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

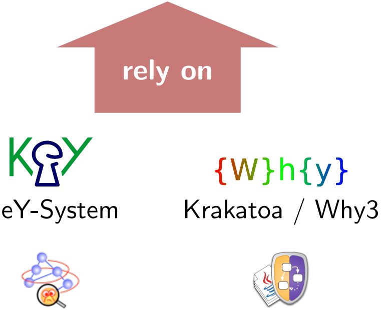
Safety guarantees of Java

- definedness
- type safety
- security architecture (sandbox)

The need for a formal model of Java

Safety guarantees of Java

- definedness
- type safety
- security architecture (sandbox)



rely on

KeY-System

Krakatoa / Why3

Java Path Finder

Joana

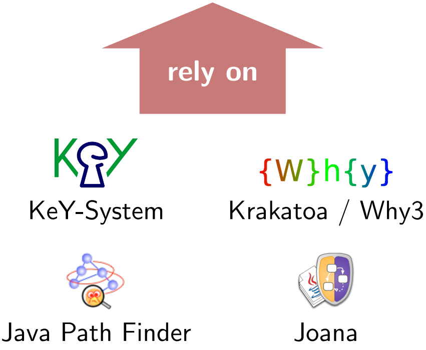
The need for a formal model of Java

Concurrency in Java

- threads
- synchronisation primitives
- memory model

Safety guarantees of Java

- definedness
- type safety
- security architecture (sandbox)



rely on

KeY-System

Krakatoa / Why3

Java Path Finder

Joana

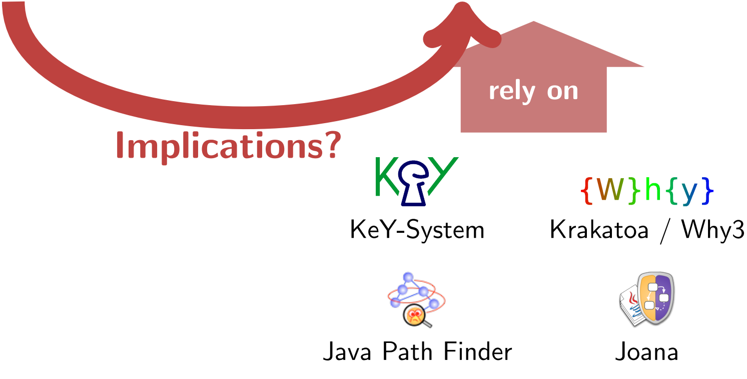
The need for a formal model of Java

Concurrency in Java

- threads
- synchronisation primitives
- **memory model**

Safety guarantees of Java

- definedness
- **type safety**
- security architecture (sandbox)



Implications?

rely on

KeY-System

Krakatoa / Why3

Java Path Finder

Joana

Why do we need a memory model?

initially: `x = y = 0;`

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

Why do we need a memory model?

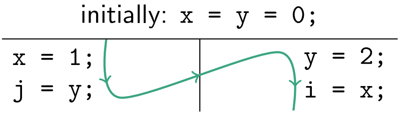
initially: `x = y = 0;`

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

interleaving semantics

	`j == 0`	`j == 2`
`i == 0`		
`i == 1`		

Why do we need a memory model?



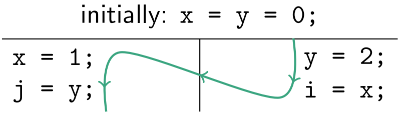
initially: `x = y = 0;`

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

interleaving semantics

	j == 0	j == 2
i == 0		
i == 1	✓	

Why do we need a memory model?



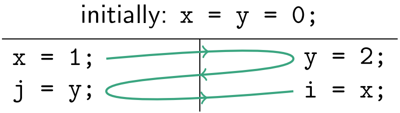
initially: `x = y = 0;`

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

interleaving semantics

	`j == 0`	`j == 2`
`i == 0`		✓
`i == 1`	✓	

Why do we need a memory model?



