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... $\Longrightarrow$

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... $\Longrightarrow$ Static type system?

Race Conditions

Race conditions:

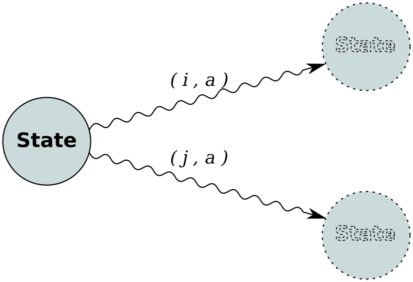
- are bugs in shared memory concurrent software.
- are caused by incorrect synchronisation.
- are hard to reproduce.
- can corrupt program state.
- can lead to strange program behaviour.

Testing hard... $\Longrightarrow$ Static type system?

Can prove it correct...

Instantaneous Race Condition

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.

General Approach

Enforcing synchronisation is the key:

```
sync (e') {

        ...
        e.f = 10;
        ...
}
```

General Approach

Enforcing synchronisation is the key:

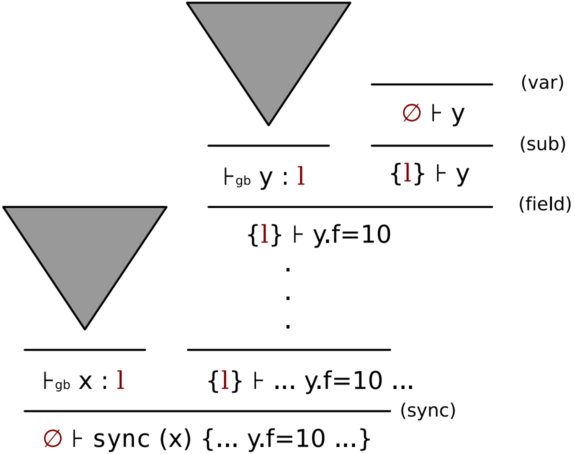
```
sync (e') {

        ...
        e.f = 10;
        ...
}
```

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

$\vdash_{gb}$ e' : l

$\vdash_{gb} e : \mathrm{l}$

Example



Type System (for illustrative purposes only!)

$$\frac{}{\emptyset \vdash \texttt{this}}\text{(Var)} \qquad \frac{\mathbb{L} \vdash e \quad \vdash_{gb} e : \mathrm{l} \quad \mathrm{l} \in \mathbb{L}}{\mathbb{L} \vdash e.f}\text{(Field)} \qquad \frac{\mathbb{L} \vdash e' \quad \vdash_{gb} e' : \mathrm{l} \quad \mathbb{L} \cup \{\mathrm{l}\} \vdash e}{\mathbb{L} \vdash \texttt{sync}\ e'\ e}\text{(Sync)}$$

$$\frac{\mathbb{L}' \vdash e \quad \mathbb{L}' \subseteq \mathbb{L}}{\mathbb{L} \vdash e}\text{(Sub)}$$

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 = 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

$$
\dfrac{
  \dfrac{}{\vdash_{gb} \mathrm{x.f} : \mathrm{x.f}}\,\text{(path)}
  \qquad
  \dfrac{
    \dfrac{}{\vdash_{gb} \mathrm{x.f} : \mathrm{x.f}}\,\text{(path)}
    \qquad
    \dfrac{
      \dfrac{}{\vdash_{gb} \mathrm{x} : \mathrm{x}}\,\text{(path)}
      \qquad
      \dfrac{
        \dfrac{}{\emptyset \vdash \mathrm{x}}\,\text{(var)}
      }{\{\mathrm{x},\mathrm{x.f}\} \vdash \mathrm{x}}\,\text{(sub)}
    }{\{\mathrm{x},\mathrm{x.f}\} \vdash \mathrm{x.f}}\,\text{(field)}
  }{\{\mathrm{x},\mathrm{x.f}\} \vdash \mathrm{x.f.g{=}10}}\,\text{(field)}
}{\{\mathrm{x}\} \vdash \mathrm{sync}\ (\mathrm{x.f})\ \{\mathrm{x.f.g{=}20}\}}\,\text{(sync)}
$$

A problem

$\emptyset \vdash$ `sync (x) { x=y ; x.f=20 }`

A problem

$\emptyset \vdash$ `sync (x) { x=y ; x.f=20 }`

↑ accesses the object y

A problem

$\emptyset \vdash$ `sync (x) { x=y ; x.f=20 }`

↑ accesses the object `y`

similarly...

$\{x\} \vdash$ `sync (x.f) { x.f=y ; x.f.g=20 }`

↑ accesses the object `y`

A problem

$\emptyset \vdash$ `sync (x) { x=y ; x.f=20 }`

↑ accesses the object `y`

similarly...

$\{x\} \vdash$ `sync (x.f) { x.f=y ; x.f.g=20 }`

↑ 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;
        }
}
```

Iteration

```
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?

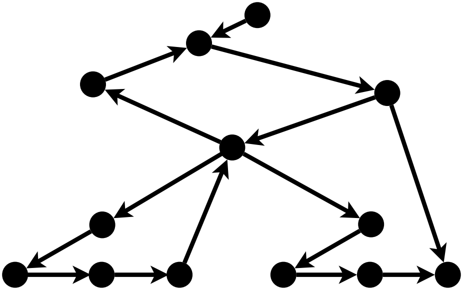
This demonstrates that:

- 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.

