A Unified, Machine-Checked Formalisation of Java and the Java Memory Model

Andreas Lochbihler

funded by DFG grants Sn11/10-1,2

PROGRAMMING PARADIGMS GROUP

```
theorem drf:
  assumes sync: "correctly_synchronized P E"
  and legal: "legal_execution P E (E, ws)"
  shows "sequentially_consistent P (E, ws)"
using legal_wf_execD[OF legal] legal_ED[OF legal] sync
proof(rule drf_lemma)
  fix r
  assume "r ∈ read_actions E"

  from legal obtain J where E: "E ∈ E"
    and wf_exec: "P ⊩ (E, ws) √"
    and J: "P ⊩ (E, ws) justified_by J"
```
Why do we need a memory model?

initially: \( x = y = 0; \)

\[
\begin{align*}
  x &= 1; & y &= 2; \\
  j &= y; & i &= x;
\end{align*}
\]
Why do we need a memory model?

Initially: $x = y = 0$

- $x = 1$
- $j = y$
- $i = x$

<table>
<thead>
<tr>
<th></th>
<th>$i = 0$</th>
<th>$i = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j = 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$j = 2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interleaving semantics
Why do we need a memory model?

initially: \( x = y = 0 \);  
\[
\begin{align*}
x &= 1;  
y &= 2;  
j &= y;  
i &= x;
\end{align*}
\]

interleaving semantics

\[
\begin{array}{c|c|c}
& j = 0 & j = 2 \\
i &= 0 & \checkmark \\
i &= 1 & \checkmark
\end{array}
\]
Why do we need a memory model?

initially: \( x = y = 0 \);
\[
\begin{align*}
x &= 1; \\
j &= y; \\
y &= 2; \\
i &= x;
\end{align*}
\]

interleaving semantics

\[
\begin{array}{c|cc}
j &= 0 & j &= 2 \\
i &= 0 & \checkmark & \checkmark \\
i &= 1 & \checkmark & \\
\end{array}
\]
Why do we need a memory model?

Initially: $x = y = 0$;

$x = 1$; $y = 2$;
$j = y$; $i = x$;

Interleaving semantics

<table>
<thead>
<tr>
<th>$j$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>✔</td>
</tr>
</tbody>
</table>

Compiler and hardware reorder statements.
Why do we need a memory model?

Initially: $x = y = 0$

<table>
<thead>
<tr>
<th>x = 1;</th>
<th>y = 2;</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = y;</td>
<td>i = x;</td>
</tr>
</tbody>
</table>

Interleaving semantics

<table>
<thead>
<tr>
<th>j == 0</th>
<th>j == 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i == 0</td>
<td>X</td>
</tr>
<tr>
<td>i == 1</td>
<td>✓</td>
</tr>
</tbody>
</table>

Compiler and hardware reorder statements

$x = 1; j = y; y = 2; i = x;$
Why do we need a memory model?

initially: $x = y = 0$

Compiler and hardware reorder statements

interleaving semantics

<table>
<thead>
<tr>
<th></th>
<th>$j = 0$</th>
<th>$j = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0$</td>
<td>X</td>
<td>√</td>
</tr>
<tr>
<td>$i = 1$</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>
Why do we need a memory model?

initially: \( x = y = 0 \);

- \( x = 1 \); \( y = 2 \);
- \( j = y \); \( i = x \);

interleaving semantics

\[
\begin{array}{c|cc}
& j == 0 & j == 2 \\
i == 0 & X & \checkmark \\
i == 1 & \checkmark & \checkmark \\
\end{array}
\]

compiler and hardware reorder statements

\[
\begin{array}{c|cc}
& j == 0 & j == 2 \\
i == 0 & \checkmark & \checkmark \\
i == 1 & \checkmark & X \\
\end{array}
\]

\[
\begin{array}{c|cc}
& j == 0 & j == 2 \\
i == 0 & \checkmark & \checkmark \\
i == 1 & \checkmark & X \\
\end{array}
\]
Why do we need a memory model?

