Atomicity for Reliable Concurrent Software

Part 3a:
Types for Race-Freedom
and Atomicity

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 } ✓

replace formals this,r
by actuals x,y

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

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

Soundness Theorem:
Well-typed programs are race-free

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); }   ------> check: {g} [this:=x,r:=y, g:=m] ⊆ {m} ✓
   synchronized (m) { x.add(y); }
}
assert x.i == 6;
```


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 ↑

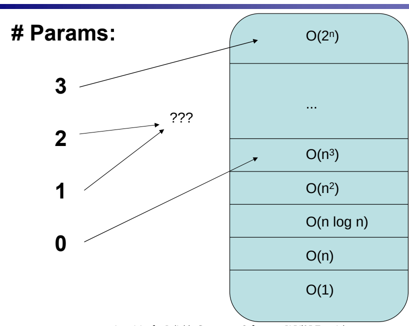
↓ Construct Assignment From Annotations ↓

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

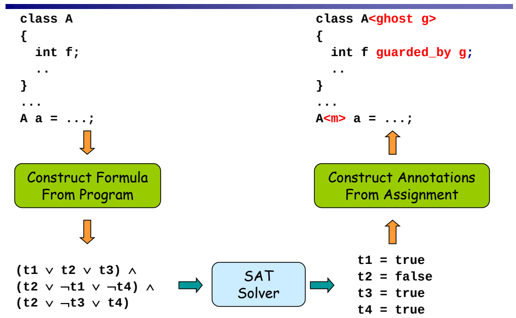
→ SAT Solver →

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

Restricted Cases



Rcc/Sat Type Inference Tool



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 α1;
  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 α1;
  void add(Ref<α2> 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 α1;
  void add(Ref<α2> 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 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] $\in \beta$

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

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 $\alpha_1, \alpha_2, \beta$

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 $\alpha_1, \alpha_2, \beta$

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 }

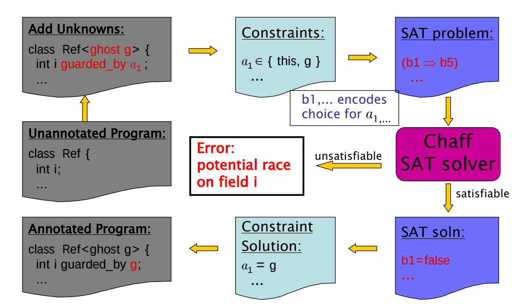
Use boolean variables b1,...,b5 to encode choices for $\alpha_1, \alpha_2, \beta$

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



Part 3a:

Types for Atomicity

Checking Atomicity

```
atomic void inc() {
  int t;
  synchronized (this) {
    t = i;
    i = t + 1;
  }
}
```

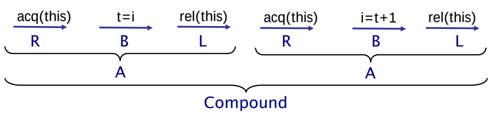
R: right-mover	lock acquire
L: left-mover	lock release
B: both-mover	race-free variable access
A: atomic	conflicting variable access

- Reducible blocks have form: (R|B)* [A] (L|B)*
- composition rules:
  - right ; mover = right
  - right ; left = atomic
  - atomic ; atomic = cmpd

Checking Atomicity (cont.)

```
~~atomic~~ void inc() {
  int t;
  synchronized (this) {
    t = i;
  }
  synchronized (this) {
    i = t + 1;
  }
}
```

R: right-mover	lock acquire
L: left-mover	lock release
B: both-mover	race-free variable access
A: atomic	conflicting variable access



java.lang.Vector

```
interface Collection {
  atomic int length();
  atomic void toArray(Object a[]);
}

class Vector {
  int count;
  Object data[];

X atomic Vector(Collection c) {
    count = c.length();              atomic
    data = new Object[count];        mover      compound
    ...
    c.toArray(data);                 atomic
  }
}
```


Conditional Atomicity

```
atomic void deposit(int n) {
  synchronized(this) {     right
    int j = bal;           mover
    bal = j + n;           mover   atomic
  }                        left
}

Xatomic void depositTwice(int n) {
  synchronized(this) {
    deposit(n);            atomic
    deposit(n);            atomic
  }
}
```


Conditional Atomicity

```
                                              if this already held
atomic void deposit(int n) {
  synchronized(this) {     right               mover
    int j = bal;           mover               mover
    bal = j + n;           mover   atomic      mover   mover
  }                        left                mover
}

atomic void depositTwice(int n) {
  synchronized(this) {
    deposit(n);            atomic
    deposit(n);            atomic
  }
}
```


Conditional Atomicity

```
(this ? mover : atomic) void deposit(int n) {
  synchronized(this) {
    int j = bal;
    bal = j + n;
  }
}

atomic void depositTwice(int n) {
  synchronized(this) {
    deposit(n);            (this ? mover : atomic)
    deposit(n);            (this ? mover : atomic)
  }
}
```


Conditional Atomicity Details

- In conditional atomicity $(x?b_1:b_2)$, x must be a final expression

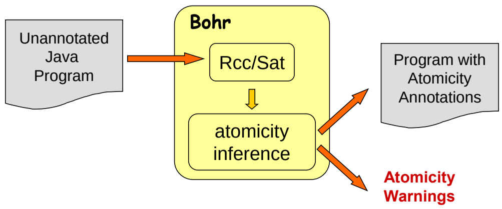
```
(x ? mover : compound) void m() {...}

atomic void mutate() {
  synchronized(x) {
    x = y;
    m();  // is m() a mover???
  }
}
```

- Composition rules

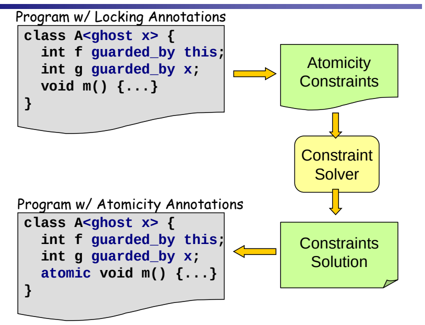
  $a ; (x?b_1:b_2) = x ? (a;b_1) : (a;b_2)$

Bohr

- Type inference for atomicity
  - finds smallest atomicity for each method



Atomicity Inference




