Analysis of Concurrent Software

Types for Race Freedom

Cormac Flanagan
UC Santa Cruz

Stephen N. Freund
Williams College

Shaz Qadeer
Microsoft Research

Moore's Law

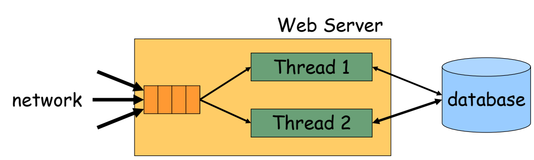
- Transistors per chip doubles every 18 months
- Single-threaded performance doubles every 2 years
  - faster clocks, deep pipelines, multiple issue
  - wonderful!

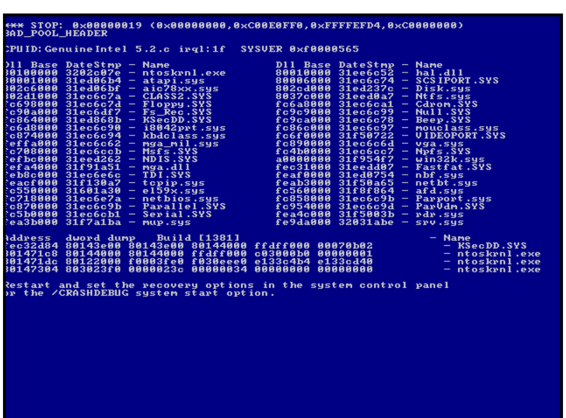
Moore's Law is Over

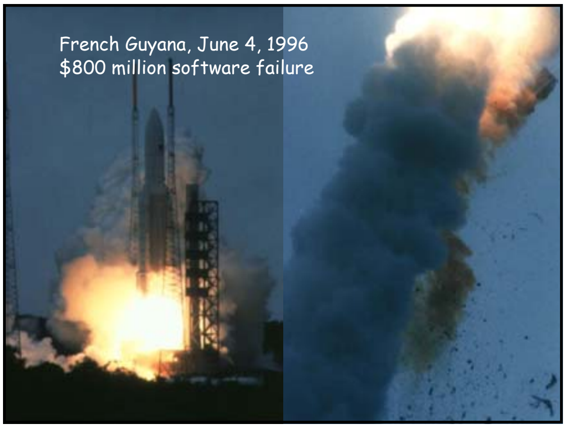
- Sure, we can pack more transistors ...
  - ... but can't use them effectively to make single-threaded programs faster
- Multi-core is the future of hardware
- Multi-threading is the future of software

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)







Economic Impact

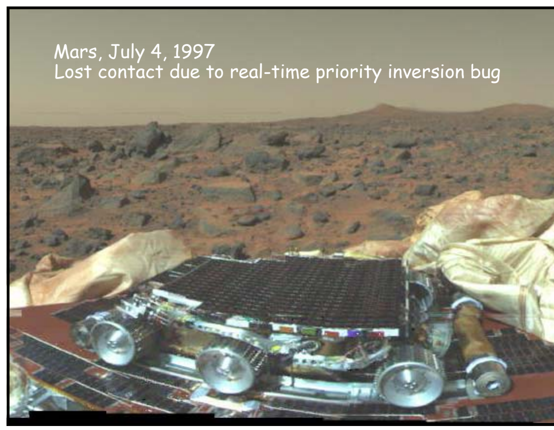
- NIST study

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, ...



