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

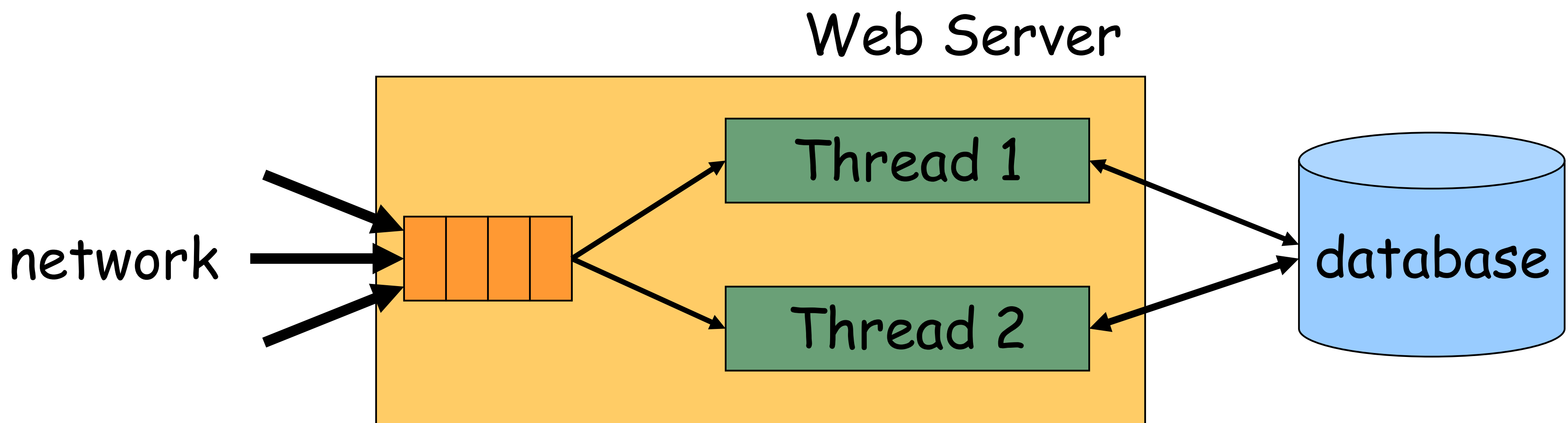
- Transistors per chip doubles every 18 months
- Single-threaded performance doubles every 2 years
  - faster clocks, deep pipelines, multiple issue
  - wonderful!

