Making the Java Memory Model Safe*

Andreas Lochbihler

Institute for Information Security
ETH Zurich

*supported by DFG Sn11/10-1,2
The need for a formal model of Java

Safety guarantees of Java

- definedness
- type safety
- security architecture (sandbox)
The need for a formal model of Java

Safety guarantees of Java
- definedness
- type safety
- security architecture (sandbox)

KeY-System Krakatoa / Why3
Java Path Finder Joana

Andreas Lochbihler (ETH Zürich) Making the Java Memory Model Safe
The need for a formal model of Java

Concurrency in Java
- threads
- synchronisation primitives
- memory model

Safety guarantees of Java
- definedness
- type safety
- security architecture (sandbox)

KeY-System Krakatoa / Why3
Java Path Finder
Joana

Andreas Lochbihler (ETH Zürich)
Making the Java Memory Model Safe
The need for a formal model of Java

Concurrency in Java
- threads
- synchronisation primitives
- memory model

Safety guarantees of Java
- definedness
- type safety
- security architecture (sandbox)

Implications?

KeY-System
Krakatoa / Why3
Java Path Finder
Joana

Andreas Lochbihler (ETH Zürich)
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>
Why do we need a memory model?

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

interleaving semantics
Why do we need a memory model?

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

\[
x = 1; \\
j = y; \\
i = x;
\]

\[
y = 2; \\
i = x;
\]

interleaving semantics

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

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

\[
\begin{align*}
x &= 1; \\
{j} &= y; \\
i &= x; \\
y &= 2;
\end{align*}
\]

Interleaving semantics:

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

\[\text{initially: } x = y = 0;\]
Why do we need a memory model?

Initially: $x = y = 0$;

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

interleaving semantics

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

Initially: $x = y = 0$;

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

$j = y$;

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>
Why do we need a memory model?

Initially: $x = y = 0$;

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

$j = y$;

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>

Compiler and hardware reorder statements

$j = y$; $i = x$;

$x = 1$; $y = 2$;
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>

Java memory model

<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>
Why do we need a memory model?

Data races

initially: $x = y = 0,$

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

Java memory model

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

Compiler and hardware reorder statements

$j = y;$ $i = x;$ $x = 1;$ $y = 2;$
Semantics in layers

Java memory model

set of well-formed candidate executions

operational semantics

shared memory
Semantics in layers

Java memory model

set of well-formed
candidate executions

operational
semantics

shared
memory

allocation &
type information

Andreas Lochbihler (ETH Zürich)
Making the Java Memory Model Safe
Semantics in layers

Java memory model

set of well-formed
candidate executions

operational
semantics

\[ t : \alpha \]

shared
memory

allocation &
type information

Andreas Lochbihler  (ETH Zürich)
Java memory model

set of well-formed candidate executions

operational semantics

thread communication

shared memory

allocation & type information
Semantics in layers

Java memory model

set of well-formed candidate executions

operational semantics

thread communication

transition system

allocation & type information

shared memory
Java memory model

set of well-formed candidate executions

\[
\{ [t_1 : \alpha_1, t_2 : \alpha_2, \ldots], \\
[t'_1 : \alpha'_1, t'_2 : \alpha'_2, \ldots], \\
[t''_1 : \alpha''_1, t''_2 : \alpha''_2, \ldots], \ldots \}
\]

paths in the transition system

thread communication

operational semantics

shared memory

allocation & type information
Semantics in layers

Java memory model

legality constraints
pair read and write ops

set of well-formed candidate executions

\{ [t_1 : \alpha_1, t_2 : \alpha_2, \ldots], [t'_1 : \alpha'_1, t'_2 : \alpha'_2, \ldots], \ldots \}

paths in the transition system

thread communication

operational semantics

allocation & type information
Semantics in layers

Java memory model

legality constraints
pair read and write ops

need set of candidate executions
cf. [Batty et al.'15]

set of well-formed candidate executions

\[
\begin{align*}
\{ [t_1 : \alpha_1, t_2 : \alpha_2, \ldots], \\
[t'_1 : \alpha'_1, t'_2 : \alpha'_2, \ldots], & \text{ legal} \\
[t''_1 : \alpha''_1, t''_2 : \alpha''_2, \ldots], & \ldots \}
\end{align*}
\]

paths in the transition system

thread communication

operational semantics

\[ t : \alpha \]

shared memory

allocation & type information

Andreas Lochbihler (ETH Zürich)
Dynamic method lookup finds a unique method.

```java
class A { void m() {} }

initially: x = y = null;

r1 = x;
if (r1 != null) r1.m();
y = new A();

r2 = y;
x = r2;
```

Type safety for method calls
Dynamic method lookup finds a unique method.

JMM allows reordering with allocations.

class A { void m() {} }
initially: x = y = null;

r1 = x;
if (r1 != null) r1.m();
y = new A();

reorder

r2 = y;
x = r2;
Dynamic method lookup finds a unique method.

JMM allows reordering with allocations.

```java
class A {
    void m() {}
}

initially: x = y = null;

r1 = x;
if (r1 != null) r1.m();
y = new A();

r2 = y;
x = r2;
```

Separate type information of addresses from their allocation!
Index addresses by dynamic type!

Andreas Lochbihler (ETH Zürich)
Type safety for method calls

Dynamic method lookup finds a unique method.

