Atomicity Analysis of Concurrent Software

Cormac Flanagan
UC Santa Cruz

Stephen N. Freund
Williams College

Shaz Qadeer
Microsoft Research

Types Against Races

Moore's Law

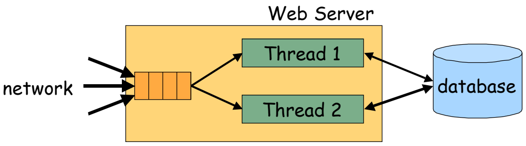
- Transistors per chip doubles every 18 months
- Single-threaded performance doubles every 2 years
  - faster clocks, deep pipelines, multiple issue
  - wonderful!

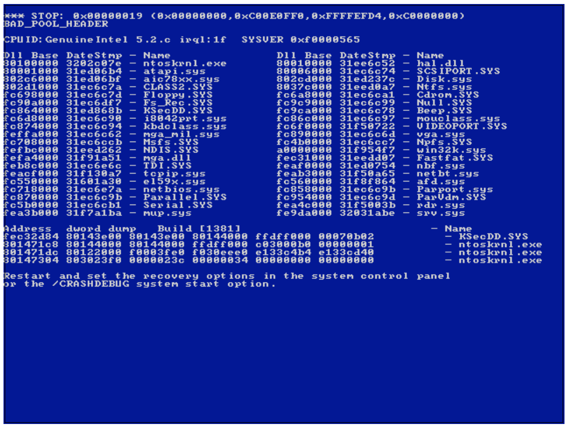
Moore's Law is Over

- Sure, we can pack more transistors ...
  - ... but can't use them effectively to make single-threaded programs faster
- Multi-core is the future of hardware
- Multi-threading is the future of software

Programming With Threads

- Decompose program into parallel threads
- Advantages
  - exploit multiple cores/processors
  - some threads progress, even if others block
- Increasingly common (Java, C#, GUIs, servers)





```
*** STOP: 0x00000019 (0x00000000,0xC00E0FF0,0xFFFFEFD4,0xC0000000)
BAD_POOL_HEADER

CPUID:GenuineIntel 5.2.c irql:1f  SYSVER 0xf0000565

Dll Base DateStmp - Name                 Dll Base DateStmp - Name
80100000 3202c07e - ntoskrnl.exe         80010000 31ee6c52 - hal.dll
80001000 31ed06b4 - atapi.sys            80006000 31ec6c74 - SCSIPORT.SYS
802c6000 31ed06bf - aic78xx.sys          802cd000 31ed237c - Disk.sys
802d1000 31ec6c7a - CLASS2.SYS           8037c000 31eed0a7 - Ntfs.sys
fc698000 31ec6c7d - Floppy.SYS           fc6a8000 31ec6ca1 - Cdrom.SYS
fc90a000 31ec6df7 - Fs_Rec.SYS           fc9c9000 31ec6c99 - Null.SYS
fc864000 31ed868b - KSecDD.SYS           fc9ca000 31ec6c78 - Beep.SYS
fc6d8000 31ec6c90 - i8042prt.sys         fc86c000 31ec6c97 - mouclass.sys
fc874000 31ec6c94 - kbdclass.sys         fc6f0000 31f50722 - VIDEOPORT.SYS
feffa000 31ec6c62 - mga_mil.sys          fc890000 31ec6c6d - vga.sys
fc708000 31ec6ccb - Msfs.SYS             fc4b0000 31ec6cc7 - Npfs.SYS
fefbc000 31eed262 - NDIS.SYS             a0000000 31f954f7 - win32k.sys
fefa4000 31f91a51 - mga.dll              fec31000 31eedd07 - Fastfat.SYS
feb8c000 31ec6e6c - TDI.SYS              feaf0000 31ed0754 - nbf.sys
feacf000 31f130a7 - tcpip.sys            feab3000 31f50a65 - netbt.sys
fc550000 31601a30 - el59x.sys            fc560000 31f8f864 - afd.sys
fc718000 31ec6e7a - netbios.sys          fc858000 31ec6c9b - Parport.sys
fc870000 31ec6c9b - Parallel.SYS         fc954000 31ec6c9d - ParVdm.SYS
fc5b0000 31ec6cb1 - Serial.SYS           fea4c000 31f5003b - rdr.sys
fea3b000 31f7a1ba - mup.sys              fe9da000 32031abe - srv.sys

Address  dword dump   Build [1381]                            - Name
fec32d84 80143e00 80143e00 80144000 ffdff000 00070b02          - KSecDD.SYS
801471c8 80144000 80144000 ffdff000 c03000b0 00000001          - ntoskrnl.exe
801471dc 80122000 f0003fe0 f030eee0 e133c4b4 e133cd40          - ntoskrnl.exe
80147304 803023f0 0000023c 00000034 00000000 00000000          - ntoskrnl.exe

Restart and set the recovery options in the system control panel
or the /CRASHDEBUG system start option.
```

