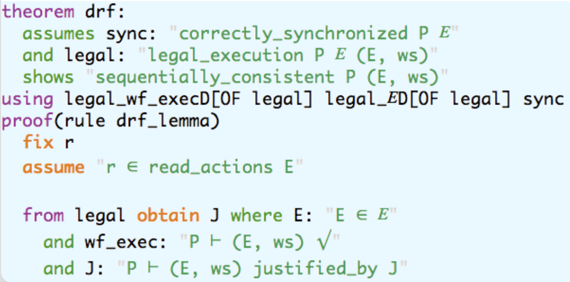
Karlsruhe Institute of Technology

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"
```

KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.kit.edu

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?

interleaving semantics

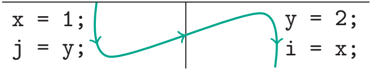
initially: `x = y = 0;`

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

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

Why do we need a memory model?

initially: `x = y = 0;`

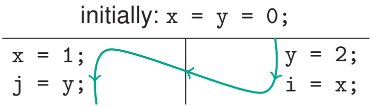


`x = 1;`	`y = 2;`
`j = y;`	`i = x;`

interleaving semantics

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

Why do we need a memory model?



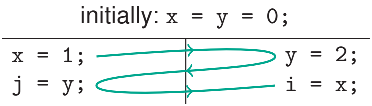
initially: `x = y = 0;`

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

interleaving semantics

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

Why do we need a memory model?



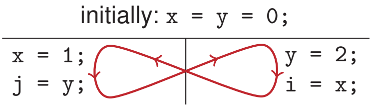
initially: `x = y = 0;`

`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;`

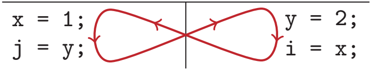
`x = 1;` `j = y;`	`y = 2;` `i = x;`

interleaving semantics

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

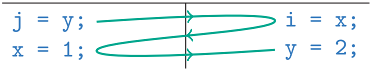
Why do we need a memory model?

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;`

interleaving semantics

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

	`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`	X	✓
`i == 1`	✓	✓

compiler and hardware reorder statements

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

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

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`	✓	✓

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

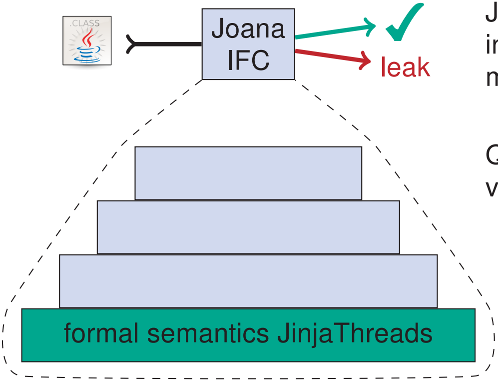
1. allow compiler optimisations
   too restricted
   [Cenciarelli et al. 07; Ševčík, Aspinall 08; Torlak et al. 10]

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

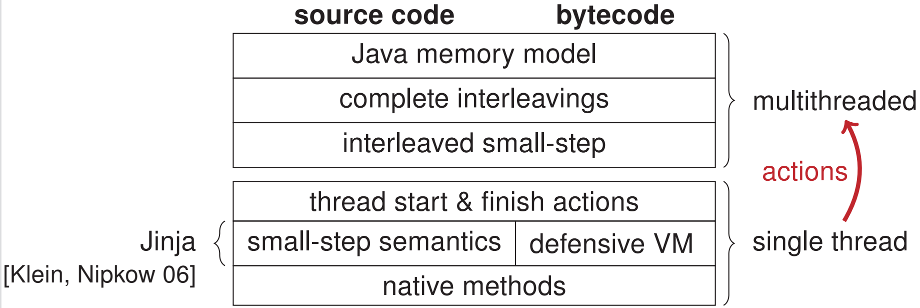
JinjaThreads

sequential features

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

concurrency

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



source code **bytecode**

Java memory model

complete interleavings

interleaved small-step

thread start & finish actions

small-step semantics | defensive VM

native methods

multithreaded

actions

single thread

Jinja [Klein, Nipkow 06]

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

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	…, t1:[Init y 0], …
	2. allocation	t1:[Init t2's fields]
	3. execute constructor	t1:[], …
	4. spawn	t1:[], t1:[Spawn t2], t1:[]
	5. start	t2:[Start]
	6. read y	t2:[Read y *v*]
	7. print y	t2:[External print *v*]
	8. finish	t2:[Finish]
	9. join	t1:[NotInterrupted t1, Join t2]
	10. finish	t1:[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	. . . , `t1`:[Init `y` 0], . . .
→ 2. allocation	`t1`:[Init `t2`'s fields]
3. execute constructor	`t1`:[], . . .
