

Lightweight Checkers For Atomicity (Part 2)

Stephen Freund
Williams College
(on sabbatical at UCSC)

Cormac Flanagan
University of California,
Santa Cruz

Types for Race Freedom and Atomicity

- Check for interference errors statically
 - locking discipline to prevent races
 - atomicity requirements
- Reduce programmer overhead with inference
 - locking discipline
 - atomicity requirements

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 {
  sync(x,y) { x.add(y); }
  sync(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 {
  sync(x,y) { x.add(y); }
  sync(x,y) { x.add(y); }
}
assert x.i == 6;
```

check: $this \in \{this, r\}$ ✓

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 {
  sync(x,y) { x.add(y); }
  sync(x,y) { x.add(y); }
}
assert x.i == 6;
```

check: $this \in \{this, r\}$ ✓
 check: $this[this:=r] = r \in \{this, r\}$ ✓
 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 {
  sync(x,y) { x.add(y); }
  sync(x,y) { x.add(y); }
}
assert x.i == 6;
```

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

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);
parallel {
  sync(x,y) { x.add(y); } .....→ check: {this,r}[this:=x,r:=y] ⊆ {x,y} ✓
  sync(x,y) { x.add(y); } .....→ check: {this,r}[this:=x,r:=y] ⊆ {x,y} ✓
}
assert x.i == 6;

```

replace this by r

replace formals this,r by actuals x,y

Soundness Theorem:
 Well-typed programs are race-free

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 {
  sync(m) { x.add(y); }
  sync(m) { x.add(y); }
}
assert x.i == 6;

```

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 {
  sync(m) { x.add(y); }
  sync(m) { x.add(y); }
}
assert x.i == 6;

```

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

```

Type Inference

```

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)

```

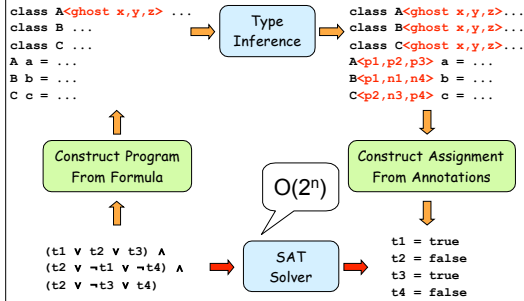
⇒ SAT Solver ⇒

```

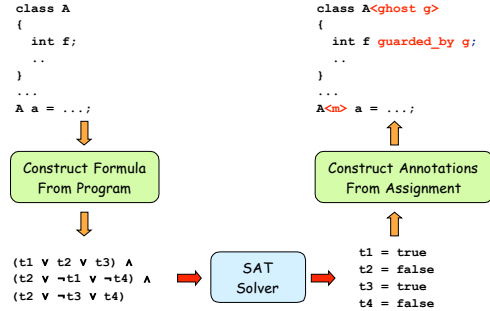
t1 = true
t2 = false
t3 = true
t4 = false

```

Reducing SAT to Type Inference



Rcc/Sat Type Inference Tool



Reducing Type Inference to SAT

```

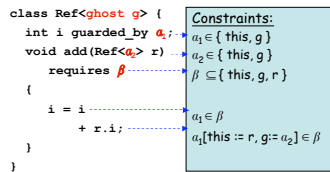
class Ref {
  int i;
  void add(Ref r)
  {
    i = i + r.i;
  }
}
    
```

Reducing Type Inference to SAT

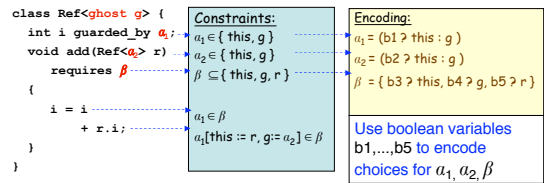
```