Concurrency is a problem

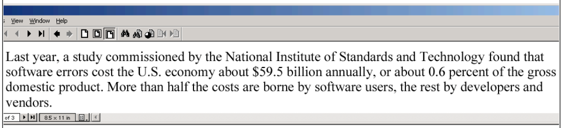
- Windows 2000 hot fixes
  - – Concurrency errors most common defects among “detectable errors”
  - – Incorrect synchronization and protocol errors most common defects among all coding errors
- Windows Server 2003 late cycle defects
  - – Synchronization errors second in the list, next to buffer overruns

Security vulnerabilities involving race conditions

- Buffer overruns
- Phishing attacks
  - – “A systematic approach to uncover visual ambiguity vulnerabilities”, MSR Technical Report MSR-TR-2006-48 by Chen et al.

Economic Impact

- NIST study



Last year, a study commissioned by the National Institute of Standards and Technology found that software errors cost the U.S. economy about $59.5 billion annually, or about 0.6 percent of the gross domestic product. More than half the costs are borne by software users, the rest by developers and vendors.

`http://www.nist.gov/director/prog-ofc/report02-3.pdf`

Non-Determinism, Heisenbugs

- Multithreaded programs are non-deterministic
  - – behavior depends on interleaving of threads
- Extremely difficult to test
  - – exponentially many interleavings
  - – during testing, many interleavings behave correctly
  - – post-deployment, other interleavings fail
- Complicates code reviews, static analysis, ...

Mars, July 4, 1997
Lost contact due to real-time priority inversion bug

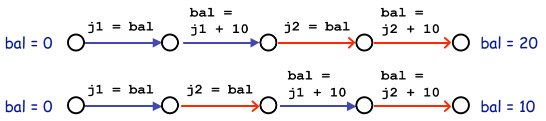
400 horses
100 microprocessors

Bank Account Implementation

```
class Account {
  private int bal = 0;

  public void deposit(int n) {
    int j = bal;
    bal = j + n;
  }
}
```

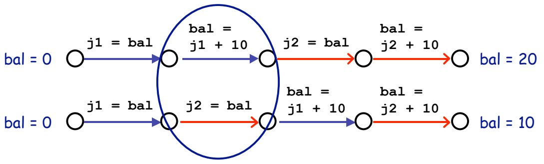
Bank Account Implementation

```
class Account {
  private int bal = 0;

  public void deposit(int n) {
    int j = bal;
    bal = j + n;
  }
}
```



Bank Account Implementation

A *race condition* occurs if two threads access a shared variable at the same time, and at least one of the accesses is a write



Race Conditions

```
class Ref {
  int i;
  void add(Ref r) {
    i = i
        + r.i;
  }
}
```

Race Conditions

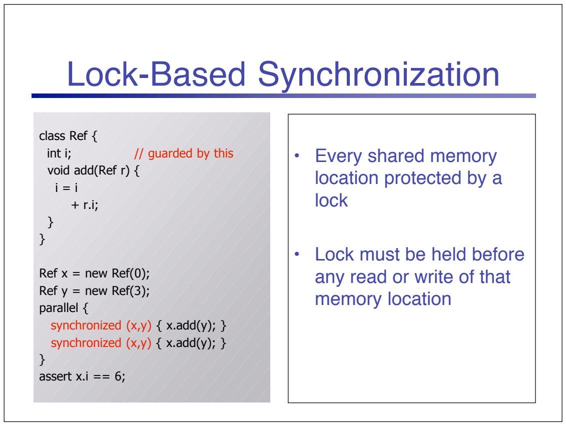
```
class Ref {
  int i;
  void add(Ref r) {
    i = i
        + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);

    x.add(y);
    x.add(y);

assert x.i == 6;
```

