Lightweight Analyses For Reliable Concurrency

Stephen Freund
Williams College

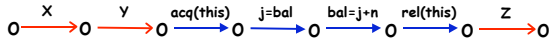
joint work with Cormac Flanagan (UCSC),
Shaz Qadeer (MSR)

Motivations for Atomicity

1. Beyond Race Conditions
2. Enables Sequential Reasoning
3. Simple Specification

Atomicity

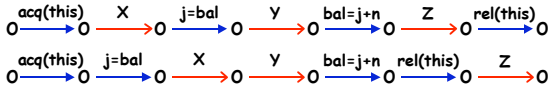
- Serialized execution of deposit



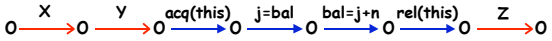

```
class Account {
  int bal;
  void deposit(int n) {
    synchronized (this) {
      int j = bal;
      bal = j + n;
    }
  }
}
```


Atomicity

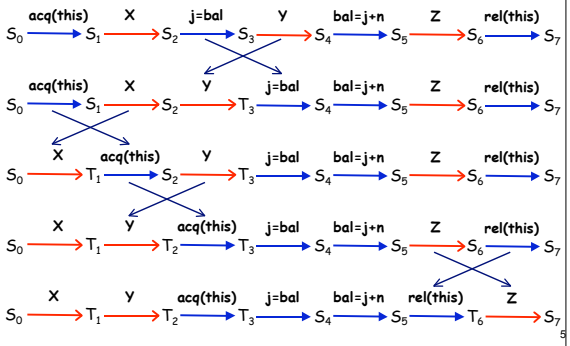
- A method is atomic if concurrent threads do not interfere with its behavior
- Guarantees that for every execution



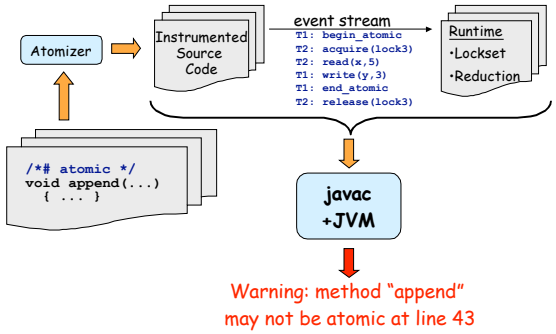
there is a serial execution with the same behavior



Reduction [Lipton 75]



Atomizer: Instrumentation Architecture



Lockset/Reduction Algorithms

- Applicable to other languages as well
  - nothing specific to Java
    - lexically scoped locking, relatively clean semantics
- Analysis -> language design
  - language / library support for feature
  - analysis tools to help programmers use feature
  - languages evolve to include ideas from analysis
    - higher level of abstractions
    - atomicity, transactions, ...

7

Atomizer Wrap Up

- Successful?
  - gained experience w/ large programs quickly
  - demonstrate atomicity is useful property
  - experiment with extensions
- Limitations of Atomizer
  - coverage, whole program
  - annotating large code bases
- Static type systems
  - modular checking
  - inferring specifications
  - computational / expressiveness issues

8

Part 2: Types for Race-Freedom and Atomicity

9

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

10

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;
```

11

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

12

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

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                  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); }
}
assert x.i == 6;
```

replace this by r

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

Lock Equality

- Type system must determine if lock is in set:
  - $r \in \{ this, r \}$
  - same as $r = this \vee r = r$
- Semantic Equivalence
  - $e_1 = e_2$ 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 semantic equivalence with syntactic equivalence
  - two locks are equal only if names are identical
  - `this` = `this`, `r.f` = `r.f`, etc.
  - lock names must also be constant

```
Object o;
int x guarded_by o;

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


Lock Equality

- 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
  - could use alias analysis

RaceFreeJava

- Concurrent extension of CLASSICJAVA [Flatt-Krishnamurthi-Felleisen 99]
- Expression typing judgement

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

Program Environment Lock set

19

Typing Rules

- Thread creation

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

- Synchronization

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

20

Field Access

$$\frac{P; E; ls \vdash e : c \\ P; E \vdash (t\ fd\ \texttt{guarded\_by}\ l) \in c \qquad \begin{array}{l} e \text{ has class } c \\ fd \text{ is declared in } c \end{array} \\ P; E \vdash [e/\texttt{this}]l \in ls \qquad \text{lock } l \text{ is held}}{P; E; ls \vdash e.fd : [e/\texttt{this}]t}$$

21

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

22

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

Iterative GFP algorithm:

- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations

23

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

Iterative GFP algorithm:

- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations

24

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

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

31

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 {
   sync(m) { x.add(y); }
   sync(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

32

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 {
   sync(m) { x.add(y); }
   sync(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

33

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;
```