Bank Account Implementation

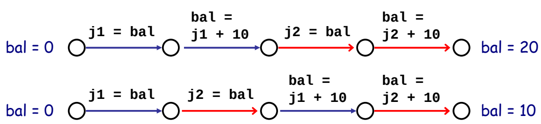
```
class Account {
  private int bal = 0;

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

Bank Account Implementation

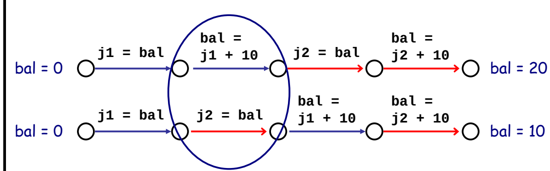
```
class Account {
  private int bal = 0;

  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                                check: this ∈ { this, r } ✓
      + 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                                check: this ∈ { this, r } ✓
      + r.i;                             check: this[this:=r] = r ∈ { this, r } ✓
  }                                             (replace this by r)
}

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                                check: this ∈ { this, r } ✓
      + r.i;                             check: this[this:=r] = r ∈ { this, r } ✓
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);                       (replace formals this,r by actuals x,y)
parallel {
  synchronized (x,y) { x.add(y); }       check: {this,r}[this:=x,r:=y] ⊆ { x, 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                                check: this ∈ { this, r } ✓
      + r.i;                             check: this[this:=r] = r ∈ { this, r } ✓
  }
}

Ref x = new Ref(0);
Ref y = new Ref(3);                       (replace formals this,r by actuals x,y)
parallel {
  synchronized (x,y) { x.add(y); }       check: {this,r}[this:=x,r:=y] ⊆ { x, y } ✓
  synchronized (x,y) { x.add(y); }       check: {this,r}[this:=x,r:=y] ⊆ { x, y } ✓
}
assert x.i == 6;
```

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 (Halting Problem)

Lock Equality

- Approximate (undecidable) semantic equality by syntactic equality
  - two locks exprs 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

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

Program Environment Lock set

Typing Rules

- Thread creation

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

- Synchronization

$$\frac{\begin{array}{l} P; E \vdash_{\text{final}} e_1 : c \\ P; E; ls \cup \{e_1\} \vdash e_2 : t \end{array}}{P; E; ls \vdash \texttt{synchronized}\ e_1\ \texttt{in}\ e_2 : t}$$

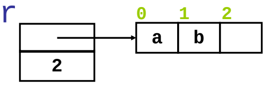
lock is constant
add to lock set

Field Access

$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash (t\ fd\ \texttt{guarded\_by}\ l) \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \end{array}}{P; E; ls \vdash e.fd : [e/\texttt{this}]t}$$

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

java.util.Vector



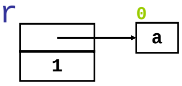

```
class Vector {
  Object elementData[] /*# guarded_by this */;
  int elementCount     /*# guarded_by this */;

  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

- HARD
  - iterative GFP algorithm does not work
  - check may fail because of *two* annotations
    - which should we remove?
  - requires backtracking search

Type Inference with Ghosts

```
class A
{
  int f;
}
class B<ghost y>
...
A a = ...;
```

→ Type Inference →

```
class A<ghost g>
{
  int f guarded_by g;
}
class B<ghost y>
...
A<m> a = ...;
```


Boolean Satisfiability

```
(t1 ∨ t2 ∨ t3) ∧
(t2 ∨ ¬t1 ∨ ¬t4) ∧
(t2 ∨ ¬t3 ∨ t4)
```

→ SAT Solver →

```
t1 = true
t2 = false
t3 = true
t4 = true
```


Reducing SAT to Type Inference

```
class A<ghost x,y,z> ...
class B ...
class C ...
A a = ...
B b = ...
C c = ...
```

→ Type Inference →

```
class A<ghost x,y,z>...
class B<ghost x,y,z>...
class C<ghost x,y,z>...
A<p1,p2,p3> a = ...
B<p1,n1,n4> b = ...
C<p2,n3,p4> c = ...
```

Construct Program From Formula ↑

Construct Assignment From Annotations ↓

```
(t1 ∨ t2 ∨ t3) ∧
(t2 ∨ ¬t1 ∨ ¬t4) ∧
(t2 ∨ ¬t3 ∨ t4)
```

→ SAT Solver →

```
t1 = true
t2 = false
t3 = true
t4 = true
```


Rcc/Sat Type Inference Tool

```
class A
{
  int f;
  ..
}
...
A a = ...;
```

```
class A<ghost g>
{
  int f guarded_by g;
  ..
}
...
A<m> a = ...;
```