Java memory model

<table>
<thead>
<tr>
<th>Initially: $x = y = 0$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1$;</td>
</tr>
<tr>
<td>$j = y$;</td>
</tr>
<tr>
<td>$y = 2$;</td>
</tr>
<tr>
<td>$i = x$;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$j = 0$</th>
<th>$j = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0$</td>
<td>✓</td>
</tr>
<tr>
<td>$i = 1$</td>
<td>✓</td>
</tr>
</tbody>
</table>

compiler and hardware reorder statements

<table>
<thead>
<tr>
<th>$j = 0$</th>
<th>$j = 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0$</td>
<td>✓</td>
</tr>
<tr>
<td>$i = 1$</td>
<td>✓</td>
</tr>
</tbody>
</table>
The Java memory model: goals

1. allow compiler optimisations

2. interleaving semantics for data-race-free programs (DRF guarantee)

3. give semantics to all Java programs

4. support type safety and security architecture
The Java memory model: goals

1. allow compiler optimisations
   too restricted
   [Cenciarelli et al. 07; Ševčík, Aspinall 08; Torlak et al. 10]

2. interleaving semantics for data-race-free programs (DRF guarantee)

3. give semantics to all Java programs

4. support type safety and security architecture
The Java memory model: goals

1. allow compiler optimisations too restricted
   [Cenciarelli et al. 07; Ševčík, Aspinall 08; Torlak et al. 10]

2. interleaving semantics for data-race-free programs (DRF guarantee)
   proofs with holes
   [Manson et al. 05; Aspinall, Ševčík 07; Huisman, Petri 07]

3. give semantics to all Java programs

4. support type safety and security architecture
The Java memory model: goals

1. allow compiler optimisations
   too restricted
   [Cenciarelli et al. 07; Ševčík, Aspinall 08; Torlak et al. 10]

2. interleaving semantics for data-race-free programs (DRF guarantee)
   proofs with holes
   [Manson et al. 05; Aspinall, Ševčík 07; Huisman, Petri 07]

3. give semantics to all Java programs
   informal, loose connection with Java
   main cause for technical complexity

4. support type safety and security architecture
The Java memory model: goals

1. allow compiler optimisations
too restricted
[Cenciarelli et al. 07; Ševčík, Aspinall 08; Torlak et al. 10]

2. interleaving semantics for data-race-free programs (DRF guarantee)
proofs with holes
[Manson et al. 05; Aspinall, Ševčík 07; Huisman, Petri 07]

3. give semantics to all Java programs
informal, loose connection with Java
main cause for technical complexity

4. support type safety and security architecture
open
The Java memory model: goals

1. allow compiler optimisations too restricted [Cenciarelli et al. 07; Ševčík, Aspinall 08; Torlak et al. 10]

2. interleaving semantics for data-race-free programs (DRF guarantee) proofs with holes formally proven for Java-like language [Manson et al. 05; Aspinall, Ševčík 07; Huisman, Petri 07]

3. give semantics to all Java programs informal, loose connection with Java-like language formalised main cause for technical complexity

4. support type safety and security architecture open
Quis custodiet ipsos custodes?

Joana: [Hammer, Snelting 09] information flow control for multithreaded Java

Quis custodiet: verify IFC algorithm

- analyses assume interleaving semantics
  ⇒ DRF guarantee makes them applicable to DRF programs
**JinjaThreads**

**sequential features**
- classes, objects, fields, arrays
- inheritance and late binding
- exceptions
- imperative features

**concurrency**
- thread creation
- synchronisation
- wait-notify
- join, interruption

### source code

<table>
<thead>
<tr>
<th>Java memory model</th>
</tr>
</thead>
<tbody>
<tr>
<td>complete interleavings</td>
</tr>
<tr>
<td>interleaved small-step</td>
</tr>
</tbody>
</table>

### bytecode

<table>
<thead>
<tr>
<th>thread start &amp; finish actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>small-step semantics</td>
</tr>
<tr>
<td>native methods</td>
</tr>
</tbody>
</table>