Race Conditions

```
class Ref {
  int i;
  void add(Ref r) {
    i = i
        + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
    x.add(y);    // two calls happen
    x.add(y);    // in parallel
}
assert x.i == 6;
```

Race condition on x.i

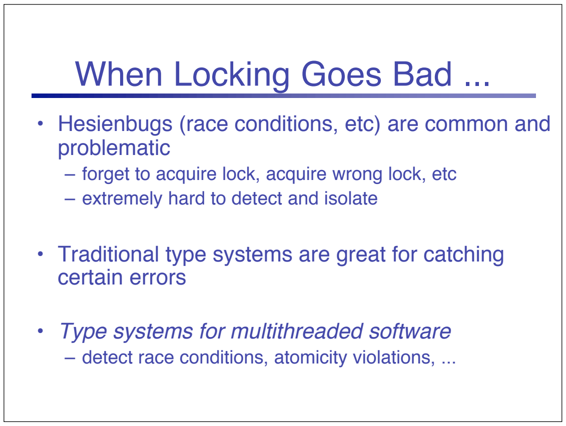
Assertion may fail



Lock-Based Synchronization

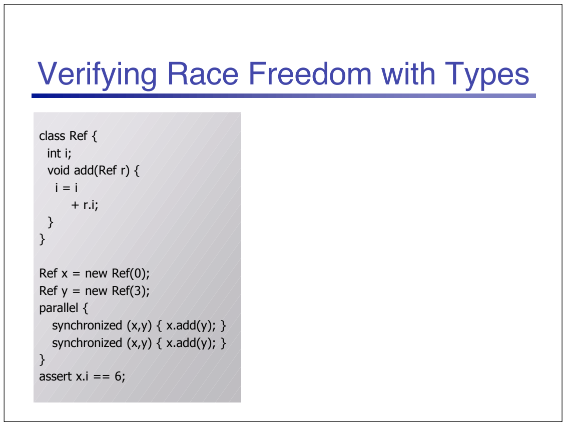
```
class Ref {
  int i;              // guarded by this
  void add(Ref r) {
    i = i
        + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

- Every shared memory location protected by a lock
- Lock must be held before any read or write of that memory location



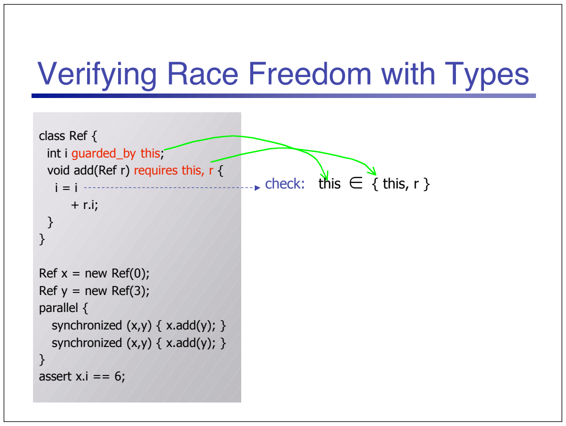
When Locking Goes Bad ...

- Hesienbugs (race conditions, etc) are common and problematic
  - forget to acquire lock, acquire wrong lock, etc
  - extremely hard to detect and isolate
- Traditional type systems are great for catching certain errors
- *Type systems for multithreaded software*
  - detect race conditions, atomicity violations, ...



Verifying Race Freedom with Types

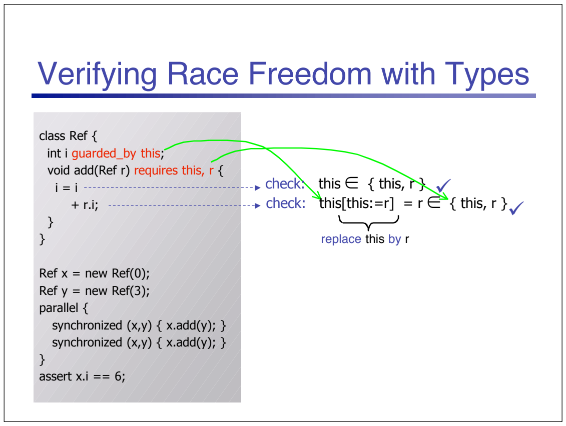
```
class Ref {
  int i;
  void add(Ref r) {
    i = i
        + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```