- 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

34

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

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

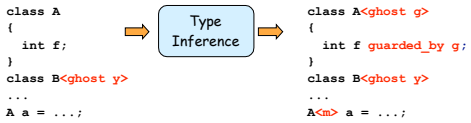
35

Type Inference with Ghosts

- HARD problem
- Iterative GFP algorithm does not work
- Check may fail because of *two* annotations
  - which should we remove?
  - no nice monotonicity properties
- Requires backtracking search

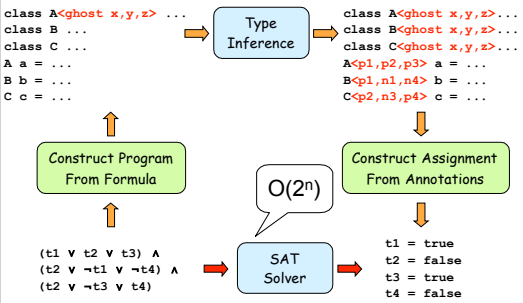
36

Type Inference With Ghosts

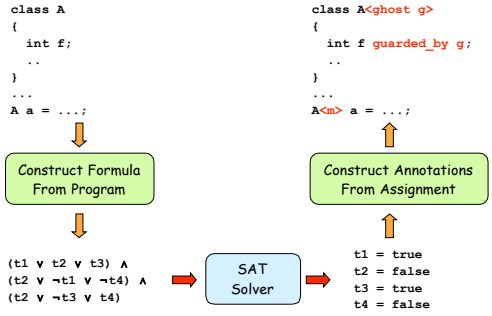


Boolean Satisfiability

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 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_1 to each field declaration
  - type inference resolves α_1 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_1 to each field declaration
  - type inference resolves α_1 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_1 to each field declaration
  - type inference resolves α_1 to some lock
- Add <α_2> to each class reference
- Add requires β_1 to each method
  - type inference resolves β_1 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:

$\alpha_1 \in \{ this, g \}$

$\alpha_2 \in \{ this, g \}$

$\beta \subseteq \{ this, g, r \}$

$\alpha_1 \in \beta$

$\alpha_1[this := r, g := \alpha_2] \in \beta$

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:

$\alpha_1 \in \{ this, g \}$

$\alpha_2 \in \{ this, g \}$

$\beta \subseteq \{ this, g, r \}$

$\alpha_1 \in \beta$

$\alpha_1[this := r, g := \alpha_2] \in \beta$

Encoding:

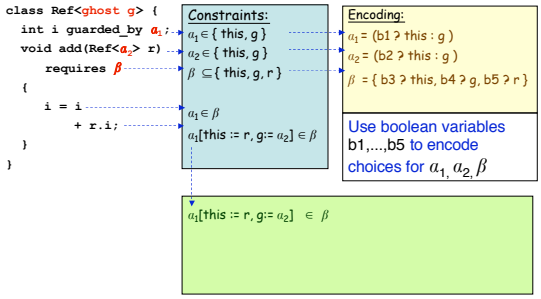
$\alpha_1 = (b1 \; ? \; this : g)$

$\alpha_2 = (b2 \; ? \; this : g)$

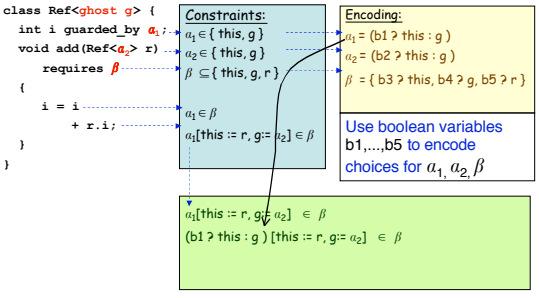
$\beta = \{ b3 \; ? \; this, b4 \; ? \; g, b5 \; ? \; r \}$