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

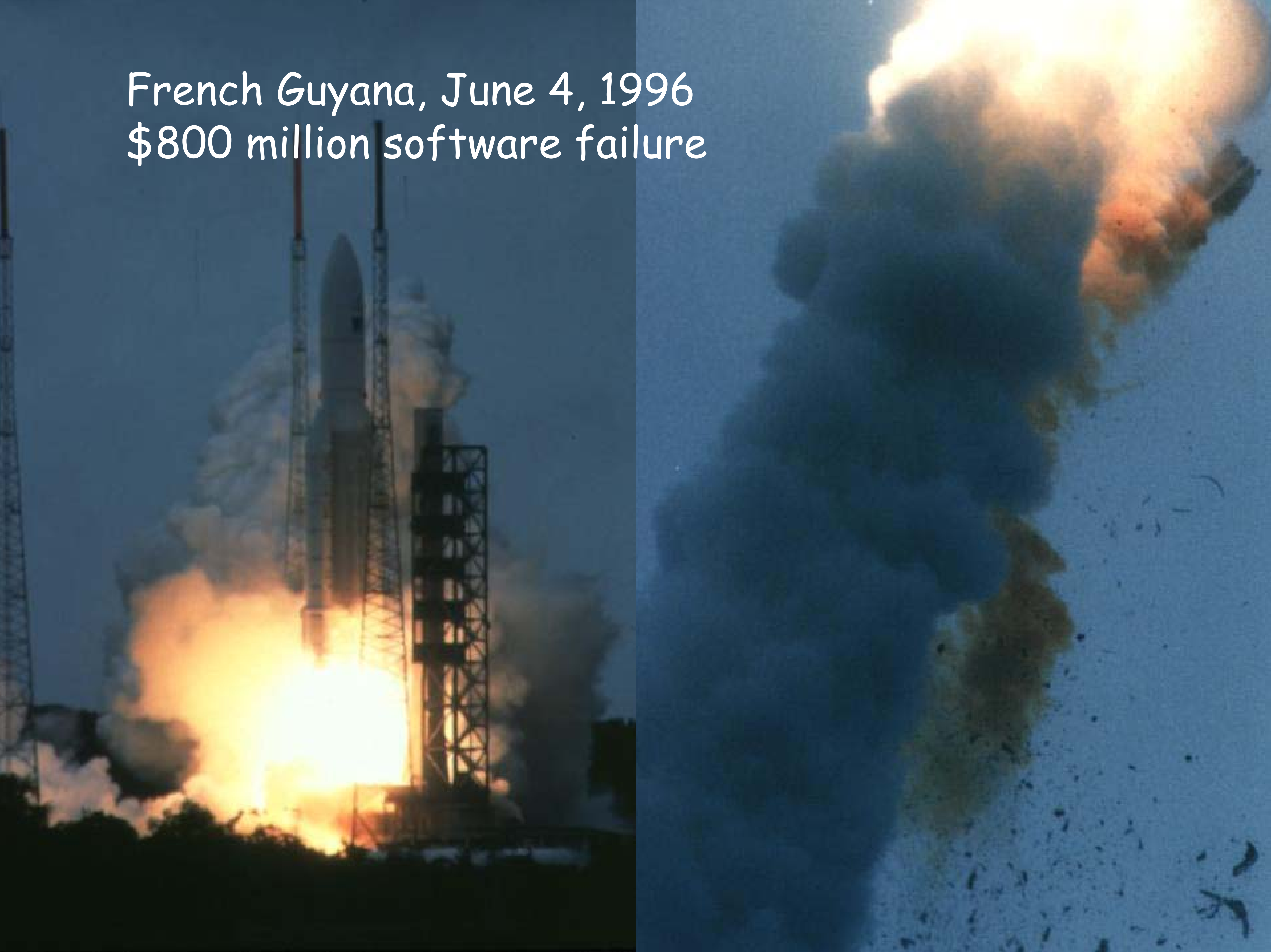
Programming With Threads

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



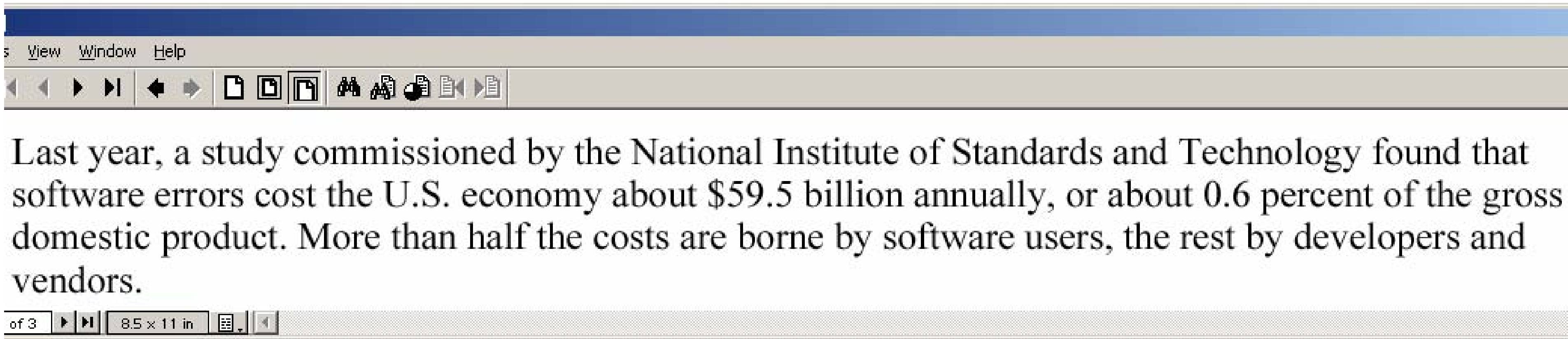
```
*** STOP: 0x00000019 (0x00000000,0xC00E0FF0,0xFFFFEFD4,0xC0000000)
BAD_POOL_HEADER

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

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

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

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

More BSOD Embarrassments


French Guyana, June 4, 1996
$800 million software failure

Economic Impact

- NIST study



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

Non-Determinism, Heisenbugs

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

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

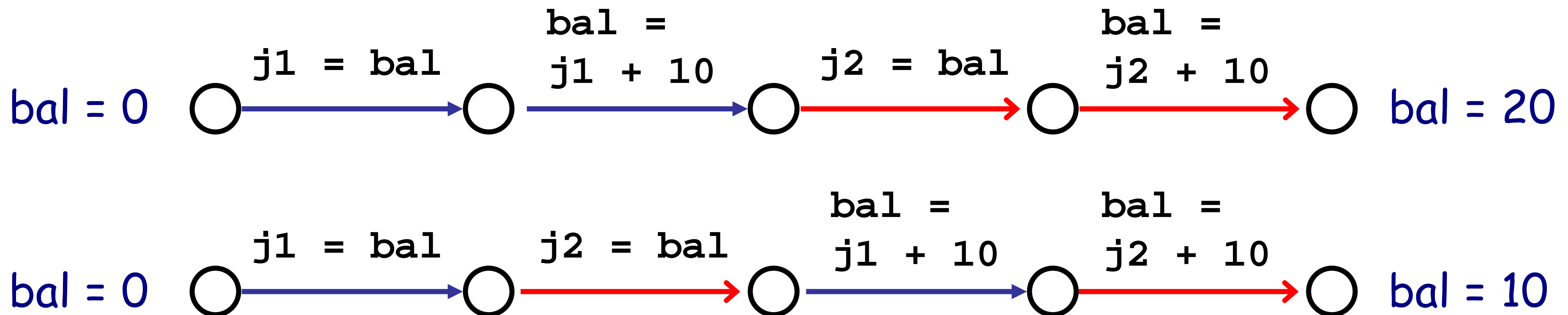
400 horses
100 microprocessors

Bank Account Implementation

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

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

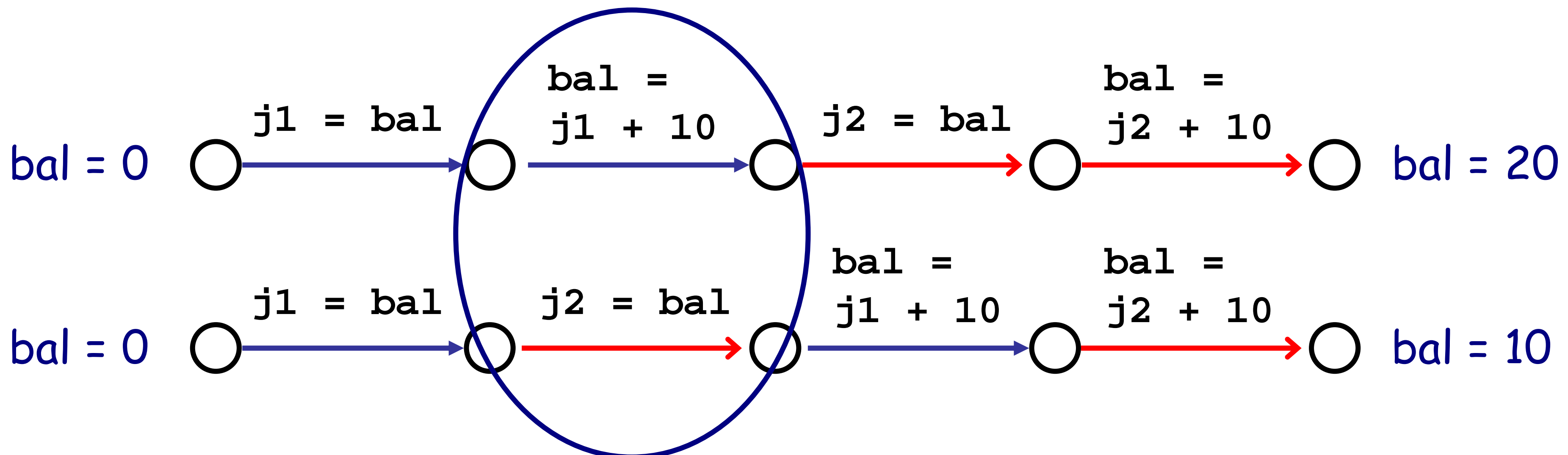
Bank Account Implementation

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

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



Bank Account Implementation

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



Race Conditions

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

Race Conditions

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

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

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

assert x.i == 6;
```