Verifying Race Freedom with Types

```
class Ref {
  int i guarded_by this;
  void add(Ref r) requires this, r {
    i = i
        + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

check: this ∈ { this, r }



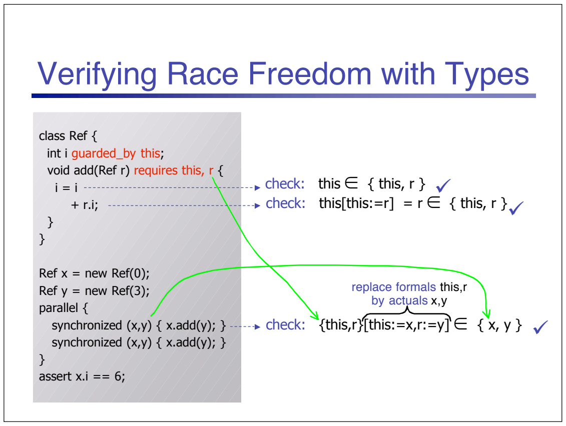
Verifying Race Freedom with Types

```
class Ref {
  int i guarded_by this;
  void add(Ref r) requires this, r {
    i = i
        + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

check: this ∈ { this, r } ✓

check: this[this:=r] = r ∈ { this, r } ✓

replace this by r



Verifying Race Freedom with Types

```
class Ref {
  int i guarded_by this;
  void add(Ref r) requires this, r {
    i = i
        + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

check: this ∈ { this, r } ✓

check: this[this:=r] = r ∈ { this, r } ✓

replace formals this,r by actuals x,y

check: {this,r}[this:=x,r:=y] ∈ { x, y } ✓

Verifying Race Freedom with Types

```
class Ref {
  int i guarded_by this;
  void add(Ref r) requires this, r {
    i = i
       + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

check: this ∈ { this, r } ✓

check: this[this:=r] = r ∈ { this, r } ✓

check: $\{this,r\}\overbrace{[this:=x,r:=y]}^{\text{replace formals this,r by actuals x,y}} \in \{ x, y \}$ ✓

check: {this,r}[this:=x,r:=y] ∈ { x, y } ✓

Soundness Theorem:
Well-typed programs are race-free

One Problem ...

```
Object o;
int x guarded_by o;

fork { sync(o) { x++; } }

fork { o = new Object();
       sync(o) { x++; }
     }
```

- Lock expressions must be constant

Lock Equality

- Type system checks if lock is in lock set
  - r ∈ { this, r }
  - same as r = this ∨ r = r
- Semantic equality
  - $e_1 = e_2$ if e_1 and e_2 refer to same object
  - need to test whether two program expressions evaluate to same value
  - undecidable in general

Lock Equality

- Approximate (undecidable) semantic equality by syntactic equality
  - two locks expressions are considered equal only if syntactically identical
- Conservative approximation

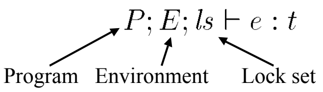
```
class A {
  void f() requires this { ... }
}

A p = new A();
q = p;
synch(q) {  p.f(); }
```

this[this:=p] = p ∈ { q } X

- Not a major source of imprecision

RaceFreeJava

- Concurrent extension of CLASSICJAVA [Flatt-Krishnamurthi-Felleisen 99]
- Judgement for typing expressions



$$P; E; ls \vdash e : t$$

Typing Rules

- Thread creation

$$\frac{P; E; \emptyset \vdash e : t}{P; E; ls \vdash \texttt{fork}\ e : \texttt{int}}$$