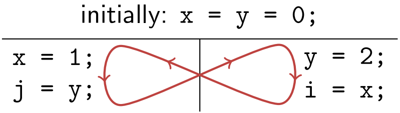
initially: `x = y = 0;`

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

interleaving semantics

	`j == 0`	`j == 2`
`i == 0`		✓
`i == 1`	✓	✓

Why do we need a memory model?



initially: `x = y = 0;`

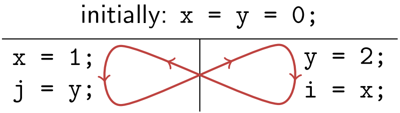
`x = 1;`	`y = 2;`
`j = y;`	`i = x;`

interleaving semantics

	`j == 0`	`j == 2`
`i == 0`	✗	✓
`i == 1`	✓	✓

Why do we need a memory model?

interleaving semantics

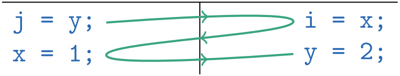


initially: `x = y = 0;`

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

	j == 0	j == 2
i == 0	✗	✓
i == 1	✓	✓

compiler and hardware reorder statements



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

	j == 0	j == 2
i == 0	✓	
i == 1		

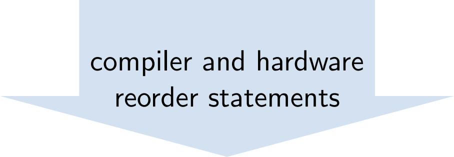
Why do we need a memory model?

Java memory model

initially: x = y = 0;

```
x = 1;          |          y = 2;
j = y;          |          i = x;
```

	j == 0	j == 2
i == 0	✓	✓
i == 1	✓	✓

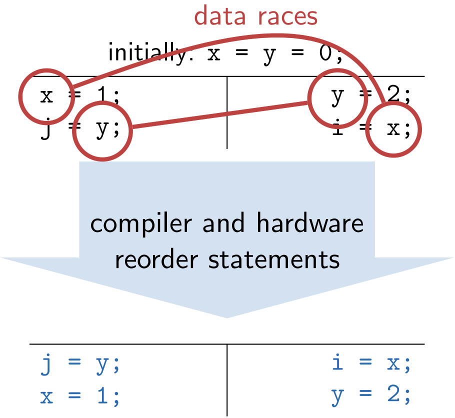


compiler and hardware reorder statements

```
j = y;          |          i = x;
x = 1;          |          y = 2;
```

	j == 0	j == 2
i == 0	✓	
i == 1		

Why do we need a memory model?



data races

```
      initially: x = y = 0;
x = 1;                  y = 2;
j = y;                  i = x;
```