Race Conditions

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

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

Race condition on x.i

Assertion may fail

Lock-Based Synchronization

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

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

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

When Locking Goes Bad ...

- Hesienbugs (race conditions, etc) are common and problematic
  - forget to acquire lock, acquire wrong lock, etc
  - extremely hard to detect and isolate

- Traditional type systems are great for catching certain errors

- *Type systems for multithreaded software*
  - detect race conditions, atomicity violations, ...

Verifying Race Freedom with Types

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

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

Verifying Race Freedom with Types

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

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

check: $this \in \{ this, r \}$ ✓

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 {
   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: $this \in \{ this, r \}$ ✓

check: $this[this{:=}r] = r \in \{ this, r \}$ ✓

replace formals this,r by actuals x,y

check: $\{this,r\}[this{:=}x,r{:=}y] \subseteq \{ x, y \}$ ✓

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: $this \in \{ this, r \}$ ✓

check: $this[this:=r] = r \in \{ this, r \}$ ✓

replace formals this,r
by actuals x,y

check: $\{this,r\}[this:=x,r:=y] \subseteq \{ x, y \}$ ✓

check: $\{this,r\}[this:=x,r:=y] \subseteq \{ x, y \}$ ✓

Soundness Theorem:
Well-typed programs are race-free

One Problem ...

```
Object o;
int x guarded_by o;

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

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

- Lock expressions must be constant

Lock Equality

- Type system checks if lock is in lock set
  - $r \in \{ this, r \}$
  - same as $r = this \lor r = r$

- 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(); }
```

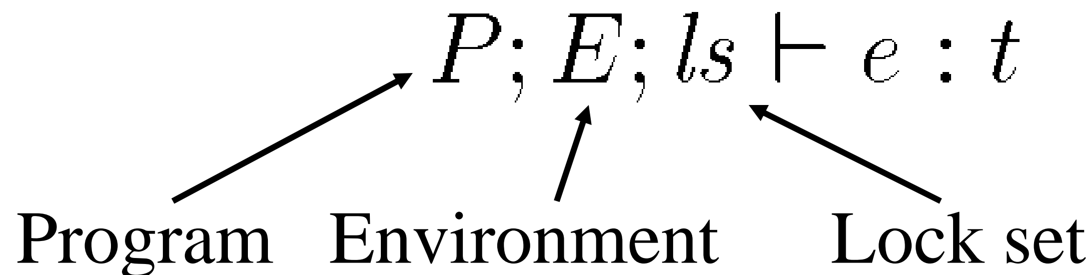
$this[this:=p] = p \in \{ q \}$ X

- Not a major source of imprecision

RaceFreeJava

- Concurrent extension of CLASSICJAVA [Flatt-Krishnamurthi-Felleisen 99]

- Judgement for typing expressions



Typing Rules

- Thread creation

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

- Synchronization

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

Field Access

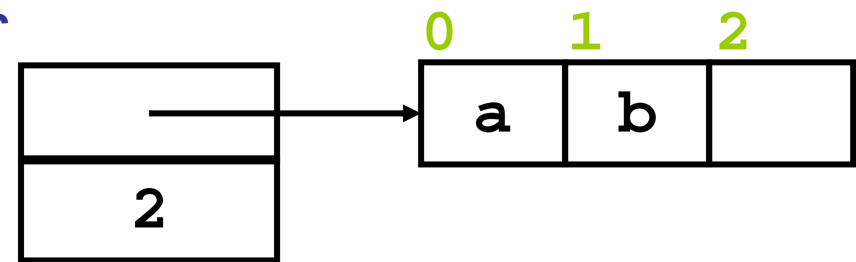
$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash (t\ fd\ \texttt{guarded_by}\ l) \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \end{array}}{P; E; ls \vdash e.fd : [e/\texttt{this}]t}$$

