamma \vdash_{gb} e : u}$

```
peer Object x = new peer Object();
peer Object y = new peer Object();
rep Object z = new rep Object();
sync (x) { y.f = 20 } // OK
sync (x) { z.f = 20 } // error!
```
class Node {
    peer Node next ;
    int cargo
}

rep Node i = ...;

csync (i) {
    while (i!=null) {
        i.cargo = 20;
        i = i.next;
    }
}
Problem with any

**Problem:**

```java
any Object x = new peer Object();
any Object z = new rep Object();
sync (x) { z.f = 20 } // OK, but race condition!
```
Problem with any

Problem:

```
any Object x = new peer Object();
any Object z = new rep Object();
sync (x) { z.f = 20 } // OK, but race condition!
```

Solution:

```
Γ ⊢ e : u
u ≠ any
Γ ⊢_{gb} e : u
```
Problem with any

Problem:

```java
any Object x = new peer Object();
any Object z = new rep Object();
sync (x) { z.f = 20 } // OK, but race condition!
```

Solution:

```
\[ \Gamma \vdash e : u \]
\[ u \neq \text{any} \]
\[ \Gamma \vdash gb e : u \]
\[ \Gamma \vdash gb p : p \]
```

Universe Types for Race Safety
Examples

```
peer getPeer() { ... }
any getAny() { ... }

any Object x = ...;
peer Object y = ...;
rep Object z = ...;

sync (x) { x.f } // OK (path)
sync (y) { y.f } // OK (path) (universes)
sync (y) { z.f } // error!
sync (getPeer()) { y.f } // OK (universes)
sync (getAny()) { x.f } // error!
sync (x) { x=... ; x.f } // error!
sync (x) { x.f ; x=... } // error!  (not flow sensitive)
```
Conclusion

Advantages of ownership:

- Locks scale with size of program

Disadvantages of ownership:

- Require ownership annotations

Notions of ownership also used by

- C. Flanagan et al (ESOP’99, CONCUR’99, PLDI’00, LICS’00, TLDI’03, PLDI’03, SAS’04, POPL’04, SPIN’04, TLDI’05, ECOOP’05)
- C. Boyapati et al (OOPSLA’01, OOPSLA’02)
- **Autolocker** – B. McCloskey et al (POPL’06)
Summary

We have

- given a race-safety type system that uses a $\vdash_{gb}$ judgement.
- Shown the weakness of a path-based $\vdash_{gb} p : p$
- put objects into boxes and restricted assignment
  - with a static set of regions, and
  - with dynamic set of universes that grows at runtime
- in order to build a more powerful $\vdash_{gb}$.
- used the simple path-based $\vdash_{gb}$ with the universes $\vdash_{gb}$, to allow locking of any.
Atomicity

A race-safe block of code is atomic if its sync. is two-phase:

// GOOD
atomic {
  sync (x) {
    sync (y) {
      ...
    }
    ...  
  }
}

// UGLY (but good, and useful too)
atomic {
  sync (x) {
    sync (y) {
      sync (x) {
        ...
      }
      ...
    }
  }
}

// BAD
atomic {
  sync (x) {
    ...  
  }
  sync (y) {
    ...  
  }
}