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

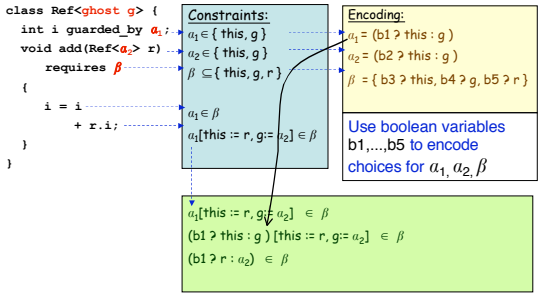
Reducing Type Inference to SAT



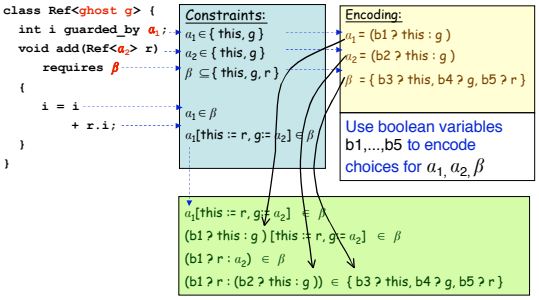
Reducing Type Inference to SAT



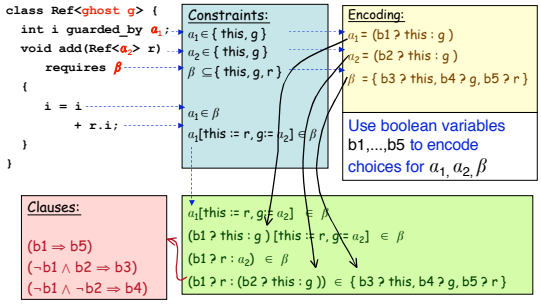
Reducing Type Inference to SAT



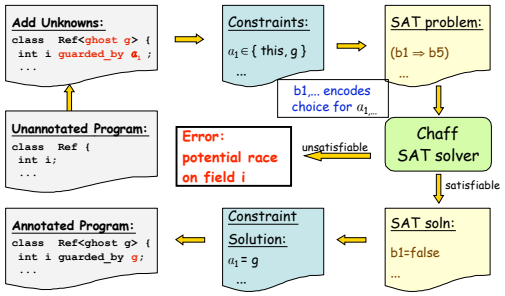
Reducing Type Inference to SAT



Reducing Type Inference to SAT



Overview of Type Inference



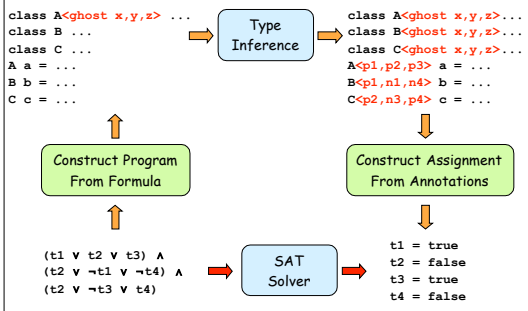
Performance

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

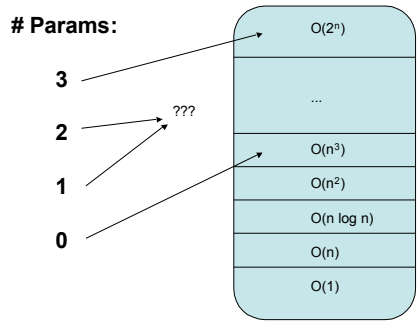
55

Reducing SAT to Type Inference



56

Complexity of Restricted Cases



57

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; }
}
```

58

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
- $\alpha \in \{ y, this, no\_lock \}$
- $\alpha \in \{ y, this \}$
- $\alpha \in \{ y, no\_lock \}$
- $\alpha \in \{ y, no\_lock \}$
- $\alpha \in \{ this, no\_lock \}$

59

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().

60

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 X
$\alpha \in$ { 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

67

Summary

- Leverage existing SAT solvers
  - easy to use
  - can validate ideas quickly
  - may be fast enough
    - even if original problem is not in NP
- Reporting good errors is important
  - can be hard to do for constraint solvers
  - type inference as an optimization problem
  - simple weighting scheme
- Exploring special purpose constraint solvers

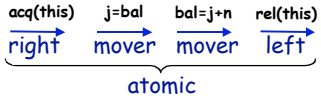
68

Types for Atomicity

69

Code Classification

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



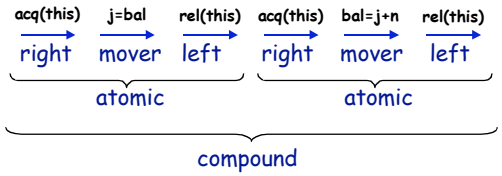
- composition rules:

  right; mover = right right; left = atomic

70

Composing Atomicities

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



71

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
  }
}
```

72

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
    int j = bal;            mover
    bal = j + n;            mover   } atomic
  }                         left
}

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

mover, mover, mover, mover } mover

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)
  }
}
```


Atomicity Details

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

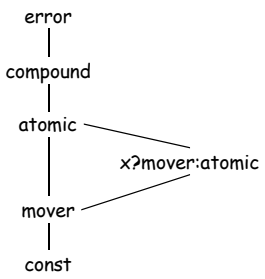
```
(x ? mover : compound) void m() {...}