e has class *c*

fd is declared in *c*

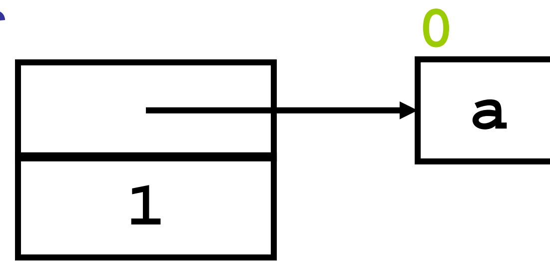
lock *l* is held

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

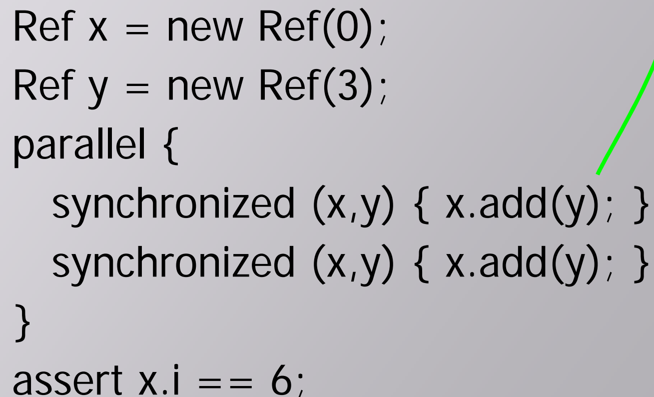
```
static final Object m =new Object();

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;
  }
}
```

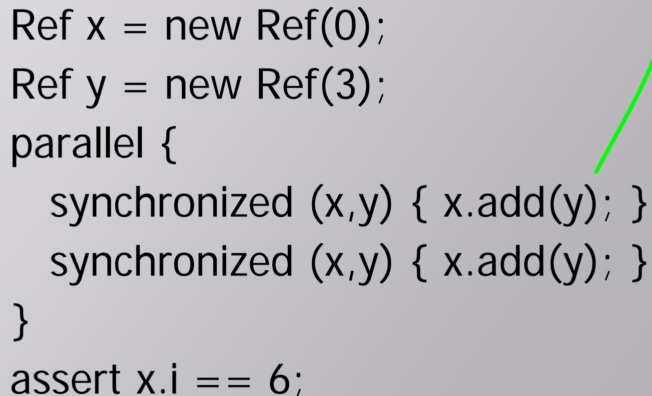


Iterative GFP algorithm:

- [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;
```

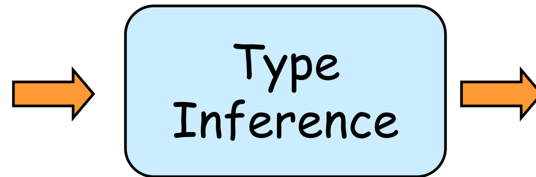
check: {g} [this:=x,r:=y, g:=m] ⊆ {m} ✓

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

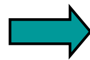
```
class A
{
  int f;
}
class B<ghost y>
...
A a = ...;
```

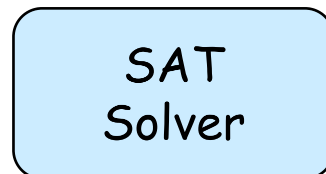