compiler and hardware reorder statements

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

Java memory model

	j == 0	j == 2
i == 0	✓	✓
i == 1	✓	✓

	j == 0	j == 2
i == 0	✓	
i == 1		

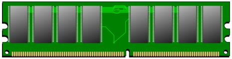
Semantics in layers

Java memory model

set of well-formed candidate executions

operational semantics

shared memory



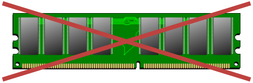
Semantics in layers

Java memory model

set of well-formed
candidate executions

operational
semantics

~~shared
memory~~



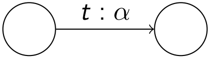
allocation &
type information

Semantics in layers

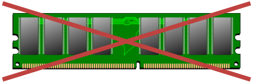
Java memory model

set of well-formed
candidate executions

operational
semantics



~~shared
memory~~



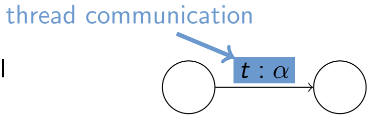
allocation &
type information

Semantics in layers

Java memory model

set of well-formed
candidate executions

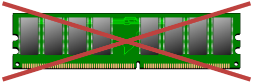
operational
semantics



thread communication

$t : \alpha$

~~shared
memory~~



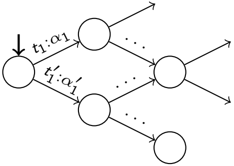
allocation &
type information

Semantics in layers

Java memory model

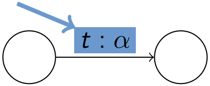
set of well-formed
candidate executions

transition system

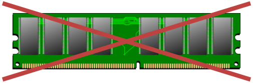


operational
semantics

thread communication



~~shared
memory~~



allocation &
type information

Semantics in layers

Java memory model

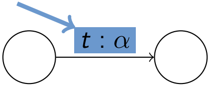
set of well-formed candidate executions

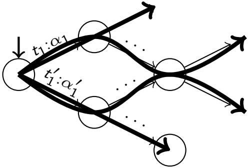
$$\begin{aligned}\{ &[t_1 : \alpha_1, t_2 : \alpha_2, \ldots],\\ &[t_1' : \alpha_1', t_2' : \alpha_2', \ldots],\\ &[t_1'' : \alpha_1'', t_2'' : \alpha_2'', \ldots], \ldots \}\end{aligned}$$

paths in the transition system

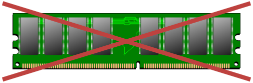
thread communication

operational semantics





~~shared memory~~



allocation & type information

Semantics in layers

Java memory model

legality constraints
pair read and write ops

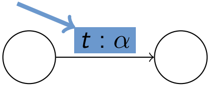
set of well-formed
candidate executions

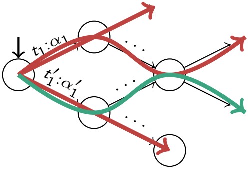
$$\Big\{ \cancel{[t_1 : \alpha_1, t_2 : \alpha_2, \ldots]}, \; [t'_1 : \alpha'_1, t'_2 : \alpha'_2, \ldots], \longleftarrow \text{legal} \; \cancel{[t''_1 : \alpha''_1, t''_2 : \alpha''_2, \ldots]}, \ldots \Big\}$$

paths in the
transition system

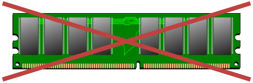
operational
semantics

thread communication





~~shared memory~~



allocation &
type information

Semantics in layers

Java memory model

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

pair read and write ops

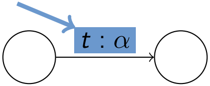
set of well-formed candidate executions

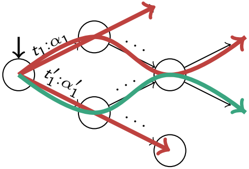
$$\left\{ \cancel{[t_1 : \alpha_1, t_2 : \alpha_2, \ldots]}, \; [t'_1 : \alpha'_1, t'_2 : \alpha'_2, \ldots], \longleftarrow \text{legal} \; \cancel{[t''_1 : \alpha''_1, t''_2 : \alpha''_2, \ldots]}, \ldots \right\}$$

paths in the transition system

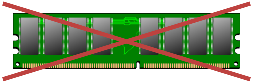
operational semantics

thread communication





~~shared memory~~



allocation & type information

Type safety for method calls

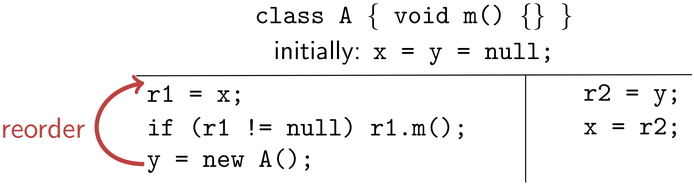
Dynamic method lookup finds a unique method.

```
            class A { void m() {} }
             initially: x = y = null;
-------------------------------------------------
r1 = x;                          |     r2 = y;
if (r1 != null) r1.m();          |     x = r2;
y = new A();                     |
```

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;`	`r2 = y;`
`if (r1 != null) r1.m();`	`x = r2;`
`y = new A();`	

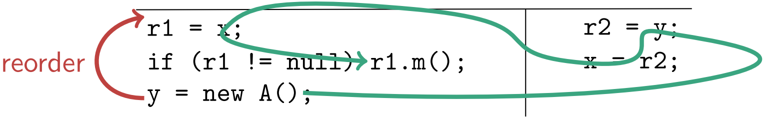
Type safety for method calls

Dynamic method lookup finds a unique method.