```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {

  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

1. Add atomicity variables


```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {

  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

2. Generate constraints over atomicity variables

$s \leq \alpha_i$

Atomicity expression

s ::= const | mover | ...
| α
| $s_1 ; s_2$
| $x \,?\, s_1 : s_2$
| $S(l, s)$
| $WFA(E, s)$

3. Find assignment A


```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;        ⟶  (const; this?mover:error)
      j = this.bal + n;
    }
  }
}

class Bank {

  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```



```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;          ((const; this?mover:error);
      j = this.bal + n;    ⟶    (const; this?mover:error))
    }
  }
}

class Bank {

  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

```
class Account {
  int bal guarded_by this;

  α_1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}
```

→ S(this, ((const; this?mover:error); (const; this?mover:error)))

S(l,b): atomicity of `synchronized(l) { e }` where e has resolved atomicity b

S(l, mover)	=	l ? mover : atomic
S(l, atomic)	=	atomic
S(l, compound)	=	compound
S(l, l?b_1:b_2)	=	S(l,b_1)
S(l, m?b_1:b_2)	=	m ? S(l,b_1) : S(l,b_2) if $l \neq m$


```
class Account {
  int bal guarded_by this;

  α_1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {

  α_2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

S(this, ((const; this?mover:error); (const; this?mover:error)))

$\leq \alpha_1$


```
class Account {
  int bal guarded_by this;

  α_1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {

  α_2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

S(this, ((const; this?mover:error); (const; this?mover:error)))

$\leq \alpha_1$

replace **this** with name of receiver

(const; (this?mover:error)[this:=c])


```
class Account {
  int bal guarded_by this;

  α_1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {

  α_2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

S(this, ((const; this?mover:error); (const; this?mover:error)))

$\leq \alpha_1$

((const; c?mover:error); (const; α_1[this := c]))

Delayed Substitution


```
class Account {
  int bal guarded_by this;

  α_1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {

  α_2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

S(this, ((const; this?mover:error); (const; this?mover:error)))

$\leq \alpha_1$

S(c, ((const; c?mover:error); (const; α_1[this := c])))

$\leq \alpha_2$

Delayed Substitutions

- Given α[x := e]
  - suppose α becomes (x?mover:atomic) and e does not have const atomicity
  - then (e?mover:atomic) is not valid
- WFA(E, b) = smallest atomicity b' where
  - b ≤ b'
  - b' is well-typed and constant in E
- WFA(E, (e?mover:atomic)) = atomic

```
class Account {
  int bal guarded_by this;

  α1 void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {


  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```

S(this,
((const; this?mover:error);
(const; this?mover:error)))
$\leq \alpha_1$

S(c,
((const; c?mover:error);
(const; WFA(E, α_1[this:=c]))))
$\leq \alpha_2$

3. Compute Least Fixed Point

- Initial assignment A: $\alpha_1 = \alpha_2 =$ const
- Algorithm:
  - pick constraint $s \leq \alpha$ such that $A(s) \leq A(\alpha)$
  - set $A = A[\alpha := A(\alpha) \sqcup A(s)]$
  - repeat until quiescence

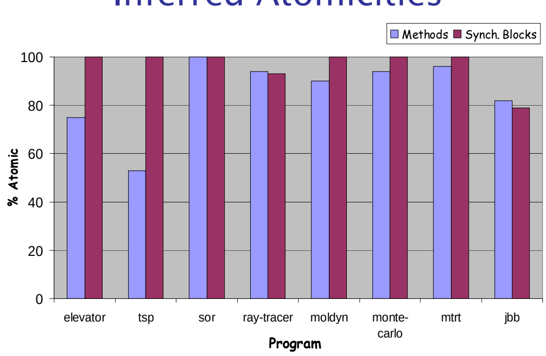

```
class Account {
  int bal guarded_by this;

  (this ? mover : atomic) void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {


  (c ? mover : atomic) void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}
```


Validation

Program	Size (KLOC)	Time (s)	Time (s/KLOC)
elevator	0.5	0.6	1.1
tsp	0.7	1.4	2.0
sor	0.7	0.8	1.2
raytracer	2.0	1.7	0.9
moldyn	1.4	4.9	3.5
montecarlo	3.7	1.5	0.4
mtrt	11.3	7.8	0.7
jbb	30.5	11.2	0.4

(excludes Rcc/Sat time)

Inferred Atomicities



Thread-Safe Classes

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]

Summary

- Type inference for rccjava is NP-complete
  - ghost parameters require backtracking search
- Reduce type inference to SAT
  - adequately fast up to 30,000 LOC
  - precise: 92-100% of fields verified race free
- Type checker and inference for atomicity
  - leverages information about race conditions
  - over 80% of methods in jbb are atomic