```
class A<ghost g>
{
  int f guarded_by g;
}
class B<ghost y>
...
A<m> a = ...;
```

Boolean Satisfiability

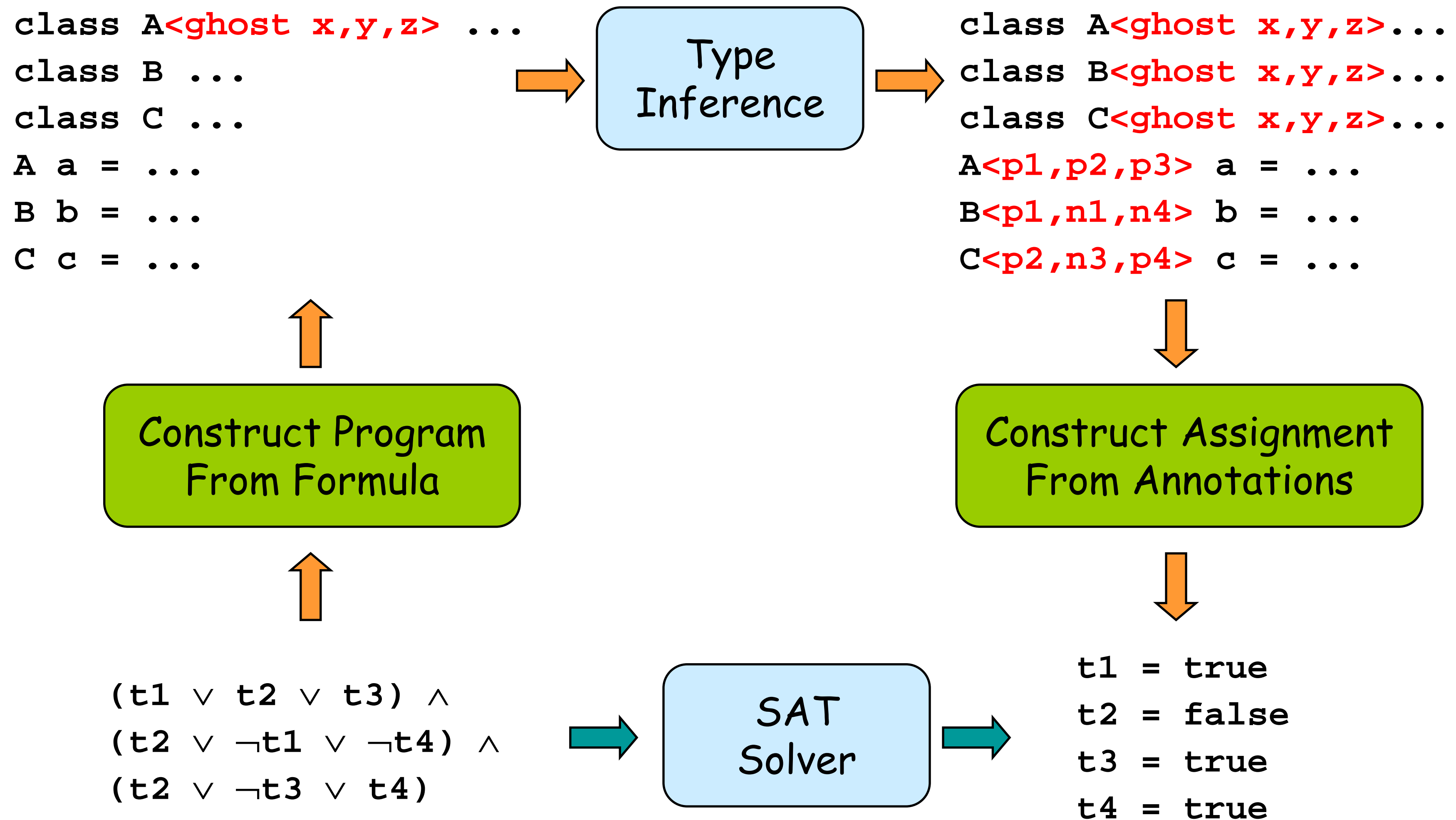
$(t1 \vee t2 \vee t3) \wedge$
$(t2 \vee \neg t1 \vee \neg t4) \wedge$
$(t2 \vee \neg t3 \vee t4)$





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

Reducing SAT to Type Inference



Rcc/Sat Type Inference Tool

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

Construct Formula From Program

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

SAT Solver

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

Construct Annotations From Assignment

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

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 $<\alpha_2>$ 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 $<\alpha_2>$ 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:

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

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

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

$\alpha_1 \in \beta$

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