↓ Construct Formula From Program ↓

↑ Construct Annotations From Assignment ↑

```
(t1 ∨ t2 ∨ t3) ∧
(t2 ∨ ¬t1 ∨ ¬t4) ∧
(t2 ∨ ¬t3 ∨ t4)
```

→ SAT Solver →

```
t1 = true
t2 = false
t3 = true
t4 = true
```


Reducing Type Inference to SAT

```
class Ref {
  int i;
  void add(Ref r)

  {
    i = i
        + r.i;
  }
}
```


Reducing Type Inference to SAT

```
class Ref<ghost g1,g2,...,gn> {
  int i;
  void add(Ref r)

  {
    i = i
        + r.i;
  }
}
```

Reducing Type Inference to SAT

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

- Add ghost parameters <ghost g> to each class declaration

Reducing Type Inference to SAT

class Ref<ghost g> {
int i guarded_by α_1;
void add(Ref r)
{
i = i
+ r.i;
}
}

- Add ghost parameters <ghost g> to each class declaration
- Add guarded_by α_i to each field declaration
  - type inference resolves α_i to some lock

Reducing Type Inference to SAT

class Ref<ghost g> {
int i guarded_by α_1;
void add(Ref<α_2> r)
{
i = i
+ r.i;
}
}

- Add ghost parameters <ghost g> to each class declaration
- Add guarded_by α_i to each field declaration
  - type inference resolves α_i to some lock
- Add <α_2> to each class reference

Reducing Type Inference to SAT

class Ref<ghost g> {
int i guarded_by α_1;
void add(Ref<α_2> r)
requires β
{
i = i
+ r.i;
}
}

- Add ghost parameters <ghost g> to each class declaration
- Add guarded_by α_i to each field declaration
  - type inference resolves α_i to some lock
- Add <α_2> to each class reference
- Add requires β_i to each method
  - type inference resolves β_i to some set of locks

Reducing Type Inference to SAT

class Ref<ghost g> {
int i guarded_by α_1; → $\alpha_1 \in$ { this, g }
void add(Ref<α_2> r) → $\alpha_2 \in$ { this, g }
requires β → $\beta \subseteq$ { this, g, r }
{
i = i → $\alpha_1 \in \beta$
+ r.i; → α_1[this := r, g:= α_2] $\in \beta$
}
}

Constraints:
$\alpha_1 \in$ { this, g }
$\alpha_2 \in$ { this, g }
$\beta \subseteq$ { this, g, r }
$\alpha_1 \in \beta$
α_1[this := r, g:= α_2] $\in \beta$

Reducing Type Inference to SAT

class Ref<ghost g> {
int i guarded_by α_1;
void add(Ref<α_2> r)
requires β
{
i = i
+ r.i;
}
}

Constraints:
$\alpha_1 \in$ { this, g }
$\alpha_2 \in$ { this, g }
$\beta \subseteq$ { this, g, r }
$\alpha_1 \in \beta$
α_1[this := r, g:= α_2] $\in \beta$

Encoding:
α_1 = (b1 ? this : g)
α_2 = (b2 ? this : g)
β = { b3 ? this, b4 ? g, b5 ? r }

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

Reducing Type Inference to SAT

```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref<α2> r)
    requires β
  {
    i = i
      + r.i;
  }
}
```

Constraints:

$\alpha_1 \in$ { this, g }
$\alpha_2 \in$ { this, g }
$\beta \subseteq$ { this, g, r }
$\alpha_1 \in \beta$
α_1[this := r, g:= α_2] $\in \beta$

Encoding:

α_1 = (b1 ? this : g)
α_2 = (b2 ? this : g)
β = { b3 ? this, b4 ? g, b5 ? r }

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

α_1[this := r, g:= α_2] $\in \beta$

Reducing Type Inference to SAT

```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref<α2> r)
    requires β
  {
    i = i
      + r.i;
  }
}
```

Constraints:

