Types for Data Races
[redacted slides]

Adam Chlipala
MIT

[a small modification of Armando Solar-Lezama's slides, which were adapted from Flanagan and Freund]

6.820
October 9, 2013
Related Reading


[But don't worry too much about mastering this topic from the standpoint of 6.820. Probably only read the paper if you're curious to go beyond the scope of the class.]
Recap

A change in a private input cannot affect a public output.

Data with a label $L_h$ cannot be written to a location with label $L_l$ if $L_l \leq L_h$.

Wikipedia

```java
Wikipedia wp = getWP();
wp.write(rx);
Wikipedia{
    void write(String[] txt);
}
```
Data Races

class Account {
    private int bal = 0;

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

Data Race:
Two threads access the same memory location,
one of the accesses is a write,
and there is no synchronization in between.
Strategy

How do programmers avoid races?
- Only access shared data while holding the “right” lock.
- All threads must agree on what the right lock for a piece of data is.
- The decision of what the right lock is should be easy to describe.
- Otherwise, it’s easy to get confused!

We can make this into a safety policy!
Strategy

In order to avoid races, we will design a type system to enforce the following safety property:

- When a memory location L is accessed by a thread, the set of locks held by the thread must be a superset of the set of locks that protect L.

Challenges:
The language

Start with a simple language with classes and references:

\[ e ::= \text{new } c \quad \text{(allocate)} \]
\[ x \quad \text{(variable)} \]
\[ e.f_d \quad \text{(field access)} \]
\[ e.f_d = e \quad \text{(field update)} \]
\[ e.mn(e^*) \quad \text{(method call)} \]
\[ \text{let } arg = e \text{ in } e \quad \text{(variable binding)} \]

Add threads and synchronization:

\[ \text{synchronized } e \text{ in } e \quad \text{(synchronization)} \]
\[ \text{fork } e \quad \text{(fork)} \]
Java synchronization

Every object has a lock associated with it. A synchronized block acquires and releases the lock of an object.
Stating Locking Requirements

class Account {
    private int bal guarded_by this = 0;

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

    public void transferAll(Account r) {
        int j = bal;
        int k = r.bal;
        bal = j+k;
        r.bal = 0;
    }
}
Stating Locking Requirements

class Account {
    private Guard g;
    private int bal guarded_by g = 0;

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

    public void transferAll(Account r) requires g, r.g {
        int j = bal;
        int k = r.bal;
        bal = j+k;
        r.bal = 0;
    }
}
class Account {
    private final Guard g;
    private int bal guarded_by g = 0;

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

    public void transferAll(Account r) requires g, r.g {
        int j = bal;
        int k = r.bal;
        bal = j+k;
        r.bal = 0;
    }
}
class Account<Ghost l> {  
   private int bal guarded_by l = 0;

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

   public void transferAll(Account<l> r) requires l {  
      int j = bal;  
      int k = r.bal;  
      bal = j+k;  
      r.bal = 0;  
   }
}
Type Checking

Lock Set must be included as part of the environment.

\[ P; E; ls \vdash e : t \]

Program  Environment  Lock set
Type Checking

class Account {
    private int bal  guarded_by this = 0;

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

    public void transferAll(Account r) requires this, r {
        int j = bal;
        int k = r.bal;
        bal = j+k;
        r.bal = 0;
        Account a = getAccnt(10220);
        Account b = getAccnt(22123);
        synchronized(a, b){
            a.transferAll(b);
        }
    }
}
class Account {
    private final Guard g guarded_by this;
    private int bal guarded_by g = 0;

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

    public void transferAll(Account r) requires g, r.g {
        int j = bal;
        int k = r.bal;
        bal = j + k;
        r.bal = 0;
    }
}