- multithreaded actions
- single thread

[Klein, Nipkow 06]
Connecting JinjaThreads with the JMM

Initially: \( y = 0 \);

```java
T t2 = new T();
t2.start();
t2.join();
```

```java
class T extends Thread {
    public void run() {
        print(y);
    }
}
```
Connecting JinjaThreads with the JMM

initially: \( y = 0; \)

\[
\begin{align*}
T \ t2 &= \text{new} \ T(); \\
t2.\text{start}(); \\
t2.\text{join}();
\end{align*}
\]

\[
\text{class T extends Thread \{} \\
\quad \text{public void run() \{} \\
\quad \quad \text{print}(y); \} \}
\]

A. interleave threads and record actions

→ 1. bootstrap
2. allocation
3. execute constructor
4. spawn
5. start
6. read \( y \)
7. print \( y \)
8. finish
9. join
10. finish

\[
\begin{align*}
\ldots, \ t1: &\text{[Init } y \ 0], \ldots \\
\ t1: &\text{[Init t2’s fields]} \\
\ t1: &[], \ldots \\
\ t1: &[], \ t1: &\text{[Spawn t2 ], t1:[]} \\
\ t2: &\text{[Start]} \\
\ t2: &\text{[Read y } v]\] \\
\ t2: &\text{[External print } v]\] \\
\ t2: &\text{[Finish]} \\
\ t1: &\text{[NotInterrupted t1, Join t2]} \\
\ t1: &\text{[Finish]} 
\end{align*}
\]
initially: \( y = 0 \);

\[
\begin{array}{ll}
T \ t2 = \text{new} \ T(); & \text{class} \ T \ \text{extends} \ \text{Thread} \ \{ \\
\text{t2.start();} & \quad \quad \text{public} \ \text{void} \ \text{run}(); \ \{ \\
\text{t2.join();} & \quad \quad \quad \text{print}(y); \ \} \ \} \\
\end{array}
\]

A. interleave threads and record actions

1. bootstrap
→ 2. allocation
3. execute constructor
4. spawn
5. start
6. read \( y \)
7. print \( y \)
8. finish
9. join
10. finish

...; \ t1: [Init y 0], ...
\( t1: [\text{Init} \ t2\text{’s fields}] \)
\( t1: [], ... \)
\( t1: [], t1: [\text{Spawn} \ t2], t1: [] \)
\( t2: [\text{Start}] \)
\( t2: [\text{Read} \ y \ v] \)
\( t2: [\text{External print} \ v] \)
\( t2: [\text{Finish}] \)
\( t1: [\text{NotInterrupted} \ t1, \text{Join} \ t2] \)
\( t1: [\text{Finish}] \)
Connecting JinjaThreads with the JMM

initially: \( y = 0 \);

\[
\begin{align*}
T &\, t2 = \text{new} \, T() ; \\
t2 &\text{.start();} \\
t2 &\text{.join();}
\end{align*}
\]

\[
\begin{align*}
\text{class T extends Thread} &\{ \\
&\quad \text{public void run()} \{ \\
&\quad \quad \text{print}(y); \} \}
\end{align*}
\]

A. interleave threads and record actions

1. bootstrap \( t1 \):
   \( \text{Init} \, y \, 0 \), ...
2. allocation \( t1 \):
   \( \text{Init} \, t2's \, fields \)
3. execute constructor
   \( t1[:] \), ...
4. spawn \( t1[:] \), \( t1: [\text{Spawn} \, t2] \), \( t1[:] \)
5. start \( t2: [\text{Start}] \)
6. read \( y \)
   \( t2: [\text{Read} \, y \, v] \)
7. print \( y \)
   \( t2: [\text{External print} \, v] \)
8. finish \( t2: [\text{Finish}] \)
9. join \( t1: [\text{NotInterrupted} \, t1, \, \text{Join} \, t2] \)
10. finish \( t1: [\text{Finish}] \)