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

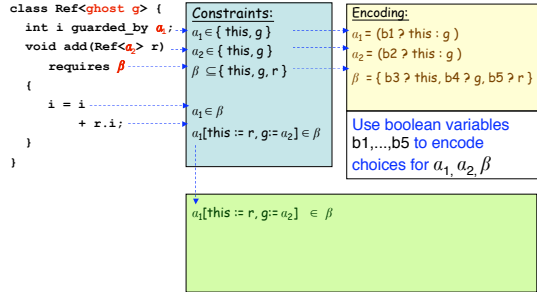
Reducing Type Inference to SAT



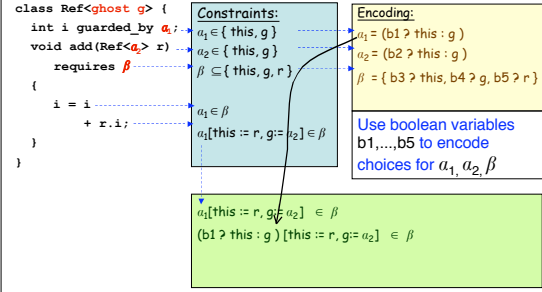
Reducing Type Inference to SAT



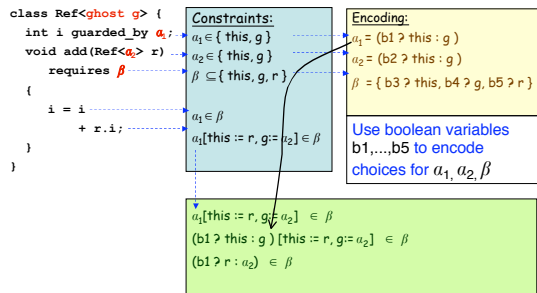
Reducing Type Inference to SAT



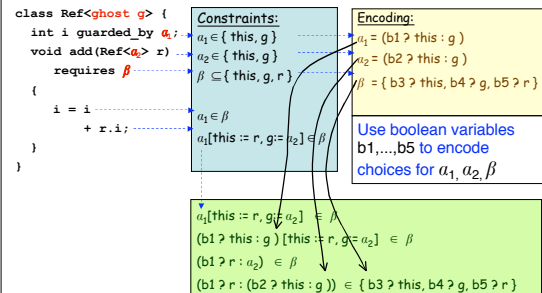
Reducing Type Inference to SAT



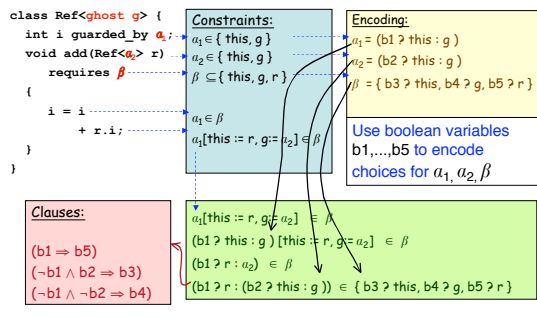
Reducing Type Inference to SAT



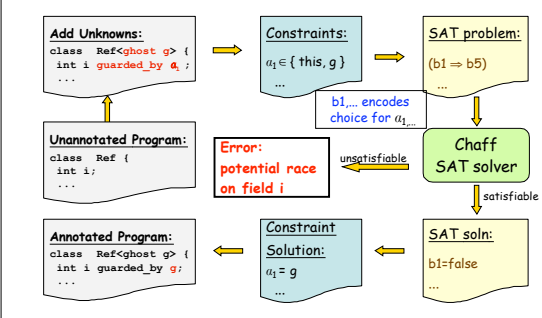
Reducing Type Inference to SAT



Reducing Type Inference to SAT



Overview of Type Inference

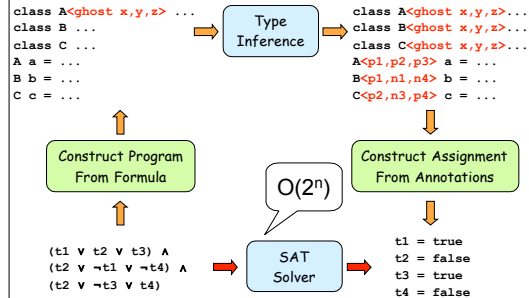


Performance

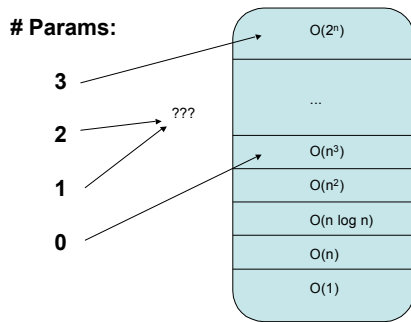
Program	Size (LOC)	Time (s)	Time/Field (s)	Number Constraints	Formula Vars	Formula Clauses
elevator	529	5.0	0.22	215	1,449	3,831
tsp	723	6.9	0.19	233	2,090	7,151
sor	687	4.5	0.15	130	562	1,205
raytracer	1,982	21.0	0.27	801	9,436	29,841
moldyn	1,408	12.6	0.12	904	4,011	10,036
montecarlo	3,674	20.7	0.19	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