$\alpha_1 \in$ { this, g }
$\alpha_2 \in$ { this, g }
$\beta \subseteq$ { this, g, r }
$\alpha_1 \in \beta$
α_1[this := r, g:= α_2] $\in \beta$

Encoding:

α_1 = (b1 ? this : g)
α_2 = (b2 ? this : g)
β = { b3 ? this, b4 ? g, b5 ? r }

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

α_1[this := r, g:= α_2] $\in \beta$
(b1 ? this : g) [this := r, g:= α_2] $\in \beta$

Reducing Type Inference to SAT

```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref<α2> r)
    requires β
  {
    i = i
      + r.i;
  }
}
```

Constraints:

$\alpha_1 \in$ { this, g }
$\alpha_2 \in$ { this, g }
$\beta \subseteq$ { this, g, r }
$\alpha_1 \in \beta$
α_1[this := r, g:= α_2] $\in \beta$

Encoding:

α_1 = (b1 ? this : g)
α_2 = (b2 ? this : g)
β = { b3 ? this, b4 ? g, b5 ? r }

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

α_1[this := r, g:= α_2] $\in \beta$
(b1 ? this : g) [this := r, g:= α_2] $\in \beta$
(b1 ? r : α_2) $\in \beta$

Reducing Type Inference to SAT

```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref<α2> r)
    requires β
  {
    i = i
      + r.i;
  }
}
```

Constraints:

$\alpha_1 \in$ { this, g }
$\alpha_2 \in$ { this, g }
$\beta \subseteq$ { this, g, r }
$\alpha_1 \in \beta$
α_1[this := r, g:= α_2] $\in \beta$

Encoding:

α_1 = (b1 ? this : g)
α_2 = (b2 ? this : g)
β = { b3 ? this, b4 ? g, b5 ? r }

Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

α_1[this := r, g:= α_2] $\in \beta$
(b1 ? this : g) [this := r, g:= α_2] $\in \beta$
(b1 ? r : α_2) $\in \beta$
(b1 ? r : (b2 ? this : g)) $\in$ { b3 ? this, b4 ? g, b5 ? r }

Reducing Type Inference to SAT

```
class Ref<ghost g> {
  int i guarded_by α1;
  void add(Ref<α2> r)
    requires β
  {
    i = i
      + r.i;
  }
}
```

Constraints:

$\alpha_1 \in$ { this, g }
$\alpha_2 \in$ { this, g }
$\beta \subseteq$ { this, g, r }
$\alpha_1 \in \beta$
α_1[this := r, g:= α_2] $\in \beta$

Encoding:

α_1 = (b1 ? this : g)
α_2 = (b2 ? this : g)
β = { b3 ? this, b4 ? g, b5 ? r }

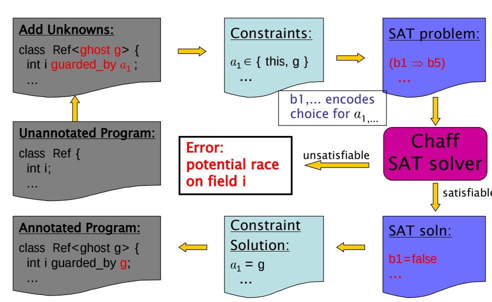
Use boolean variables b1,...,b5 to encode choices for α_1, α_2, β

Clauses:

(b1 $\Rightarrow$ b5)
($\neg$b1 $\wedge$ b2 $\Rightarrow$ b3)
($\neg$b1 $\wedge$ $\neg$b2 $\Rightarrow$ b4)

α_1[this := r, g:= α_2] $\in \beta$
(b1 ? this : g) [this := r, g:= α_2] $\in \beta$
(b1 ? r : α_2) $\in \beta$
(b1 ? r : (b2 ? this : g)) $\in$ { b3 ? this, b4 ? g, b5 ? r }