B. flatten & purge irrelevant actions

C. reconstruct orders \( \leq \) \( hb \), \( \leq \) \( so \) match reads and writes \( v = 0 \)

D. impose JMM legality constraints
Connecting JinjaThreads with the JMM

initially: \( y = 0 \);

\[
\begin{align*}
T & \ t2 = \text{new } T() \\
& \ t2.\text{start}() \\
& \ t2.\text{join}();
\end{align*}
\]

class T extends Thread {
    public void run() {
        print(y); }
    }

A. interleave threads and record actions

1. bootstrap
2. allocation
3. execute constructor
   → 4. spawn
5. start
6. read \( y \)
7. print \( y \)
8. finish
9. join
10. finish

\[\ldots, \ t1:\text{[Init } y \ 0], \ \ldots\]
\[\ t1:\text{[Init } t2\text{'s fields]}\]
\[\ t1:[]\text{, } \ldots\]
\[\ t1:[]\text{, } t1:\text{[Spawn } t2 \ ], \ t1:[]\]
\[\ t2:\text{[Start]}\]
\[\ t2:\text{[Read } y \ v]\]
\[\ t2:\text{[External print } v]\]
\[\ t2:\text{[Finish]}\]
\[\ t1:\text{[NotInterrupted } t1, \ Join \ t2]\]
\[\ t1:\text{[Finish]}\]
Connecting JinjaThreads with the JMM

initially: \( y = 0 \);

| T t2 = new T(); | class T extends Thread { |
| t2.start(); | public void run() { |
| t2.join(); | print(y); } } |

A. interleave threads and record actions

1. bootstrap
2. allocation
3. execute constructor
4. spawn
→ 5. start
6. read \( y \)
7. print \( y \)
8. finish
9. join
10. finish

... , t1:[Init \( y \) 0], ...
... , t1:[Init t2’s fields]
t1:[ ] , ...
t1:[ ] , t1:[Spawn t2 ], t1:[ ]
t2:[Start]
t2:[Read \( y \) \( v \)]
t2:[External print \( v \)]
t2:[Finish]
t1:[NotInterrupted t1, Join t2]
t1:[Finish]
Connecting JinjaThreads with the JMM

initially: \( y = 0 \);

\[
\begin{align*}
T & \ t2 = \text{new } T(); \\
& \ t2.\text{start}(); \\
& \ t2.\text{join}(); \\
\end{align*}
\]

class T extends Thread {
    public void run()
    {
        print(y); 
    }
}

A. interleave threads and record actions

1. bootstrap
2. allocation
3. execute constructor
4. spawn
5. start
→ 6. read \( y \)
7. print \( y \)
8. finish
9. join
10. finish

\[
\begin{align*}
\ldots, \ t1: [\text{Init } y \ 0], \ldots \\
\ t1: [\text{Init t2’s fields}] \\
\ t1: [\text{]}, \ldots \\
\ t1: [\text{]}, \ t1: [\text{Spawn t2 }], \ t1: [\text{]}] \\
\ t2: [\text{Start}] \\
\ t2: [\text{Read } y \ v] \\
\ t2: [\text{External print } v] \\
\ t2: [\text{Finish}] \\
\ t1: [\text{NotInterrupted t1, Join t2}] \\
\ t1: [\text{Finish}] \\
\end{align*}
\]
Connecting JinjaThreads with the JMM

initially: \( y = 0 \);

\[
\begin{align*}
T & \ t2 = \text{new} \ T() \\
t2. & \text{start}(); \\
t2. & \text{join}();
\end{align*}
\]

class \( T \) extends \( \text{Thread} \) {
  \[
  \text{public\ void\ run}()\ \{ \\
  \quad \text{print}(y);\ \}
  \}
\]

A. interleave threads and record actions

1. bootstrap
2. allocation
3. execute constructor
4. spawn
5. start
6. read \( y \)

\( \rightarrow \) 7. print \( y \)
8. finish
9. join
10. finish

non-deterministic value \( \nu \):

B. flatten & purge irrelevant actions
C. reconstruct orders \( \leq_{hb}, \leq_{so} \) match reads and writes
D. impose JMM legality constraints
Connecting JinjaThreads with the JMM

initially: \(y = 0\);

\[
\begin{align*}
T \ t2 & = \text{new} \ T(); \\
t2.\text{start}(); \\
t2.\text{join}();
\end{align*}
\]

class \(T\) extends Thread {
public void \(\text{run}\)() {
    \text{print}(y); 
} 
}