JMM allows reordering with allocations.

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

initially: `x = y = null;`

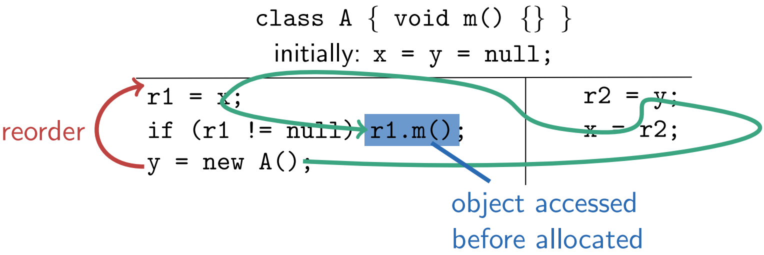


```
r1 = x;                              |  r2 = y;
if (r1 != null) r1.m();              |  x = r2;
y = new A();                         |
```

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;                           r2 = y;
if (r1 != null) r1.m();           x = r2;
y = new A();
```

reorder

object accessed
before allocated

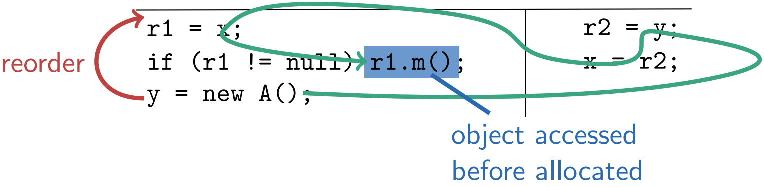
Type safety for method calls

Dynamic method lookup finds a unique method.

JMM allows reordering with allocations.

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

initially: `x = y = null;`



```
r1 = x;                          | r2 = y;
if (r1 != null) r1.m();          | x = r2;
y = new A();                     |
```

reorder

object accessed before allocated

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

Type safety for fields

Accessed fields exist and
contain only type-conform values.

Type safety for fields

progress

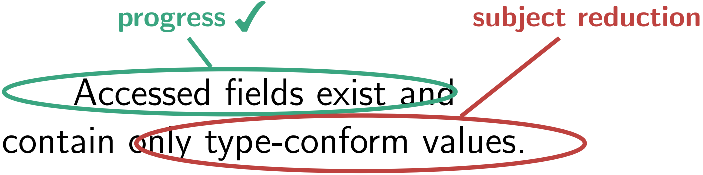
Accessed fields exist and
contain only type-conform values.

Type safety for fields

progress ✓

Accessed fields exist and contain only type-conform values.

Type safety for fields



progress ✓

subject reduction

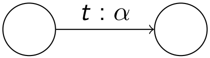
Accessed fields exist and contain only type-conform values.

Type safety for fields

progress ✓

subject reduction

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	

Type safety for fields

progress ✓

subject reduction

Accessed fields exist and contain only type-conform values.

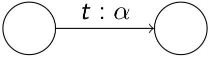
Java memory model

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

progress ✓

subject reduction

Accessed fields exist and contain only type-conform values.

Java memory model

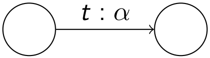
legality constraints
pair read and write ops

Show that reads in legal executions are type-correct.

set of well-formed candidate executions

$$\begin{aligned} \{ & \cancel{[t_1 : \alpha_1, t_2 : \alpha_2, \ldots]}, \\ & [t'_1 : \alpha'_1, t'_2 : \alpha'_2, \ldots], \\ & \cancel{[t''_1 : \alpha''_1, t''_2 : \alpha''_2, \ldots]}, \ldots \} \end{aligned}$$

operational semantics



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

Type safety for fields

progress ✓

subject reduction

Accessed fields exist and contain only type-conform values.

Java memory model

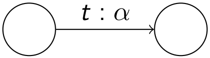
legality constraints
pair read and write ops

Show that reads in legal executions are type-correct.

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 may assume type-correct reads

Type safety for allocation

No statement about allocation!

Type safety for allocation

No statement about allocation!

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

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

`r1 = x;`
`if (!b)  r1 = new C();`
`y = r1;`
	`r2 = y;`
	`x = r2`
		`b = true;`

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

. . . 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 . . .