JMM allows reordering with allocations.

```java
class A {
    void m() {}
}

initially: x = y = null;

r1 = x;
if (r1 != null) r1.m();
y = new A();

r2 = y;
x = r2;
```

Andreas Lochbihler (ETH Zürich)
Dynamic method lookup finds a unique method.

JMM allows reordering with allocations.

Separate type information of addresses from their allocation! Index addresses by dynamic type!
Accessed fields exist and contain only type-conform values.
Type safety for fields

Accessed fields exist and contain only type-conform values.

progress
Accessed fields exist and contain only type-conform values.
Type safety for fields

- **Accessed fields exist and contain only type-conform values.**
- Subject reduction
- Progress

Subject reduction fails, when read op returns value of wrong type.

Andreas Lochbihler (ETH Zürich)
Type safety for fields

Accessible fields exist and contain only type-conform values.

Subject reduction

Subject reduction fails, when read op returns value of wrong type.
Show that reads in legal executions are type-correct.

Java memory model

Operational semantics

Legality constraints

Pair read and write ops

Set of well-formed candidate executions

\{ [t_1: \alpha_1, t_2: \alpha_2, ...],
\quad [t'_1: \alpha'_1, t'_2: \alpha'_2, ...],
\quad [t''_1: \alpha''_1, t''_2: \alpha''_2, ...], ...
\}
Type safety for fields

Accessed fields exist and contain only type-conform values.

Java memory model

legality constraints
pair read and write ops

set of well-formed candidate executions

\{
[t_1 : \alpha_1, t_2 : \alpha_2, \ldots],
[t'_1 : \alpha'_1, t'_2 : \alpha'_2, \ldots],
[t''_1 : \alpha''_1, t''_2 : \alpha''_2, \ldots], \ldots \}

operational semantics

Subject reduction fails, when read op returns value of wrong type.
Type safety for fields

Accessed fields exist and contain only type-conform values.

Java memory model

set of well-formed candidate executions

operational semantics

subject reduction

progress ✓

subject reduction

legality constraints pair read and write ops

Show that reads in legal executions are type-correct.

Subject reduction fails, when read op returns value of wrong type.

Andreas Lochbihler (ETH Zürich)  Making the Java Memory Model Safe 7 / 9
Type safety for fields

Accessed fields exist and contain only type-conform values.

Java memory model

set of well-formed candidate executions

{ [t_1 : \alpha_1, t_2 : \alpha_2, ...],
  [t'_1 : \alpha'_1, t'_2 : \alpha'_2, ...],
  [t''_1 : \alpha''_1, t''_2 : \alpha''_2, ...], ... }
No statement about allocation!

There are legal executions in which some objects are never allocated...

\begin{verbatim}
r1 = x;
r2 = y;
b = true;
if (!b) r1 = new C();
x = r2
y = r1;
\end{verbatim}

initially:
b = false; x = y = null;

allowed:
x,y \neq null, if condition is false.

...because the allocation happened in another execution.

Variations on this program allow you to forge (type-correct) references.
No statement about allocation!

There are legal executions in which some objects are never allocated . . .

<table>
<thead>
<tr>
<th>initially: b = false; x = y = null;</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = x;</td>
</tr>
<tr>
<td>if (!b) r1 = new C();</td>
</tr>
<tr>
<td>y = r1;</td>
</tr>
<tr>
<td>allowed: x,y != null, if condition is false.</td>
</tr>
</tbody>
</table>

. . . because the allocation happened in another execution.
Type safety for allocation

No statement about allocation!

There are legal executions in which some objects are never allocated . . .

<table>
<thead>
<tr>
<th>initially: b = false; x = y = null;</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1 = x;</td>
</tr>
<tr>
<td>if (!b) r1 = new C();</td>
</tr>
<tr>
<td>y = r1;</td>
</tr>
<tr>
<td>r2 = y;</td>
</tr>
<tr>
<td>x = r2</td>
</tr>
<tr>
<td>b = true;</td>
</tr>
</tbody>
</table>

allowed: x, y != null, if condition is false.

. . . because the allocation happened in another execution.

Variations on this program allow you to forge (type-correct) references.
Goals of the Java memory model:

Type safety **holds** despite forging of references
Goals of the Java memory model:

Type safety **holds** despite forging of references

Semantics for all Java program **achieved**.

Main reason for technical complexity
Goals of the Java memory model:

Type safety **holds** despite forging of references

Semantics for *all* Java program **achieved**.

Main reason for technical complexity

Security architecture (sandboxing) **compromised** by forged references
Goals of the Java memory model:

Type safety **holds** despite forging of references

Semantics for **all** Java program **achieved**.

Main reason for technical complexity

Security architecture (sandboxing) **compromised** by forged references

DRF guarantee

Interleaving semantics for programs without data races **proved**.
Goals of the Java memory model:

Type safety holds despite forging of references

Semantics for all Java program achieved.

Main reason for technical complexity

Security architecture (sandboxing) compromised by forged references

DRF guarantee

Interleaving semantics for programs without data races proved.

Compiler optimisations [Ševčík et al.]

JMM fails to allow common optimisations.
Goals of the Java memory model:

Type safety holds despite forging of references

Semantics for all Java program achieved.

Main reason for technical complexity

Security architecture (sandboxing) compromised by forged references

DRF guarantee

Interleaving semantics for programs without data races proved.

Compiler optimisations [Ševčík et al.]

JMM fails to allow common optimisations.

Work on another JMM revision has started (JEP 188).