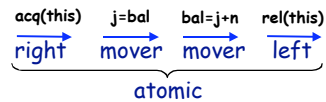
Complexity of Restricted Cases



Types for Atomicity

Code Classification

right: lock acquire
left: lock release
(both) mover: race-free variable access
atomic: conflicting variable access

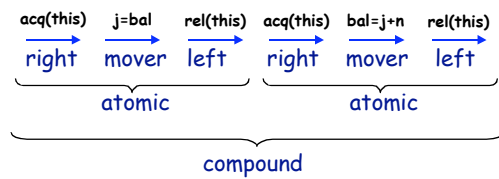


- composition rules:
right; mover = right right; left = atomic

Composing Atomicities

```

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



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();
    data = new Object[count];
    ...
    c.toArray(data);
}
```

atomic mover } compound
atomic

Conditional Atomicity

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

right mover mover left } atomic

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

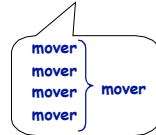
atomic atomic

Conditional Atomicity

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

right mover mover left } atomic

if this already held



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

atomic 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);
        deposit(n);
    }
}
```

(this ? mover : atomic)
(this ? mover : atomic)

Conditional Atomicity Details

- Conditional atomicity $(x?b_1:b_2)$ is well-formed only if x has **const** atomicity

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

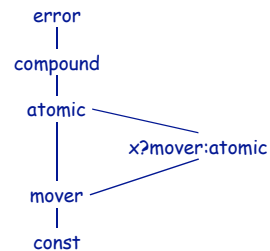
```
atomic void mutate() {
    synchronized(x) {
        x = y;
        m(); // is m() a mover???
    }
}
```

- Composition rules

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

Atomicity Details

- Partial order of *basic atomicities*

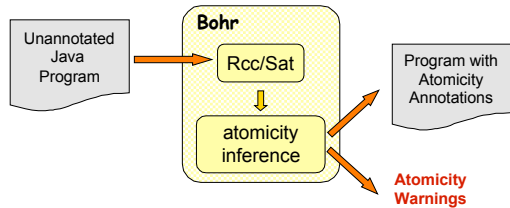


- Ordering: $<$

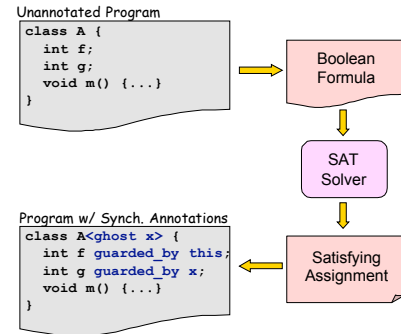
- Join operation \sqcup

Bohr

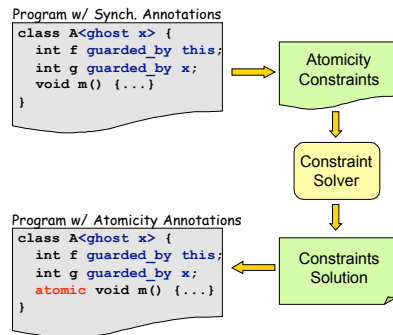
- Type inference for atomicity
 - finds smallest atomicity for each method



Rcc/Sat [Flanagan-Freund, SAS 04]



Atomicity Inference



```

class Account {
    int bal guarded_by this;

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

class Bank {

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

```

class Account {
    int bal guarded_by this;

     $\alpha_1$  void deposit(int n) {
        synchronized(this) {
            int j = this.bal;
            j = this.bal + n;
        }
    }
}

class Bank {

     $\alpha_2$  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;

     $\alpha_1$  void deposit(int n) {
        synchronized(this) {
            int j = this.bal;
            j = this.bal + n;
        }
    }
}

class Bank {

     $\alpha_2$  void double(final Account c) {
        synchronized(c) {
            int x = c.bal;
            c.deposit(x);
        }
    }
}
    
```

2. Generate constraints over atomicity variables

$s \leq \alpha_1$

Open atomicity expression

$s ::= \text{const} \mid \text{mover} \mid \dots$

3. Find assignment A:
Var \rightarrow Basic Atomicity

```

class Account {
  int bal guarded_by this;

   $\alpha_1$  void deposit(int n) {
    synchronized(this) {
      int j = this.bal;  $\longrightarrow$  (const; this?mover:error)
      j = this.bal + n;
    }
  }
}

class Bank {

   $\alpha_2$  void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}