Account a = getAccnt(10220);
Account b = getAccnt(22123);
synchronized(a.g, b.g) {
    a.transferAll(b);
}
Typing Rules

\[
\begin{align*}
\text{[EXP FORK]} \\
P; E; \emptyset & \vdash e : t \\
& \hline \\
& P; E; ls \vdash \text{fork } e : \text{int}
\end{align*}
\]
Typing Rules

\[
\begin{align*}
\text{[EXP SYNC]} & \quad \frac{P; E \vdash_{\text{final}} e_1 : c}{P; E; ls \cup \{e_1\} \vdash e_2 : t} \\
& \quad \frac{P; E; ls \vdash \text{synchronized } e_1 \text{ in } e_2 : t}{P; E; ls \vdash \text{synchronized } e_1 \text{ in } e_2 : t}
\end{align*}
\]
Typing Rules

\[\text{[EXP REF]}\]

\[
P; E; ls \vdash e : c
\]

\[
P; E \vdash ([\text{final}]_{\text{opt}} t \text{ fd guarded by } l = e') \in c
\]

\[
P; E \vdash [e/\text{this}]l \in ls
\]

\[
P; E \vdash [e/\text{this}]t
\]

\[
P; E; ls \vdash e.f\text{d} : [e/\text{this}]t
\]

\[\text{[EXP ASSIGN]}\]

\[
P; E; ls \vdash e : c
\]

\[
P; E \vdash (t \text{ fd guarded by } l = e'') \in c
\]

\[
P; E \vdash [e/\text{this}]l \in ls
\]

\[
P; E; ls \vdash e' : [e/\text{this}]t
\]

\[
P; E; ls \vdash e.f\text{d} = e' : [e/\text{this}]t
\]
Example

class Node<ghost l> {
    Node<l> next guarded_by l;
    int v guarded_by l;
}

class List {
    Node<this> head;

    void add(int x) requires this {
        Node<this> t = new Node<this>(x);
        t.next = head;
        head = t;
    }
}

{ List l = getList();
  synchronized(l) { l.add(5); }
}
Type Inference

How do we avoid adding all of these annotations?
Reducing Type Inference to SAT

class Ref {
    int i;
    void add(Ref r)
    {
        i = i
            + r.i;
    }
}
Reducing Type Inference to SAT

- Add ghost parameters `<ghost g>` to each class declaration.
Reducing Type Inference to SAT

```java
class Ref<ghost g> {
    int i guarded_by X1;
    void add(Ref r)
    {
        i = i + r.i;
    }
}
```

- Add ghost parameters `<ghost g>` to each class declaration
- Add `guarded_by Xi` to each field declaration
  - type inference resolves `Xi` to some lock
Reducing Type Inference to SAT

```java
class Ref<ghost g> {
    int i guarded_by X1;
    void add(Ref<X2> r) {
        i = i + r.i;
    }
}
```

- Add ghost parameters `<ghost g>` to each class declaration
- Add `guarded_by Xi` to each field declaration
  - type inference resolves `Xi` to some lock
- Add `<Xi>` to each class reference
Reducing Type Inference to SAT

```java
class Ref<ghost g> {
    int i guarded_by X1;
    void add(Ref<X2> r)
        requires Yi
    {
        i = i
        + r.i;
    }
}
```

- Add ghost parameters `<ghost g>` to each class declaration
- Add `guarded_by Xi` to each field declaration
  - type inference resolves `Xi` to some lock
- Add `<Xi>` to each class reference
- Add `requires Yi` to each method
  - type inference resolves `Yi` to some set of locks
Overview of Type Inference

Add Unknowns:

```java
class Ref<ghost g> {
    int i guarded_by X1;
    ...
}
```

Unannotated Program:

```java
class Ref {
    int i;
    ...
}
```

Annotated Program:

```java
class Ref<ghost g> {
    int i guarded_by g;
    ...
}
```

Constraints:

```java
X1 ∈ { this, g }
...
```

SAT problem:

`(b1 \rightarrow b5)`

SAT solver

Error:

"potential race on field i"

Constraint Solution:

```java
X1 = g
...
```

SAT soln:

`b1=false`

unsatisfiable

satisfiable
Sketch of a Type Soundness Proof

- Machine states: $s = (h, \{t_1, ..., t_N\})$
  - Global heap $h$ and current threads $t_i$, each containing an expression and a **Lock Set**.
- $s \rightarrow s'$ picks a thread to run, but **gets stuck** if a **race condition** is enabled in $s$. 