Overview of Type Inference

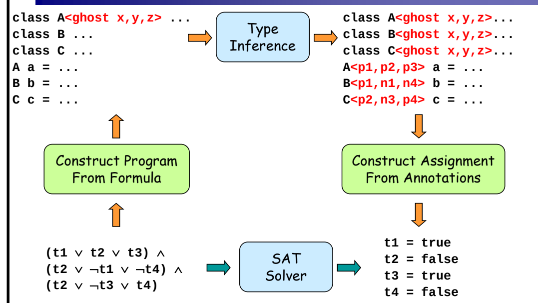


Performance

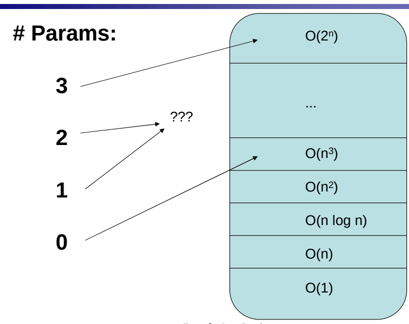
Program	Size (LOC)	Time (s)	Time/Field (s)	Number Constraints	Formula Vars	Formula Clauses
elevator	529	5.0	0.22	215	1,449	3,831
tsp	723	6.9	0.19	233	2,090	7,151
sor	687	4.5	0.15	130	562	1,205
raytracer	1,982	21.0	0.27	801	9,436	29,841
moldyn	1,408	12.6	0.12	904	4,011	10,036
montecarlo	3,674	20.7	0.19	1,097	9,003	25,974
mtrt	11,315	138.8	1.5	5,636	38,025	123,046
jbb	30,519	2,773.5	3.52	11,698	146,390	549,667

- Inferred protecting lock for 92-100% of fields
- Used preliminary read-only and escape analyses

Reducing SAT to Type Inference



Complexity of Restricted Cases



Summary

- Multithreaded heisenbugs notorious
  - race conditions, etc
- Rccjava
  - type system for race freedom
- Type inference is NP-complete
  - ghost parameters require backtracking search
- Reduce to SAT
  - adequately fast up to 30,000 LOC
  - precise: 92-100% of fields verified race free

Improved Error Reporting

```
class Ref<ghost y> {
  int c guarded_by α;
  void f1() requires y { c = 1; }
  void f2() requires y { c = 2; }
  void f3() requires this { c = 3; }
}
```

Improved Error Reporting

```
class Ref<ghost y> {
  int c guarded_by α;
  void f1() requires y { c = 1; }
  void f2() requires y { c = 2; }
  void f3() requires this { c = 3; }
}
```

Constraints
- $\alpha \in$ { y, this, no_lock }
- $\alpha \in$ { y, this }
- $\alpha \in$ { y, no_lock }
- $\alpha \in$ { y, no_lock }
- $\alpha \in$ { this, no_lock }

Improved Error Reporting

```
class Ref<ghost y> {
  int c guarded_by α;
  void f1() requires y { c = 1; }
  void f2() requires y { c = 2; }
  void f3() requires this { c = 3; }
}
```

Constraints
- $\alpha \in$ { y, this, no_lock }
- $\alpha \in$ { y, this }
- $\alpha \in$ { y, no_lock }
- $\alpha \in$ { y, no_lock }
- $\alpha \in$ { this, no_lock }

Possible Error Messages:

α = y:	Lock 'y' not held on access to 'c' in f3().

Improved Error Reporting

```
class Ref<ghost y> {
  int c guarded_by α;
  void f1() requires y { c = 1; }
  void f2() requires y { c = 2; }
  void f3() requires this { c = 3; }
}
```

Constraints
- $\alpha \in$ { y, this, no_lock }
- $\alpha \in$ { y, this }
- $\alpha \in$ { y, no_lock }
- $\alpha \in$ { y, no_lock }
- $\alpha \in$ { this, no_lock }

Possible Error Messages:

α = y:	Lock 'y' not held on access to 'c' in f3().
α = this:	Lock 'this' not held on access to 'c' in f1()&f2().