A. interleave threads and record actions

1. bootstrap \(t1\)
2. allocation \(t1\)
3. execute constructor \(t1\)
4. spawn \(t1\)
5. start \(t2\)
6. read \(y\) \(t2\)
7. print \(y\) \(t2\)
8. finish \(t2\)
9. join \(t1\)
10. finish \(t1\)

B. flatten & purge irrelevant actions

C. reconstruct orders \(\leq_h b, \leq_s o\) match reads and writes \(v = 0\)

D. impose JMM legality constraints

non-deterministic value \(v\)
Connecting JinjaThreads with the JMM

initially: \( y = 0; \)

\[
\begin{align*}
T\ t2 &= \text{new}\ T(); \\
t2.\text{start}(); \\
t2.\text{join}();
\end{align*}
\]

\[
\begin{align*}
\text{class}\ T\ \text{extends}\ \text{Thread} \{ \\
\quad \text{public}\ \text{void}\ \text{run}() \{ \\
\quad\quad \text{print}(y); \} \}
\end{align*}
\]

A. interleave threads and record actions

1. bootstrap
2. allocation
3. execute constructor
4. spawn
5. start
6. read \( y \)
7. print \( y \)
8. finish

→ 9. join
10. finish

non-deterministic value \( v \)
Connecting JinjaThreads with the JMM

initially: \( y = 0; \)

\[
\begin{align*}
T & \ t2 = \text{new } T(); \\
& t2.\text{start}(); \\
& t2.\text{join}(); \\
\text{class } T & \text{ extends Thread} \\
\& \ \text{public void } \text{run}() \{ \\
\& \ \quad \text{print}(y); \} \} \\
\end{align*}
\]

A. interleave threads and record actions

1. bootstrap
2. allocation
3. execute constructor
4. spawn
5. start
6. read \( y \)
7. print \( y \)
8. finish
9. join
10. finish

\[\text{non-deterministic value } v\]
Connecting JinjaThreads with the JMM

initially: \( y = 0 \);

\[
\begin{array}{ll}
T \ t2 = \text{new} \ T(); \\
t2.\text{start}(); \\
t2.\text{join}();
\end{array}
\quad
class \ T \text{ extends Thread} \{
\begin{array}{l}
\text{public void run}() \{
\quad \text{print}(y); \}
\end{array}
\} \]

A. interleave threads and record actions

B. flatten & purge irrelevant actions

\begin{align*}
\ldots, \ t1: \text{Init } y \ 0, \ldots \\
\ t1: \text{Init } t2\text{'s fields} \\
\ \\
\ldots \\
\ t1: \text{Spawn } t2 \\
\ t2: \text{Start} \\
\ t2: \text{Read } y \ v \\
\ t2: \text{External print } v \\
\ t2: \text{Finish} \\
\ t1: \text{Join } t2 \\
\ t1: \text{Finish}
\end{align*}
Connecting JinjaThreads with the JMM

initially: \( y = 0 \);

\[
\begin{align*}
T \ t2 &= \text{new T();} \\
t2.\text{start();} \\
t2.\text{join();}
\end{align*}
\]

\[
\begin{align*}
\text{class T extends Thread {} } \\
\quad \text{public void run() {} } \\
\quad \quad \text{print(y);} \\
\text{}}
\]

A. interleave threads and record actions

B. flatten & purge irrelevant actions

C. reconstruct orders \( \leq_{hb}, \leq_{so} \) match reads and writes

\[
\begin{align*}
\ldots, t1: \text{Init } y \ 0, \ldots \\
t1: \text{Init } t2's \text{ fields} \\
\ldots \\
t1: \text{Spawn } t2 \\
t2: \text{Start} \\
t2: \text{Read } y \ v \\
t2: \text{External print } v \\
t2: \text{Finish} \\
t1: \text{Join } t2 \\
t1: \text{Finish}
\end{align*}
\]
Connecting JinjaThreads with the JMM

initially: \( y = 0; \)

| T t2 = new T(); | class T extends Thread { |
| t2.start(); | public void run() { |
| t2.join(); | print(y); } } |