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:

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

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

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

$\alpha_1 \in \beta$

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

Encoding:

$\alpha_1 = (b1 \ ? \ \text{this} : g)$

$\alpha_2 = (b2 \ ? \ \text{this} : g)$

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

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

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:

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

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

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

$\alpha_1 \in \beta$

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

Encoding:

$\alpha_1 = (b1 \ ? \ \text{this} : g)$

$\alpha_2 = (b2 \ ? \ \text{this} : g)$

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

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

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

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 $\alpha_1, \alpha_2, \beta$

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

$(b1 \; ? \; this : g)[this := r, g := \alpha_2] \in \beta$

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:

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

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

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

$\alpha_1 \in \beta$

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

Encoding:

$\alpha_1 = (b1 \ ? \ \text{this} : g)$

$\alpha_2 = (b2 \ ? \ \text{this} : g)$

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

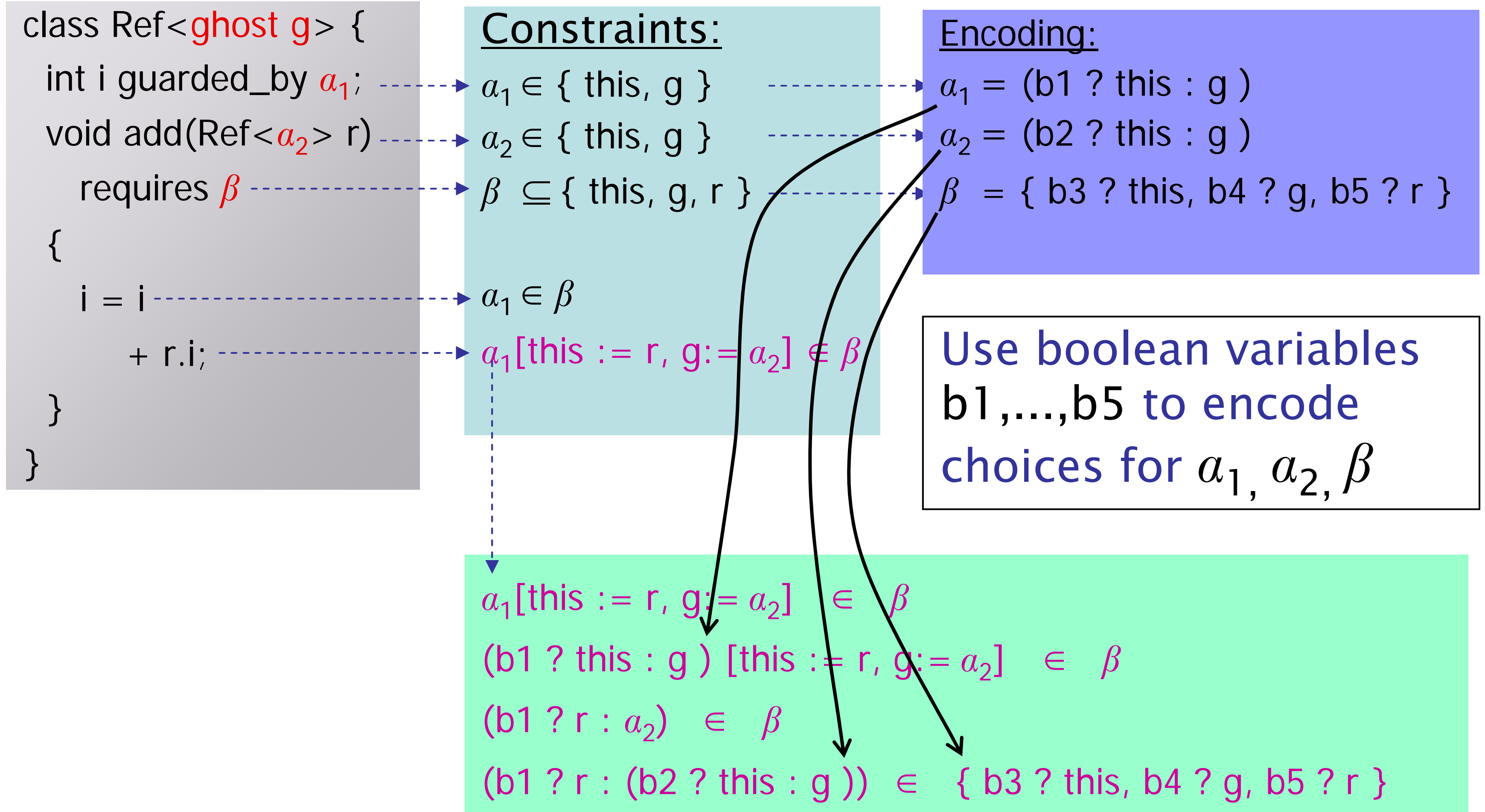
Use boolean variables b1,...,b5 to encode choices for $\alpha_1, \alpha_2, \beta$

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

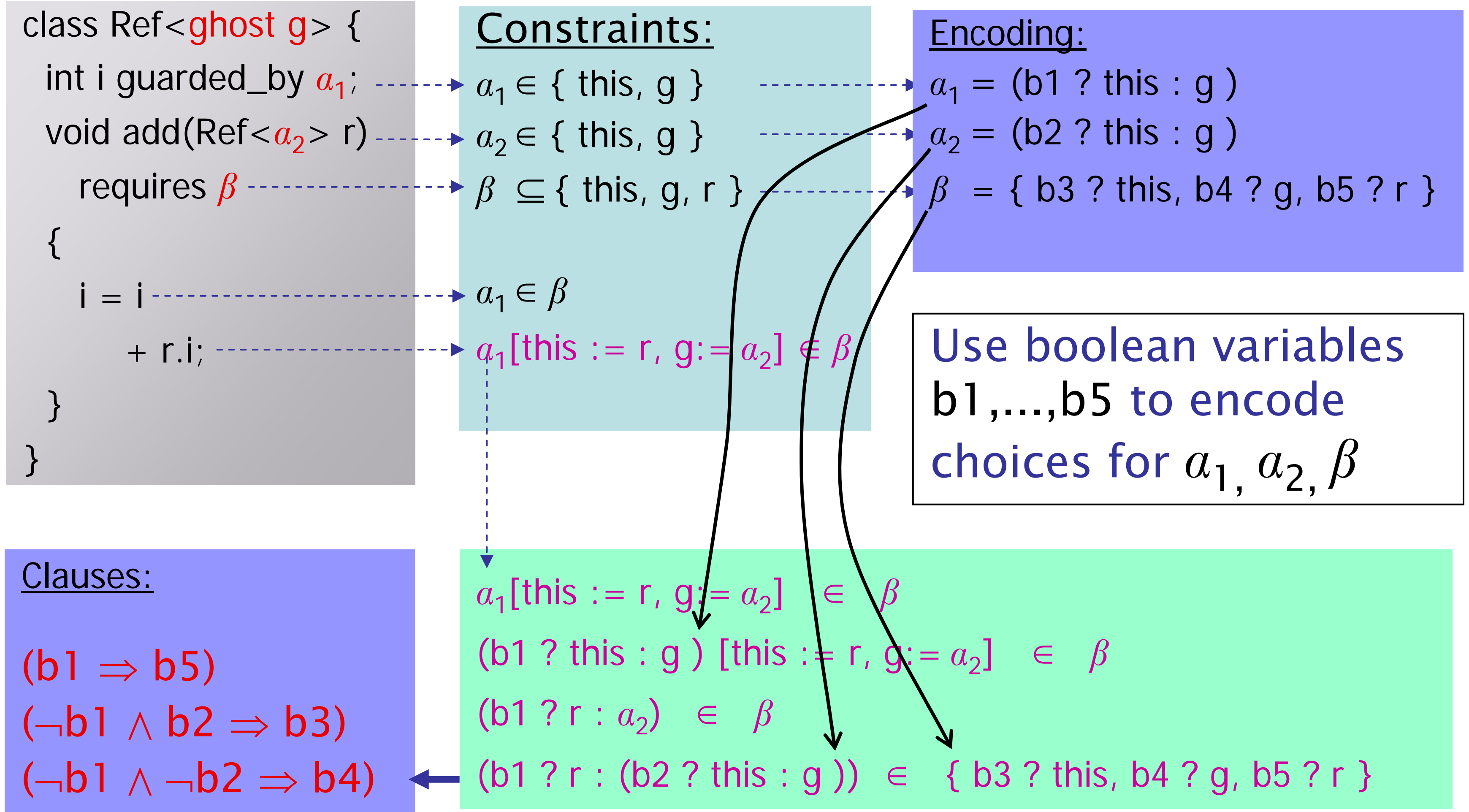
$(b1 \ ? \ \text{this} : g)[\text{this} := r, g := \alpha_2] \in \beta$

$(b1 \ ? \ r : \alpha_2) \in \beta$

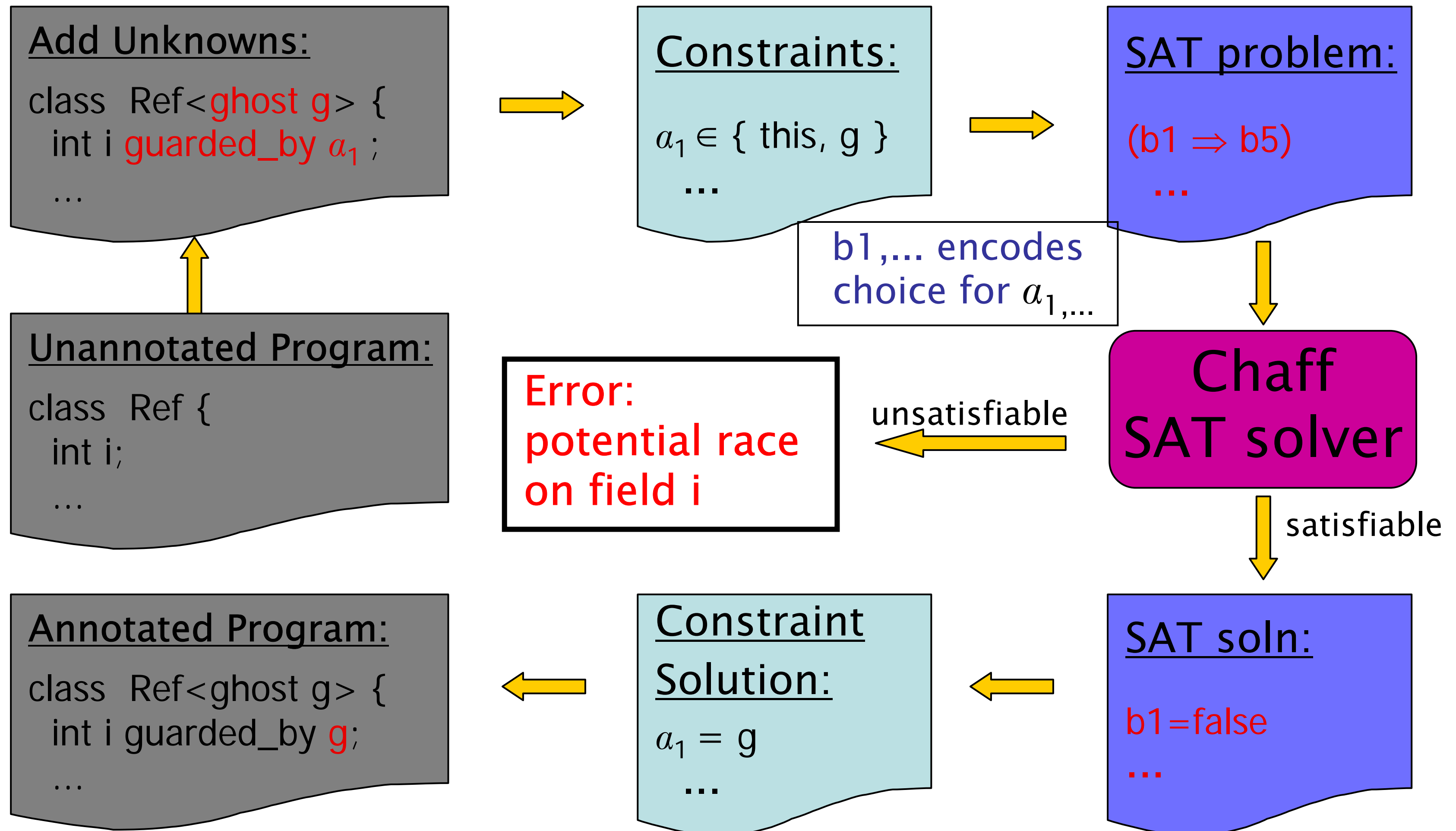
Reducing Type Inference to SAT



Reducing Type Inference to SAT