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*



Standard Operations

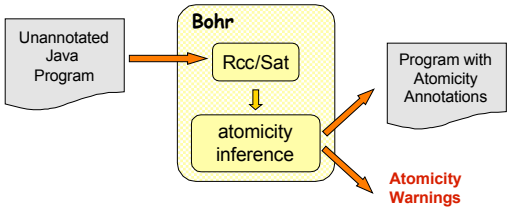
- **Join operation:** ⊔

```
void mutate() {
  if (test) {   mover
    m();        x?mover:atomic
  } else {
    n();        mover
  }
}
```

mover;
(x?mover:atomic ⊔ mover)
= x?mover:atomic

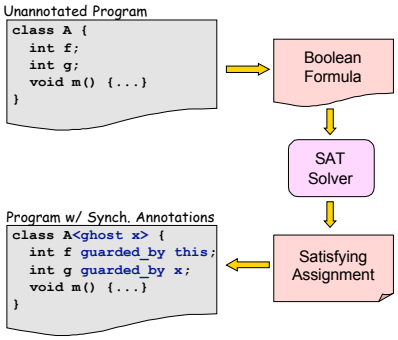
Bohr

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



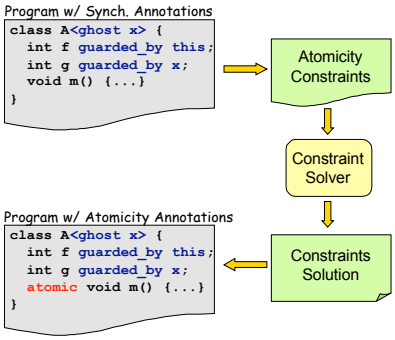
79

Rcc/Sat [Flanagan-Freund, SAS 04]



80

Atomicity Inference



81

```
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);
    }
  }
}
```

82

```
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

83

```
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

$s \leq \alpha_i$

Open atomicity expression

$s ::= const \mid mover \mid \ldots$
$\mid x\ ?\ s_1 : s_2$
$\mid \alpha$
$\mid s_1 ; s_2$
$\mid S(l, s)$
$\mid WFA(E, s)$

3. Find assignment A:
   Var → Basic Atomicity

84

```
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);
    }
  }
}
```

85

```
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);
    }
  }
}
```

86

```
class Account {
  int bal guarded_by this;

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

87

```
class Account {
  int bal guarded_by this;

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

class Bank {


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

88

```
class Account {
  int bal guarded_by this;

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

class Bank {


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

replace `this` with name of receiver

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

89

```
class Account {
  int bal guarded_by this;

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

class Bank {


  α2 void double(final Account c) {
    synchronized(c) {
      int x = c.bal;        ((const; c?mover:error);
      c.deposit(x);         (const; α1[this := c]))
    }                              Delayed
  }                                Substitution
}
```

90

```
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))) ≤ α_1

S(c, ((const; c?mover:error); (const; α_1[this := c]))) ≤ α_2

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 ≤ b'**
  - **b' is well-typed and constant in E**
- WFA(E, (f?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))) ≤ α_1

S(c, ((const; c?mover:error); (const; WFA(E, α_1[this:=c])))) ≤ α_2

3. Compute Least Fixed Point

- Initial assignment A: $\alpha_1 = \alpha_2 =$ const
- Algorithm:
  - pick constraint $s \le \alpha$ 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))) ≤ α_1

S(c, ((const; c?mover:error); (const; WFA(E, α_1[this:=c])))) ≤ α_2

$A(\alpha_1)$ = const

$A(\alpha_2)$ = const

S(this, ((const; this?mover:error); (const; this?mover:error))) ≤ α_1

S(c, ((const; c?mover:error); (const; WFA(E, α_1[this:=c])))) ≤ α_2

$A(\alpha_1)$ = const ⊔ A(S(this, ((const; this?mover:error); (const; this?mover:error))))

$A(\alpha_2)$ = const

S(this,
((const; this?mover:error);
(const; this?mover:error))) ≤ α_1

S(c,
((const; c?mover:error);
(const; WFA(E, α_1[this:=c]))))) ≤ α_2

$A(\alpha_1)$ = const ⊔ S(this, this?mover:error)