4. spawn	`t1`:[], `t1`:[Spawn `t2`], `t1`:[]
5. start	`t2`:[Start]
6. read `y`	`t2`:[Read `y` *v*]
7. print `y`	`t2`:[External print *v*]
8. finish	`t2`:[Finish]
9. join	`t1`:[NotInterrupted `t1`, Join `t2`]
10. finish	`t1`:[Finish]

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

A. interleave threads and	record actions
1. bootstrap	. . . , `t1`:[Init `y` 0], . . .
2. allocation	`t1`:[Init `t2`'s fields]
→ 3. execute constructor	`t1`:[], . . .
4. spawn	`t1`:[], `t1`:[Spawn `t2`], `t1`:[]
5. start	`t2`:[Start]
6. read `y`	`t2`:[Read `y` *v*]
7. print `y`	`t2`:[External print *v*]
8. finish	`t2`:[Finish]
9. join	`t1`:[NotInterrupted `t1`, Join `t2`]
10. finish	`t1`:[Finish]

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

A. interleave threads and record actions

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

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

A. interleave threads and record actions

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

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

A. interleave threads and	record actions
1. bootstrap	. . . , `t1`:[Init `y` 0], . . .
2. allocation	`t1`:[Init `t2`'s fields]
3. execute constructor	`t1`:[], . . .
4. spawn	`t1`:[], `t1`:[Spawn `t2`], `t1`:[]
5. start	`t2`:[Start]
→ 6. read `y`	`t2`:[Read `y` *v*]
7. print `y`	`t2`:[External print *v*]
8. finish	`t2`:[Finish]
9. join	`t1`:[NotInterrupted `t1`, Join `t2`]
10. finish	`t1`:[Finish]

non-deterministic value *v*

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

A. interleave threads and record actions

interleave threads	record actions
1. bootstrap	. . . , `t1`:[Init `y` 0], . . .
2. allocation	`t1`:[Init `t2`'s fields]
3. execute constructor	`t1`:[], . . .
4. spawn	`t1`:[], `t1`:[Spawn `t2`], `t1`:[]
5. start	`t2`:[Start]
6. read `y`	`t2`:[Read `y` *v*]
→ 7. print `y`	`t2`:[External print *v*]
8. finish	`t2`:[Finish]
9. join	`t1`:[NotInterrupted `t1`, Join `t2`]
10. finish	`t1`:[Finish]

non-deterministic value *v*

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

A. interleave threads and record actions

A. interleave threads	record actions
1. bootstrap	. . . , `t1`:[Init `y` 0], . . .
2. allocation	`t1`:[Init `t2`'s fields]
3. execute constructor	`t1`:[], . . .
4. spawn	`t1`:[], `t1`:[Spawn `t2`], `t1`:[]
5. start	`t2`:[Start]
6. read `y`	`t2`:[Read `y` *v*]
7. print `y`	`t2`:[External print *v*]
→ 8. finish	`t2`:[Finish]
9. join	`t1`:[NotInterrupted `t1`, Join `t2`]
10. finish	`t1`:[Finish]

non-deterministic value *v*

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

A. interleave threads and	record actions
1. bootstrap	. . . , `t1`:[Init `y` 0], . . .
2. allocation	`t1`:[Init `t2`'s fields]
3. execute constructor	`t1`:[], . . .
4. spawn	`t1`:[], `t1`:[Spawn `t2`], `t1`:[]
5. start	`t2`:[Start]
6. read `y`	`t2`:[Read `y` v]
7. print `y`	`t2`:[External print v]
8. finish	`t2`:[Finish]
→ 9. join	`t1`:[NotInterrupted `t1`, Join `t2`]
10. finish	`t1`:[Finish]

non-deterministic value v

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

A. interleave threads and record actions

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

non-deterministic value *v*

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

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

A. interleave threads and record actions

B. flatten & purge irrelevant actions

. . . , `t1`: Init `y` 0 , . . .
`t1`: Init `t2`'s fields
. . .