- Synchronization

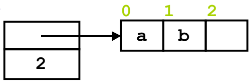
$$\frac{\begin{array}{ll} P; E \vdash_{\text{final}} e_1 : c & \text{lock is constant} \\ P; E; ls \cup \{e_1\} \vdash e_2 : t & \text{add to lock set} \end{array}}{P; E; ls \vdash \texttt{synchronized}\ e_1\ \texttt{in}\ e_2 : t}$$

Field Access

$$\frac{P; E; ls \vdash e : c \qquad P; E \vdash (t\ fd\ \text{guarded\_by}\ l) \in c \qquad P; E \vdash [e/\mathtt{this}]l \in ls}{P; E; ls \vdash e.fd : [e/\mathtt{this}]t}$$

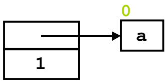
e has class *c*
fd is declared in *c*
lock *l* is held

`java.util.Vector`



```
class Vector {
  Object elementData[] /*# guarded_by this */;
  int elementCount     /*# guarded_by this */;

  synchronized int lastIndexOf(Object elem, int n) {
    for (int i = n ; i >= 0 ; i--)
      if (elem.equals(elementData[i])) return i;
    return -1;
  }

  int lastIndexOf(Object elem) {
    return lastIndexOf(elem, elementCount - 1);
  }

  synchronized void trimToSize() { ... }
  synchronized boolean remove(int index) { ... }
}
```

`java.util.Vector`



```
class Vector {
  Object elementData[] /*# guarded_by this */;
  int elementCount     /*# guarded_by this */;

  synchronized int lastIndexOf(Object elem, int n) {
    for (int i = n ; i >= 0 ; i--)
      if (elem.equals(elementData[i])) return i;
    return -1;
  }

  int lastIndexOf(Object elem) {
    return lastIndexOf(elem, elementCount - 1);
  }

  synchronized void trimToSize() { ... }
  synchronized boolean remove(int index) { ... }
}
```

Validation of `rccjava`

Program	Size (lines)	Number of annotations	Annotation time (hrs)	Races Found
Hashtable	434	60	0.5	0
Vector	440	10	0.5	1
java.io	16,000	139	16.0	4
Ambit	4,500	38	4.0	4
WebL	20,000	358	12.0	5

Basic Type Inference

```
class Ref {
  int i;
  void add(Ref r) {
    i = i + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

Basic Type Inference

```
static final Object m =new Object();

class Ref {
  int i;
  void add(Ref r) {
    i = i + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

Iterative GFP algorithm:
- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations

Basic Type Inference

```
static final Object m =new Object();

class Ref {
  int i guarded_by this, m;
  void add(Ref r) {
    i = i + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

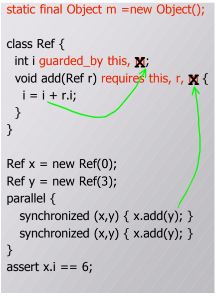
Iterative GFP algorithm:
- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations

Basic Type Inference

```
static final Object m =new Object();

class Ref {
  int i guarded_by this, m;
  void add(Ref r) requires this, r, m {
    i = i + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

Iterative GFP algorithm:
- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations

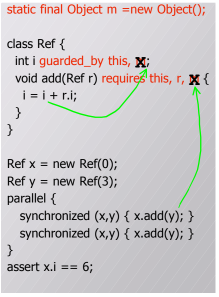
Basic Type Inference



```
static final Object m =new Object();

class Ref {
  int i guarded_by this, ~~m~~;
  void add(Ref r) requires this, r, ~~m~~ {
    i = i + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

Iterative GFP algorithm:
- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations
- Iteratively remove all incorrect annotations

Basic Type Inference



```
static final Object m =new Object();

class Ref {
  int i guarded_by this, ~~m~~;
  void add(Ref r) requires this, r, ~~m~~ {
    i = i + r.i;
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (x,y) { x.add(y); }
   synchronized (x,y) { x.add(y); }
}
assert x.i == 6;
```

Iterative GFP algorithm:
- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations
- Iteratively remove all incorrect annotations
- Check each field still has a protecting lock

Sound, complete, fast

But type system too basic

Harder Example: External Locking