$A(\alpha_2)$ = const

S(this,
((const; this?mover:error);
(const; this?mover:error))) ≤ α_1

S(c,
((const; c?mover:error);
(const; WFA(E, α_1[this:=c]))))) ≤ α_2

$A(\alpha_1)$ = const ⊔ this?mover:atomic

$A(\alpha_2)$ = const

S(this,
((const; this?mover:error);
(const; this?mover:error))) ≤ α_1

S(c,
((const; c?mover:error);
(const; WFA(E, α_1[this:=c]))))) ≤ α_2

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

$A(\alpha_2)$ = const

S(this,
((const; this?mover:error);
(const; this?mover:error))) ≤ α_1

S(c,
((const; c?mover:error);
(const; WFA(E, α_1[this:=c]))))) ≤ α_2

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

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

S(this,
((const; this?mover:error);
(const; this?mover:error))) ≤ α_1

S(c,
((const; c?mover:error);
(const; WFA(E, α_1[this:=c]))))) ≤ α_2

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

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

S(this,
((const; this?mover:error);
(const; this?mover:error))) ≤ α_1

S(c,
((const; c?mover:error);
(const; WFA(E, α_1[this:=c]))))) ≤ α_2

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

$A(\alpha_2)$ = const ⊔ c?mover:atomic

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])))) $\leq \alpha_2$

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

$A(\alpha_2)$ = 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);
    }
  }
}
```


Layered Abstractions

```
class Set {
  List elems;

  void add(Object o) {
    synchronized(this) {
      if (!elems.contains(o)) elems.add(o);
    }
  }
}

class List {

  boolean contains(Object o) { synchronized(this) ... }


  void add(Object o) { synchronized(this) ... }
}
```


Protecting Locks

```
class Set {
  List<this> elems;

  void add(Object o) {
    synchronized(this) {
      if (!elems.contains(o)) elems.add(o);
    }
  }
}

class List<protected_by x> {

  boolean contains(Object o) { synchronized(this) ... }


  void add(Object o) { synchronized(this) ... }
}
```

lock for Set held whenever lock for elems is acquired

parameter x is protecting lock

Protecting Locks

```
class Queue {
  List<none> elems;

  void enqueue(Object o) {
    elems.add(o);
  }

  Object dequeue() {
    return elems.remove(0);
  }
}
```

no lock is held on all acquires of the List lock

Generalized Lock Predicates

```
class Set {
  List<this> elems;

  (this ? mover : atomic) void add(Object o) {
    synchronized(this) {
      if (!elems.contains(o)) elems.add(o);
    }
  }
}

class List<protected_by x> {
  (x == none) ? (this?mover:atomic) : (x?mover:error)
  boolean contains(Object o) { synchronized(this) ... }

  (x == none) ? (this?mover:atomic) : (x?mover:error)
  void add(Object o) { synchronized(this) ... }
}
```

Bohr Implementation

- Full Java programming language
  - inheritance, subtyping, interfaces
  - inner classes, static fields and methods
- Extensions for common Java idioms
  - redundant synchronization
  - thread-local data
  - List class with unsynchronized and internally-synchronized subclasses
- Supports manual annotations
  - `/*# m ? mover : error */, /*# no_warn */`

109

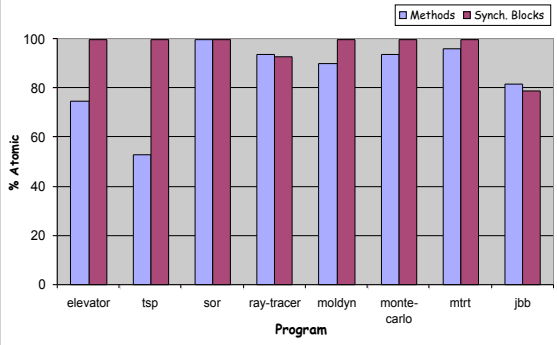
Validation

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)

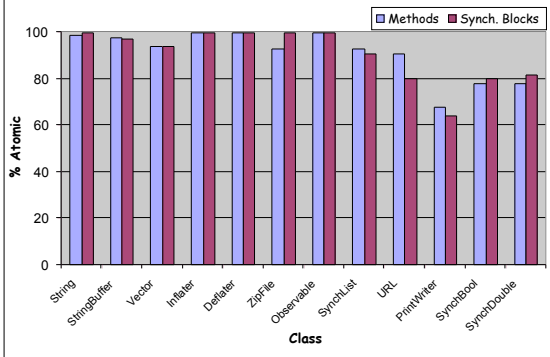
110

Inferred Atomicities



111

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]

113

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

114

alloc

```
boolean b[MAX]; // b[i]==true iff block i is free
Lock m[MAX];

// return index of newly allocated resource
//  or -1 if none found
atomic int alloc() {
   int i = 0;
   while (i < MAX) {
     acquire(m[i]);
     if (b[i]) {
       b[i] = false;
       release(m[i]);
       return i;
     }
     release(m[i]);
     i++;
   }
   return -1;
}
```

115