`t1`: Spawn `t2`
`t2`: Start
`t2`: Read `y` *v*
`t2`: External print *v*
`t2`: Finish
`t1`: Join `t2`
`t1`: Finish

non-deterministic value *v*

Connecting JinjaThreads with the JMM

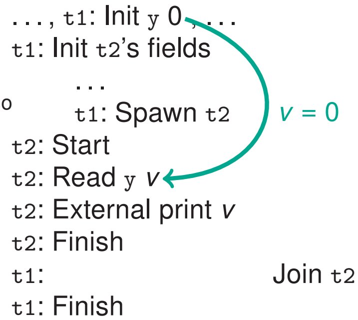
initially: `y = 0;`

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

A. interleave threads and record actions

B. flatten & purge irrelevant actions

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



…, `t1`: Init `y` 0, …

`t1`: Init `t2`'s fields

…

`t1`: Spawn `t2`

`t2`: Start

`t2`: Read `y` v

`t2`: External print v

`t2`: Finish

`t1`: Join `t2`

`t1`: Finish

Connecting JinjaThreads with the JMM

initially: `y = 0;`

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

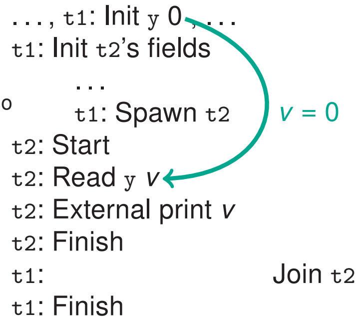
```
class T extends Thread {
    public void run() {
        print(y); } }
```

A. interleave threads and record actions

B. flatten & purge
irrelevant actions

C. reconstruct orders $\leq_{\text{hb}}$, $\leq_{\text{so}}$
match reads and writes

D. impose JMM
legality constraints



. . . , `t1`: Init `y` 0 , . . .
`t1`: Init `t2`'s fields
. . .
`t1`: Spawn `t2`
`t2`: Start
`t2`: Read `y` v
`t2`: External print v
`t2`: Finish
`t1`: Join `t2`
`t1`: Finish

$v = 0$

DRF guarantee

sequential consistency (SC) every read sees most recent write

data race two conflicting actions unrelated in $\leq_{\mathrm{hb}}$
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 $\Longrightarrow$ *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 = false; y = null;`

`x = true;`	`r1 = x;` `y = (r1 ? new A() : new B());`	`r2 = y.f();`

Implicit communication channels

1. run-time type information as global state

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

`x = true;`	`r1 = x;` `y = (r1 ? new A() : new B());`	`r2 = y.f();`

dispatch to `A.f()`
⇒ `r1 == true`

Implicit communication channels

1. run-time type information as global state

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

`x = true;`	`r1 = x;` `y = (r1 ? new A() : new B());`	`r2 = y.f();`

disallowed synchronisation

dispatch to `A.f()`
⇒ `r1 == true`

Implicit communication channels

1. run-time type information as global state

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

```
x = true; | r1 = x;                           | r2 = y.f();
          | y = (r1 ? new A() : new B());     |
```

disallowed synchronisation

dispatch to `A.f()`
⇒ `r1 == true`

2. synchronisation via `Thread.start`

initially: `x = new Thread(); y = 0;`

```
y = 1;       | try { x.start();
x.start();   | } catch (IllegalThreadStateException _) { r = y; }
```

Implicit communication channels

1. run-time type information as global state

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

`x = true;`	`r1 = x;` `y = (r1 ? new A() : new B());`	`r2 = y.f();`

disallowed synchronisation

dispatch to `A.f()`
⇒ `r1 == true`

2. synchronisation via `Thread.start`

initially: `x = new Thread(); y = 0;`

`y = 1;` `x.start();`	`try { x.start();` `} catch (IllegalThreadStateException _) { r = y; }`

data race?

Implicit communication channels

1. run-time type information as global state

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

`x = true;`	`r1 = x;` `y = (r1 ? new A() : new B());`	`r2 = y.f();`

disallowed synchronisation

dispatch to `A.f()`
⇒ `r1 == true`

2. synchronisation via `Thread.start`

initially: `x = new Thread(); y = 0;`

`y = 1;` `x.start();`	`try { x.start();` `} catch (IllegalThreadStateException _) { r = y; }`

data race?

intuition: no JMM: yes

Implicit communication channels

1. run-time type information as global state

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

`x = true;`	`r1 = x;` `y = (r1 ? new A() : new B());`	`r2 = y.f();`

dispatch to `A.f()`
⇒ `r1 == true`

disallowed synchronisation

2. synchronisation via `Thread.start`

initially: `x = new Thread(); y = 0;`

`y = 1;` `x.start();`	`try { x.start();` `} catch (IllegalThreadStateException _) { r = y; }`

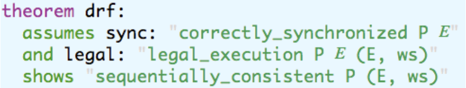
data race?