Overview of Type Inference

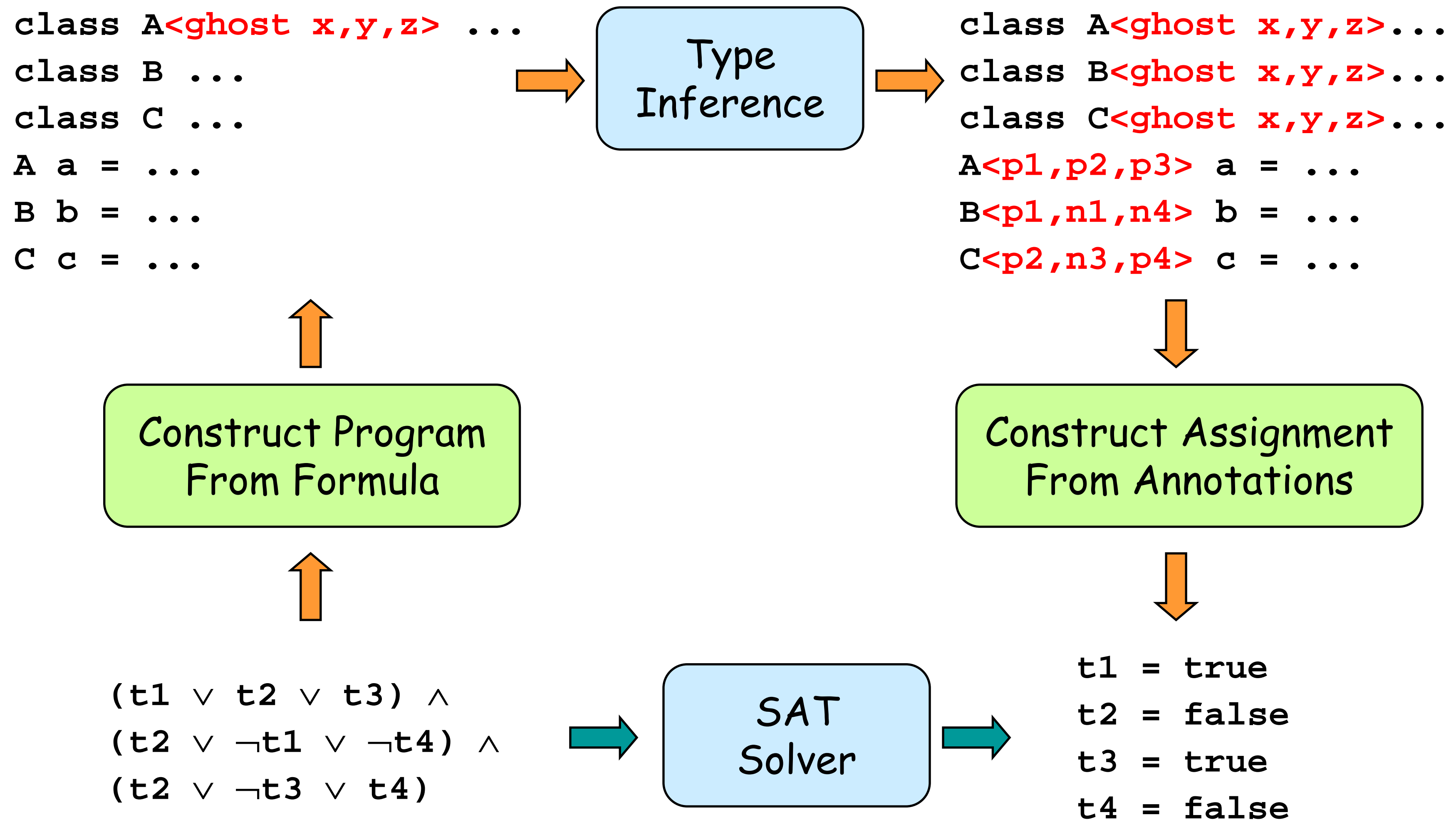


Performance

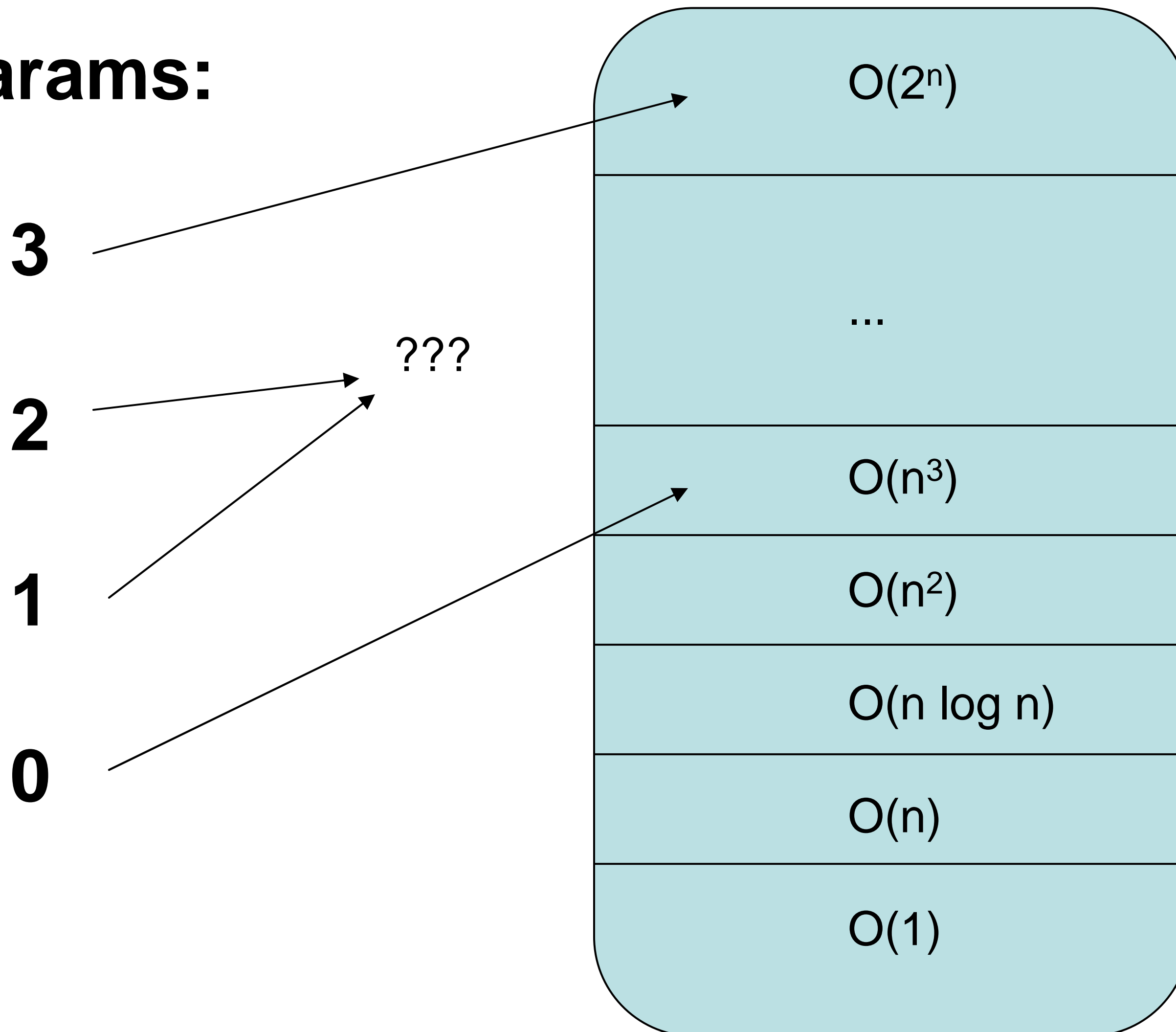
Program	Size (LOC)	Time (s)	Time/Field (s)	Number Constraints	Formula Vars	Formula Clauses
elevator	529	5.0	0.22	215	1,449	3,831
tsp	723	6.9	0.19	233	2,090	7,151
sor	687	4.5	0.15	130	562	1,205
raytracer	1,982	21.0	0.27	801	9,436	29,841
moldyn	1,408	12.6	0.12	904	4,011	10,036
montecarlo	3,674	20.7	0.19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



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:

$\alpha = y$: 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:

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

$\alpha = this$: 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:

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

$\alpha = $ 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:

$(t1 \vee t2 \vee t3)$	2
$(t2 \vee \neg t1 \vee \neg t4)$	1
$(t2 \vee \neg t3 \vee t4)$	1
$(t5 \vee \neg t1 \vee \neg t6)$	
$(t2 \vee \neg t4 \vee \neg t5)$	

  - 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]