```

```

class Account {
  int bal guarded_by this;

   $\alpha_1$  void deposit(int n) {
    synchronized(this) {
      int j = this.bal;  $\longrightarrow$  ((const; this?mover:error);
      j = this.bal + n;  $\longrightarrow$  (const; this?mover:error))
    }
  }
}

class Bank {

   $\alpha_2$  void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}

```

```

class Account {
  int bal guarded_by this;

   $\alpha_1$  void deposit(int n) {
    synchronized(this) {  $\longrightarrow$  S(this,
      int j = this.bal;  $\longrightarrow$  ((const; this?mover:error);
      j = this.bal + n;  $\longrightarrow$  (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?b1:b2) = S(l,b1)
S(l, m?b1:b2) = m ? S(l,b1) : S(l,b2) if l  $\neq$  m

```

```

class Account {
  int bal guarded_by this;

   $\alpha_1$  void deposit(int n) {
    synchronized(this) { S(this,
      int j = this.bal;  $\longrightarrow$  ((const; this?mover:error);
      j = this.bal + n;  $\longrightarrow$  (const; this?mover:error)))
    }
  }
}

class Bank {

   $\alpha_2$  void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}

```

```

class Account {
  int bal guarded_by this;

   $\alpha_1$  void deposit(int n) {
    synchronized(this) { S(this,
      int j = this.bal;  $\longrightarrow$  ((const; this?mover:error);
      j = this.bal + n;  $\longrightarrow$  (const; this?mover:error)))
    }
  }
}

class Bank {

   $\alpha_2$  void double(final Account c) {
    synchronized(c) {
      int x = c.bal;  $\longrightarrow$  (const; (this?mover:error)[this:=c])
      c.deposit(x);
    }
  }
}

```

```

class Account {
  int bal guarded_by this;

   $\alpha_1$  void deposit(int n) {
    synchronized(this) { S(this,
      int j = this.bal;  $\longrightarrow$  ((const; this?mover:error);
      j = this.bal + n;  $\longrightarrow$  (const; this?mover:error)))
    }
  }
}

class Bank {

   $\alpha_2$  void double(final Account c) {
    synchronized(c) {
      int x = c.bal;  $\longrightarrow$  ((const; c?mover:error);
      c.deposit(x);  $\longrightarrow$  (const;  $\alpha_1$ [this := c]))
    }
  }
}

```

```

class Account {
  int bal guarded_by this;

   $\alpha_1$  void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {

   $\alpha_2$  void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}

```

Delayed Substitutions

- Given $\alpha[x := f]$
 - suppose α becomes $(x?mover:atomic)$ and f does not have const atomicity in env. E
 - then $(f?mover:atomic)$ is not valid
- WFA(E, b) = smallest atomicity b' where
 - $b \leq b'$
 - b' is well-typed and constant in E
- WFA(E, (f?mover:atomic)) = atomic

```

class Account {
  int bal guarded_by this;

   $\alpha_1$  void deposit(int n) {
    synchronized(this) {
      int j = this.bal;
      j = this.bal + n;
    }
  }
}

class Bank {

   $\alpha_2$  void double(final Account c) {
    synchronized(c) {
      int x = c.bal;
      c.deposit(x);
    }
  }
}

```

3. Compute Least Fixed Point

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

```

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

S(c,
  ((const; c?mover:error);
  (const; WFA(E,  $\alpha_1$ [this:=c])))) ≤  $\alpha_2$ 

```

$A(\alpha_1) = \text{const}$

$A(\alpha_2) = \text{const}$

```

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

S(c,
  ((const; c?mover:error);
  (const; WFA(E,  $\alpha_1$ [this:=c])))) ≤  $\alpha_2$ 

```

$A(\alpha_1) = \text{const} \sqcup A(S(\text{this}, ((const; this?mover:error); (const; this?mover:error))))$

$A(\alpha_2) = \text{const}$

```

S(this,
  ((const; this?mover:error);
   (const; this?mover:error))) ≤ α1

S(c,
  ((const; c?mover:error);
   (const; WFA(E, α1[this:=c])))) ≤ α2