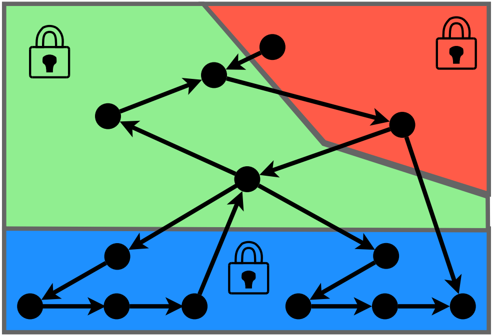
Carving the Heap

Artist's impression of a heap:



Carving the Heap

Artist's impression of a heap:



Regions

Other work has used a programmer-supplied set, e.g. $\{\mathtt{RED}, \mathtt{BLUE}\}$

The source code looks like:

```
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:

$$\Gamma \vdash e : R$$

Regions as Locks

Suppose we already have a region type system:

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

Regions as Locks

Suppose we already have a region type system:

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

Note we now need a Γ in the race type system too: $\mathbb{L}, \Gamma \vdash e : F$

Regions as Locks

Suppose we already have a region type system:

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

Note we now need a Γ in the race type system too: $\mathbb{L}, \Gamma \vdash e : F$

```
RED Object r1, r2 = ...
BLUE Object b = ...
sync(r1) {
      b.f = 10; // not allowed
      r2.f = 10; // OK
}
```

Iteration Example

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

RED Node i = ...;

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

Summary

Carving up the heap helps us verify safe locking:

- `x.f = y ; x.f.g = 10`
  (must lock l where $\Gamma \vdash_{gb} y : \mathrm{l}$)

Summary

Carving up the heap helps us verify safe locking:

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

Summary 2

Advantages of carving with regions:

- Simple
- Inference is easy

Disadvantages of regions:

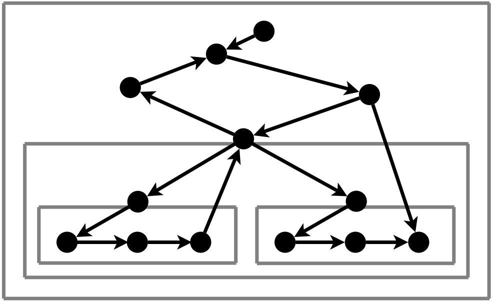
- Lock count does not scale with object count

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)

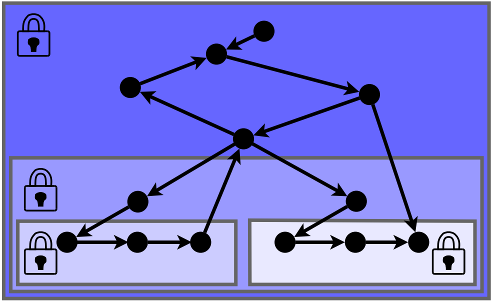
Carving the Heap Again

Ownership types impose a heap hierarchy:



Carving the Heap Again

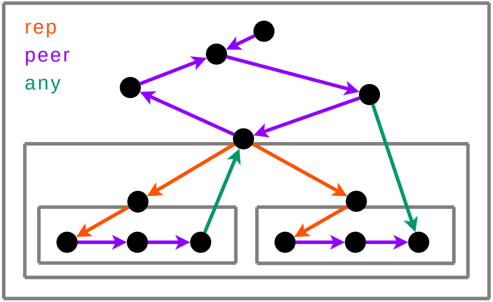
Ownership types impose a heap hierarchy:



Can use the "owner" of an object as its lock.

Universes

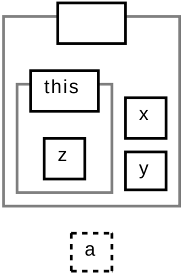
Universes form this hierarchy with 3 keywords



The keywords indicate the relative position of the referenced object.

Example

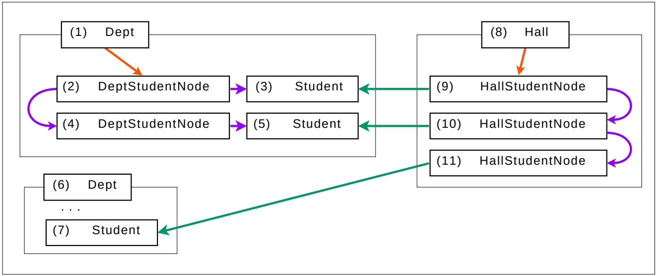
```
class C {
  peer Object m(peer Object x) {
    peer Object y = new peer Object();
    rep Object z = new rep Object();
    x = y;
    x = z; // not allowed
    any Object a = z;
    z = a; // not allowed
    return y;
  }
}
rep Object o = new rep C().m(new rep Object());
```



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 $\Gamma \vdash e : u$

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

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

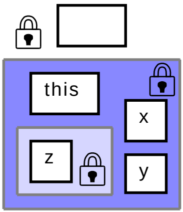
Synchronisation

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

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

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

```
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!
```



Iteration

```
class Node { peer Node next ; int cargo }

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

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!
```

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:

$$\frac{\Gamma \vdash e : u \qquad u \neq \texttt{any}}{\Gamma \vdash_{gb} e : u}$$

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:

$$\frac{\Gamma \vdash e : u \qquad u \neq \texttt{any}}{\Gamma \vdash_{gb} e : u}$$

$$\frac{}{\Gamma \vdash_{gb} p : p}$$

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) {
            ...
            ...
        }
    }
}

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

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