A. interleave threads and record actions

B. flatten \& purge irrelevant actions

\( \ldots, t1: \text{Init } y \ 0, \ldots \)
\( t1: \text{Init } t2\text{'s fields} \)
\( \ldots \)
\( t1: \text{Spawn } t2 \)
\( t2: \text{Start} \)
\( t2: \text{Read } y \ v \)
\( t2: \text{External print } v \)
\( t2: \text{Finish} \)
\( t1: \text{Join } t2 \)
\( t1: \text{Finish} \)

C. reconstruct orders \( \leq_{hb}, \leq_{so} \) match reads and writes

D. impose JMM legality constraints

\( v = 0 \)
**DRF guarantee**

**sequential consistency (SC)** every read sees most recent write

**data race** two conflicting actions unrelated in $\leq_{hb}$ read/write, write/read, write/write to non-volatile location

**data race free (DRF)** no data race in any SC execution of the program

**DRF guarantee** DRF programs *behave* like under interleaving semantics.

**Theorem**

*No data race in SC executions* $\implies$ *all executions are SC.*

**implications for Java programmers:**

- Always synchronise and forget about the JMM.
- Mark all synchronisation variables (*volatile*, *synchronized*).
- Use only allowed synchronisation primitives.
Implicit communication channels

1. run-time type information as global state

initially: \( x = \text{false}; \ y = \text{null}; \)

\[
\begin{array}{l}
\text{x = true;} \\
\text{r1 = x;} \\
\text{y = (r1 ? new A() : new B());} \\
\text{r2 = y.f();}
\end{array}
\]
Implicit communication channels

1. run-time type information as global state

<table>
<thead>
<tr>
<th>initially: ( x = \text{false}; y = \text{null}; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{true}; )</td>
</tr>
<tr>
<td>( y = (r_1 ? \text{new A() : new B()}); )</td>
</tr>
</tbody>
</table>
Implicit communication channels

1. run-time type information as global state

<table>
<thead>
<tr>
<th>x = true;</th>
<th>r1 = x;</th>
<th>y = (r1 ? new A() : new B());</th>
<th>r2 = y.f();</th>
</tr>
</thead>
</table>

initially: \( x = \text{false}; \) \( y = \text{null}; \)

disallowed synchronisation

dispatch to \( A.f() \)
\[ \Rightarrow r1 == \text{true} \]
Implicit communication channels

1. Run-time type information as global state

Initially: \( x = \text{false}; y = \text{null}; \)

\[
\begin{align*}
\text{x} &= \text{true}; & \text{r1} &= x; \\
\text{y} &= \text{(r1 ? new A() : new B());} & \text{r2} &= \text{y.f();} \\
\end{align*}
\]

2. Synchronisation via `Thread.start`

Initially: \( x = \text{new Thread(); y = 0;} \)

\[
\begin{align*}
\text{y} &= \text{1;} & \text{try} \{ \text{x.start();} \\
\text{x.start();} & \text{catch (IllegalThreadStateException \_)} \{ \text{r = y; } \}
\end{align*}
\]
Implicit communication channels

1. Run-time type information as global state

   Initially: \( x = \text{false}; y = \text{null}; \)

   \[
   \begin{array}{c|c|c}
   x = \text{true}; & r1 = x; & r2 = y.f() \\
   y = (r1 ? \text{new A()} : \text{new B()}); & & \\
   \end{array}
   \]

   Disallowed synchronisation

2. Synchronisation via \texttt{Thread.start}

   Initially: \( x = \text{new Thread()}; y = 0; \)

   \[
   \begin{array}{c|c}
   y = 1; & \text{try} \{ x.\text{start()}; \\
   x.\text{start()}; & \} \text{catch (IllegalThreadStateException _)} \{ r = y; \} \\
   \end{array}
   \]

   Data race?
Implicit communication channels

1. run-time type information as global state
   - Initially: \( x = \text{false}; y = \text{null}; \)
   - \( x = \text{true}; \)
   - \( r1 = x; \)
   - \( y = (r1 ? \text{new A()} : \text{new B()}); \)
   - \( r2 = y.f(); \)

   "disallowed synchronisation"

2. synchronisation via `Thread.start`
   - Initially: \( x = \text{new Thread(); } y = 0; \)
   - \( y = 1; \)
   - \( x.start(); \)
   - \( \text{try} \{ x.start(); \}
   - \( \text{try} \{ x.start(); \}
   - \( \text{catch} (\text{IllegalThreadStateException } _) \{ r = y; \} \)

   "data race?"