Improved Error Reporting

```
class Ref<ghost y> {
  int c guarded_by α;
  void f1() requires y { c = 1; }
  void f2() requires y { c = 2; }
  void f3() requires this { c = 3; }
}
```

Constraints
- $\alpha \in$ { y, this, no_lock }
- $\alpha \in$ { y, this }
- $\alpha \in$ { y, no_lock }
- $\alpha \in$ { y, no_lock }
- $\alpha \in$ { this, no_lock }

Possible Error Messages:

α = y:	Lock 'y' not held on access to 'c' in f3().
α = this:	Lock 'this' not held on access to 'c' in f1()&f2().
α =no_lock:	No consistent lock guarding 'c'.

Weighted Constraints

```
class Ref<ghost y> {
  int c guarded_by α;
  void f1() requires y { c = 1; }
  void f2() requires y { c = 2; }
  void f3() requires this { c = 3; }
}
```

Constraints	Weights
$\alpha \in$ { y, this, no_lock }	
$\alpha \in$ { y, this }	2
$\alpha \in$ { y, no_lock }	1
$\alpha \in$ { y, no_lock }	1
$\alpha \in$ { this, no_lock }	1

- Find solution that:
  - satisfies all un-weighted constraints, and
  - maximizes weighted sum of satisfiable weighted constraints

Weighted Constraints

```
class Ref<ghost y> {
  int c guarded_by α;
  void f1() requires y { c = 1; }
  void f2() requires y { c = 2; }
  void f3() requires this { c = 3; }
}
```

Constraints	Weights
$\alpha \in$ { y, this, no_lock }	
$\alpha \in$ { y, this }	2 ✓
$\alpha \in$ { y, no_lock }	1 ✓
$\alpha \in$ { y, no_lock }	1 ✓
$\alpha \in$ { this, no_lock }	1 X

Solution:

α = y:	Lock 'y' not held on access to 'c' in f3().

Weighted Constraints

```
class Ref<ghost y> {
  int c guarded_by α;
  void f1() requires y { c = 1; }
  void f2() requires y { c = 2; }
  void f3() requires y { c = 3; }
  void f4() requires this { c = 1; }
  void f5() requires this { c = 2; }
  void f6() requires this { c = 3; }
}
```

Constraints	Weights
$\alpha \in$ { y, this, no_lock }	
$\alpha \in$ { y, this }	2 ✗
$\alpha \in$ { y, no_lock }	1 ✓
$\alpha \in$ { y, no_lock }	1 ✓
$\alpha \in$ { y, no_lock }	1 ✓
$\alpha \in$ { this, no_lock }	1 ✓
$\alpha \in$ { this, no_lock }	1 ✓
$\alpha \in$ { this, no_lock }	1 ✓

Solution:

α =no_lock:	No consistent lock guarding 'c'.

Implementation

- Translate weighted constraints into a MAX-SAT problem
  - example:

```
(t1 ∨ t2 ∨ t3)        2
(t2 ∨ ¬t1 ∨ ¬t4)      1
(t2 ∨ ¬t3 ∨ t4)       1
(t5 ∨ ¬t1 ∨ ¬t6)
(t2 ∨ ¬t4 ∨ ¬t5)
```

  - find solution with PBS [Aloul et al 02]

Implementation

- Typical weights:
  - field access: 1
  - declaration: 2-4
- Scalability
  - MAX-SAT intractable if more than ~100 weighted clauses
  - check one field at a time (compose results)
  - only put weights on field constraints

Related Work

- Reduction
  - [Lipton 75, Lamport-Schneider 89, ...]
  - other applications:
    - type systems [Flanagan-Qadeer 03, Flanagan-Freund-Qadeer 04]
    - model checking [Stoller-Cohen 03, Flanagan-Qadeer 03]
    - dynamic analysis [Flanagan-Freund 04, Wang-Stoller 04]
- Atomicity inference
  - type and effect inference [Talpin-Jouvelot 92,...]
  - dependent types [Cardelli 88]
  - ownership, dynamic [Sastakur-Agarwal-Stoller 04]