```
            initially: b = false; x = y = null;
--------------------------------------------------------------
r1 = x;                         |    r2 = y;    |   b = true;
if (!b)  r1 = new C();          |    x = r2     |
y = r1;                         |               |
        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.

Beyond type safety [TOPLAS 2014]

Goals of the Java memory model:

Type safety **holds** despite forging of references

James Gosling • Bill Joy • Guy Steele • Gilad Bracha
The Java™ Language Specification, Third Edition
The Java™ Series
Sun
...from the Source™
Isabelle
HOL
∀
=
λ
β
α
→

Beyond type safety [TOPLAS 2014]

Goals of the Java memory model:

Type safety **holds** despite forging of references

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

Main reason for technical complexity

James Gosling • Bill Joy • Guy Steele • Gilad Bracha
The Java™ Language Specification, Third Edition
The Java™ Series
Sun
...from the Source™
Isabelle HOL

Beyond type safety [TOPLAS 2014]

Goals of the Java memory model:

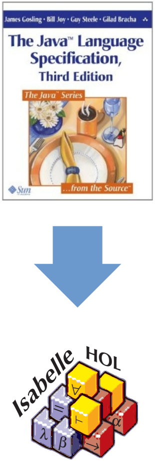
Type safety **holds** despite forging of references

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

Main reason for technical complexity

Security architecture (sandboxing)

compromised by forged references


James Gosling • Bill Joy • Guy Steele • Gilad Bracha
The Java™ Language Specification, Third Edition
The Java™ Series
Sun
...from the Source™
Isabelle HOL

Beyond type safety [TOPLAS 2014]

Goals of the Java memory model:

Type safety **holds** despite forging of references

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

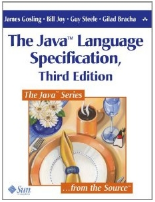
Main reason for technical complexity

Security architecture (sandboxing)

compromised by forged references

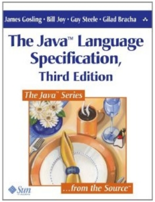
DRF guarantee

Interleaving semantics for programs without data races **proved**.


James Gosling · Bill Joy · Guy Steele · Gilad Bracha
The Java™ Language Specification, Third Edition
The Java™ Series
Sun
...from the Source™

Beyond type safety [TOPLAS 2014]

Goals of the Java memory model:


James Gosling · Bill Joy · Guy Steele · Gilad Bracha
The Java™ Language Specification, Third Edition
The Java™ Series
Sun
...from the Source™

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

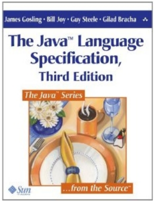
DRF guarantee

Interleaving semantics for programs without data races **proved**.

Compiler optimisations [Ševčík et al.]

JMM **fails** to allow common optimisations.

Beyond type safety [TOPLAS 2014]


James Gosling • Bill Joy • Guy Steele • Gilad Bracha
The Java™ Language Specification, Third Edition
The Java™ Series
Sun
...from the Source

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).