```
class Ref {
  int i;
  void add(Ref r) {
    i = i + r.i;
  }
}

Object m = new Object();
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (m) { x.add(y); }
   synchronized (m) { x.add(y); }
}
assert x.i == 6;
```

- Field i of x and y protected by *external* lock m
- Not typable with basic type system
  - m not in scope at i
- Requires more expressive type system with *ghost parameters*

Ghost Parameters on Classes

```
class Ref {
  int i;
  void add(Ref r) {
    i = i + r.i;
  }
}

Object m = new Object();
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (m) { x.add(y); }
   synchronized (m) { x.add(y); }
}
assert x.i == 6;
```

Ghost Parameters on Classes

```
class Ref<ghost g> {
  int i;
  void add(Ref r) {
    i = i + r.i;
  }
}

Object m = new Object();
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (m) { x.add(y); }
   synchronized (m) { x.add(y); }
}
assert x.i == 6;
```

- Ref parameterized by external ghost lock g

Ghost Parameters on Classes

```
class Ref<ghost g> {
  int i guarded_by g;
  void add(Ref r) {
    i = i + r.i;
  }
}

Object m = new Object();
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (m) { x.add(y); }
   synchronized (m) { x.add(y); }
}
assert x.i == 6;
```

- Ref parameterized by external ghost lock g
- Field i guarded by g

Ghost Parameters on Classes

```
class Ref<ghost g> {
  int i guarded_by g;
  void add(Ref r) requires g {
    i = i + r.i;
  }
}

Object m = new Object();
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (m) { x.add(y); }
   synchronized (m) { x.add(y); }
}
assert x.i == 6;
```

- Ref parameterized by external ghost lock g
- Field i guarded by g
- g held when add called

Ghost Parameters on Classes

```
class Ref<ghost g> {
  int i guarded_by g;
  void add(Ref<g> r) requires g {
    i = i + r.i;
  }
}

Object m = new Object();
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
   synchronized (m) { x.add(y); }
   synchronized (m) { x.add(y); }
}
assert x.i == 6;
```

- Ref parameterized by external ghost lock g
- Field i guarded by g
- g held when add called
- Argument r also parameterized by g

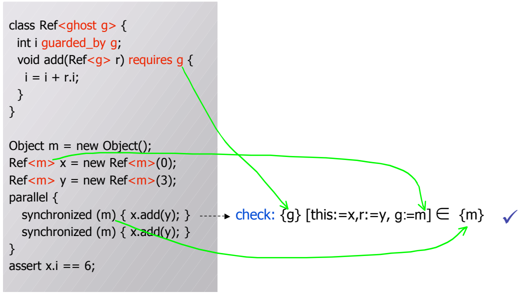
Ghost Parameters on Classes

```
class Ref<ghost g> {
  int i guarded_by g;
  void add(Ref<g> r) requires g {
    i = i + r.i;
  }
}

Object m = new Object();
Ref<m> x = new Ref<m>(0);
Ref<m> y = new Ref<m>(3);
parallel {
   synchronized (m) { x.add(y); }
   synchronized (m) { x.add(y); }
}
assert x.i == 6;
```

- Ref parameterized by external ghost lock g
- Field i guarded by g
- g held when add called
- Argument r also parameterized by g

- x and y parameterized by lock m

Type Checking Ghost Parameters



```
class Ref<ghost g> {
  int i guarded_by g;
  void add(Ref<g> r) requires g {
    i = i + r.i;
  }
}

Object m = new Object();
Ref<m> x = new Ref<m>(0);
Ref<m> y = new Ref<m>(3);
parallel {
   synchronized (m) { x.add(y); }
   synchronized (m) { x.add(y); }
}
assert x.i == 6;
```

check: {g} [this:=x,r:=y, g:=m] ∈ {m} ✓

Type Inference with Ghosts

- Type inference is NP-complete
  - iterative GFP algorithm does not work
  - ghost parameters require backtracking search
- RccSAT: Reduce to SAT
  - works up to 30 KLOC
  - precise: 92-100% of fields verified race free