disallowed synchronisation

DRF guarantee

Theorem (DRF guarantee)

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



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

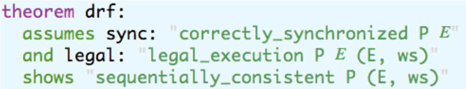
Java memory model
complete interleavings
interleaved small-step

single-thread semantics

DRF guarantee

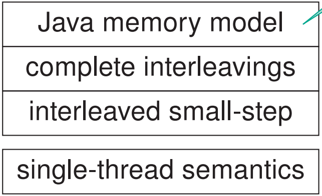
Theorem (DRF guarantee)

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



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

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

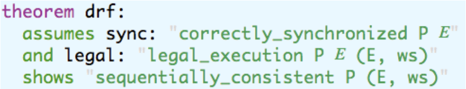


DRF guarantee

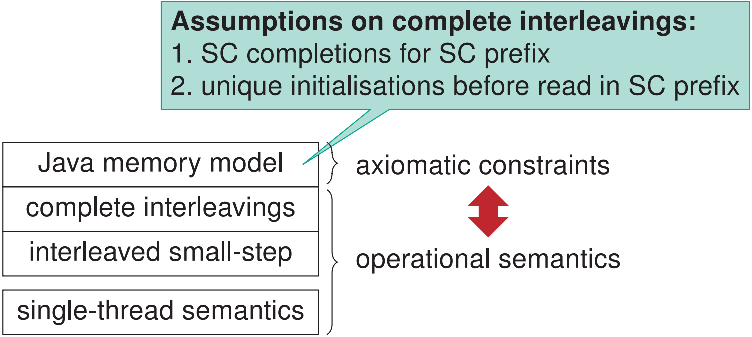
Theorem (DRF guarantee)

No data race in SC executions

$\Longrightarrow$ *all executions are SC.*



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



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

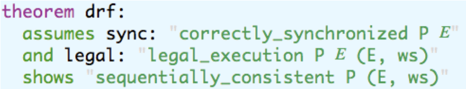
axiomatic constraints

operational semantics

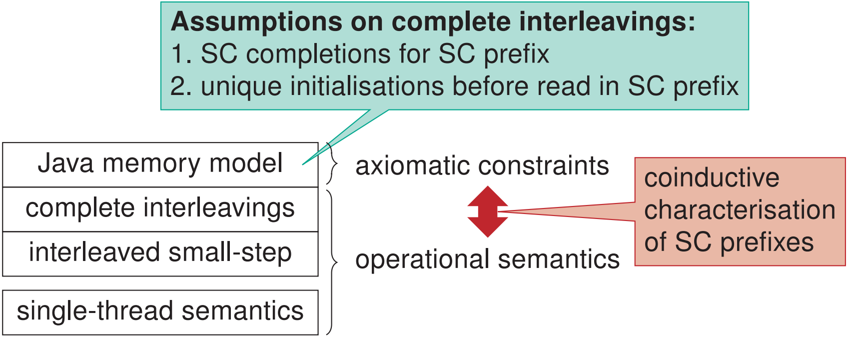
DRF guarantee

Theorem (DRF guarantee)

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



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



Assumptions on complete interleavings:

1. SC completions for SC prefix
2. unique initialisations before read in SC prefix

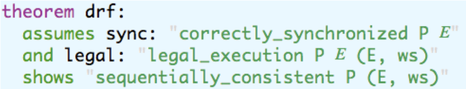
coinductive characterisation of SC prefixes

DRF guarantee

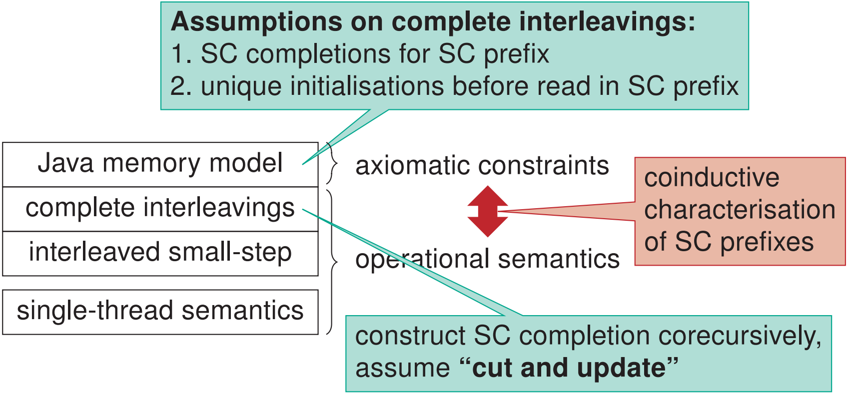
Theorem (DRF guarantee)

No data race in SC executions

$\Longrightarrow$ *all executions are SC.*



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



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

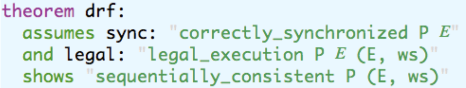
coinductive characterisation of SC prefixes

construct SC completion corecursively, assume “**cut and update**”

DRF guarantee

Theorem (DRF guarantee)

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



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

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

(

n

s

)

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