```

$A(\alpha_1) = \text{const} \sqcup S(\text{this}, \text{this?mover:error})$

$A(\alpha_2) = \text{const}$

```

S(this,
  ((const; this?mover:error);
   (const; this?mover:error))) ≤ α1

S(c,
  ((const; c?mover:error);
   (const; WFA(E, α1[this:=c])))) ≤ α2

```

$A(\alpha_1) = \text{const} \sqcup \text{this?mover:atomic}$

$A(\alpha_2) = \text{const}$

```

S(this,
  ((const; this?mover:error);
   (const; this?mover:error))) ≤ α1

S(c,
  ((const; c?mover:error);
   (const; WFA(E, α1[this:=c])))) ≤ α2

```

$A(\alpha_1) = \text{this?mover:atomic}$

$A(\alpha_2) = \text{const}$

```

S(this,
  ((const; this?mover:error);
   (const; this?mover:error))) ≤ α1

S(c,
  ((const; c?mover:error);
   (const; WFA(E, α1[this:=c])))) ≤ α2

```

$A(\alpha_1) = \text{this?mover:atomic}$

$A(\alpha_2) = \text{const} \sqcup$
 $A(S(c, ((const; c?mover:error);$
 $(const; WFA(E, \alpha_1[this:=c])))))$

```

S(this,
  ((const; this?mover:error);
   (const; this?mover:error))) ≤ α1

S(c,
  ((const; c?mover:error);
   (const; WFA(E, α1[this:=c])))) ≤ α2

```

$A(\alpha_1) = \text{this?mover:atomic}$

$A(\alpha_2) = \text{const} \sqcup$
 $S(c, ((const; c?mover:error);$
 $(const; WFA(E, \text{this?mover:atomic}[this:=c]))))$

```

S(this,
  ((const; this?mover:error);
   (const; this?mover:error))) ≤ α1

S(c,
  ((const; c?mover:error);
   (const; WFA(E, α1[this:=c])))) ≤ α2

```

$A(\alpha_1) = \text{this?mover:atomic}$

$A(\alpha_2) = \text{const} \sqcup c?mover:atomic$

```

S(this,
  ((const; this?mover:error);
   (const; this?mover:error))) ≤ α1

S(c,
  ((const; c?mover:error);
   (const; WFA(E, α1[this:=c]))) ≤ α2

```

$A(\alpha_1) = \text{this?mover:atomic}$

$A(\alpha_2) = \text{c?mover:atomic}$

```

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);
    }
  }
}

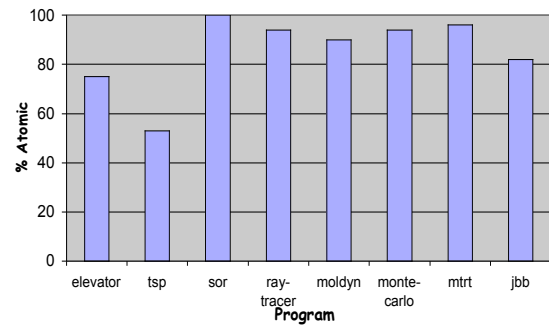
```

Validation

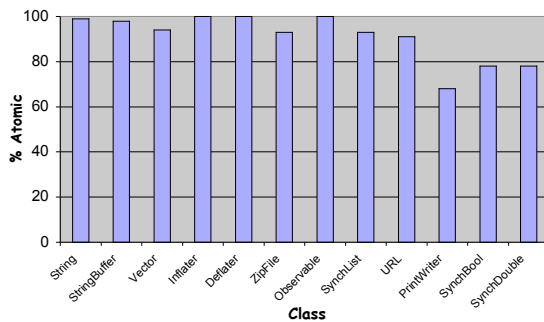
Program	Size (KLOC)	Time (s)	Time (s/KLOC)
elevator	0.5	0.6	1.1
tsp	0.7	1.4	2.0
sor	0.7	0.8	1.2
raytracer	2.0	1.7	0.9
moldyn	1.4	4.9	3.5
montecarlo	3.7	1.5	0.4
mtrt	11.3	7.8	0.7
jbb	30.5	11.2	0.4

(excludes Rcc/Sat time)

Inferred Method Atomicities



Methods in "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 (or MAX-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

Language Support for Atomicity

- Why use locks at all?
- Make `atomic` be part of the language:

```
class Account {
  int bal;
  atomic void deposit(int n) {
    int j = bal;
    bal = j + n;
  }
}
```

- Runtime system ensures atomicity is obeyed
 - locks, lock-free structures, transactions, ...
 - performance is key