   Intuition: no  JMM: yes
Implicit communication channels

1. run-time type information as global state

   initially: \( x = \text{false}; \ y = \text{null}; \)

\[
\begin{align*}
x &= \text{true}; & r1 &= x; & r2 &= y.\text{f}(); \\
y &= (r1 \ ? \ \text{new A() : new B()}); & \end{align*}
\]

2. synchronisation via \text{Thread.start}

   initially: \( x = \text{new Thread(); \ y = 0; } \)

\[
\begin{align*}
y &= 1; & \text{try} \{ \ x.\text{start();} \\
x.\text{start();} \} \ \text{catch (IllegalThreadStateException \_)} \{ \ r = y; \}
\end{align*}
\]
Theorem (DRF guarantee)

No data race in SC executions \[\implies\] all executions are SC.

Assumptions on complete interleavings:
1. SC completions for SC prefix
2. unique initialisations before read in SC prefix

Insights:
proofs abstract from form of allowed synchronisation
allocations (initialisations) complicate proofs
special treatment irrelevant for DRF programs
Theorem (DRF guarantee)

*No data race in SC executions* → *all executions are SC.*

**Assumptions on complete interleavings:**
1. SC completions for SC prefix
2. unique initialisations before read in SC prefix

Java memory model

- complete interleavings
- interleaved small-step
- single-thread semantics

**theorem drf:**

- assumes sync: "correctly_synchronized P E" and legal: "legal_execution P E (E, ws)"
- shows "sequentially_consistent P (E, ws)"
Theorem (DRF guarantee)

No data race in SC executions \(\implies\) all executions are SC.

Assumptions on complete interleavings:
1. SC completions for SC prefix
2. unique initialisations before read in SC prefix
DRF guarantee

Theorem (DRF guarantee)
No data race in SC executions \(\implies\) all executions are SC.

Assumptions on complete interleavings:
1. SC completions for SC prefix
2. unique initialisations before read in SC prefix

Java memory model
complete interleavings
interleaved small-step
single-thread semantics

axiomatic constraints

coinductive characterisation of SC prefixes

operational semantics

theorem drf:
assumes sync: "correctly_synchronized P E"
and legal: "legal_execution P E (E, ws)"
shows "sequentially_consistent P (E, ws)"
Theorem (DRF guarantee)

No data race in SC executions \(\implies\) all executions are SC.

Assumptions on complete interleavings:
1. SC completions for SC prefix
2. unique initialisations before read in SC prefix

Java memory model

- complete interleavings
- interleaved small-step
- single-thread semantics

axiomatic constraints

operational semantics

construct SC completion corecursively, assume "cut and update"

coinductive characterisation of SC prefixes

```
them drf:
  assumes sync: "correctly_synchronized P E" and legal: "legal_execution P E (E, ws)"
  shows "sequentially_consistent P (E, ws)"
```
**Theorem (DRF guarantee)**

No data race in SC executions \[\Rightarrow\] all executions are SC.

**Assumptions on complete interleavings:**
1. SC completions for SC prefix

**Insights:**
- proofs abstract from form of *allowed* synchronisation
- allocations (initialisations) complicate proofs
- special treatment irrelevant for DRF programs

**Construct SC completion corecursively, assume “cut and update”**
Conclusion

Results:

1. rigorous link between Java and JMM
   complete set of Java multithreading

2. DRF guarantee holds definitely
   ⇒ DRF guarantee formally available, e.g., for program analyses

3. all definitions and proofs machine-checked

Outlook: JMM too weak for programs with races  [forthcoming PhD thesis]

type safety  weak version holds
  but unallocated memory can be accessed

security architecture  compromised, values can appear out of thin air