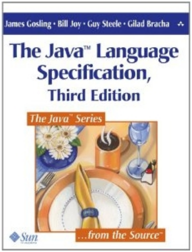
Mechanising a type-safe model of multithreaded Java with a verified compiler

Andreas Lochbihler

Digital Asset (Switzerland) GmbH



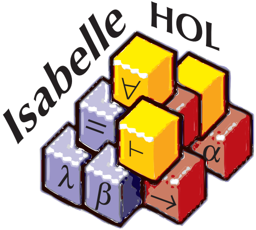
James Gosling • Bill Joy • Guy Steele • Gilad Bracha

The Java™ Language Specification, Third Edition

The Java™ Series

...from the Source™

Sun



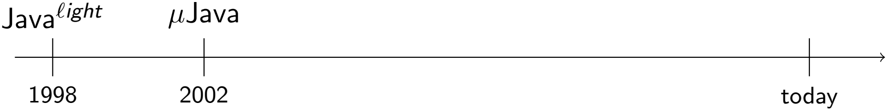
Timeline

Java$^{\ell ight}$

1998

today

Timeline



Java$^{\ell ight}$

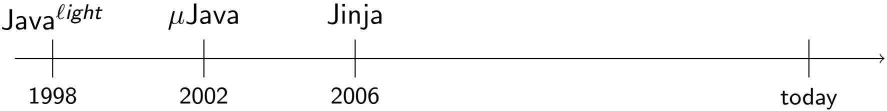
μJava

1998

2002

today

Timeline



Java$^{\ell ight}$

μJava

Jinja

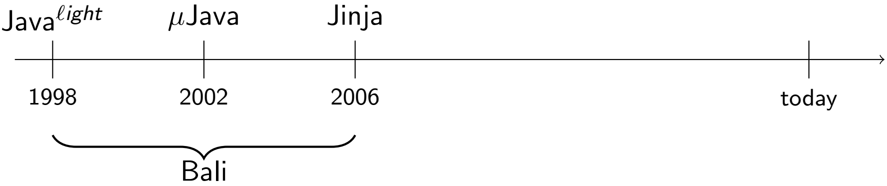
1998

2002

2006

today

Timeline



Java$^{\ell ight}$

μJava

Jinja

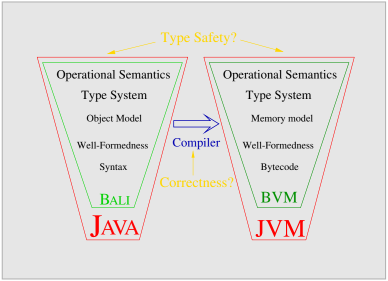
1998

2002

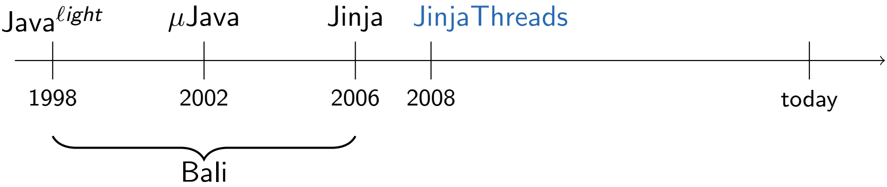
2006

today

Bali



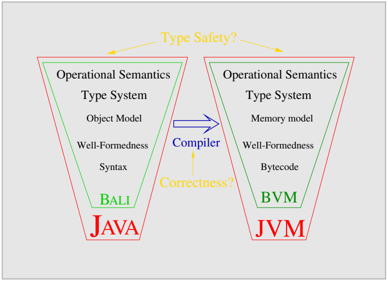
Timeline



Java$^{\ell ight}$ μJava Jinja JinjaThreads

1998 2002 2006 2008 today

Bali

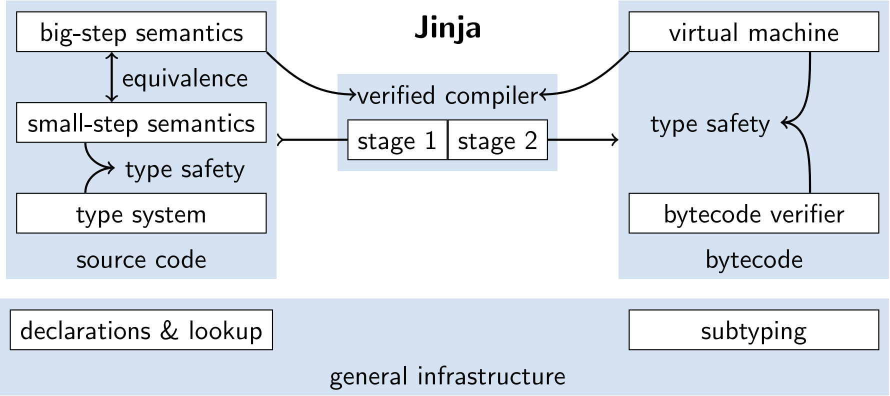


+

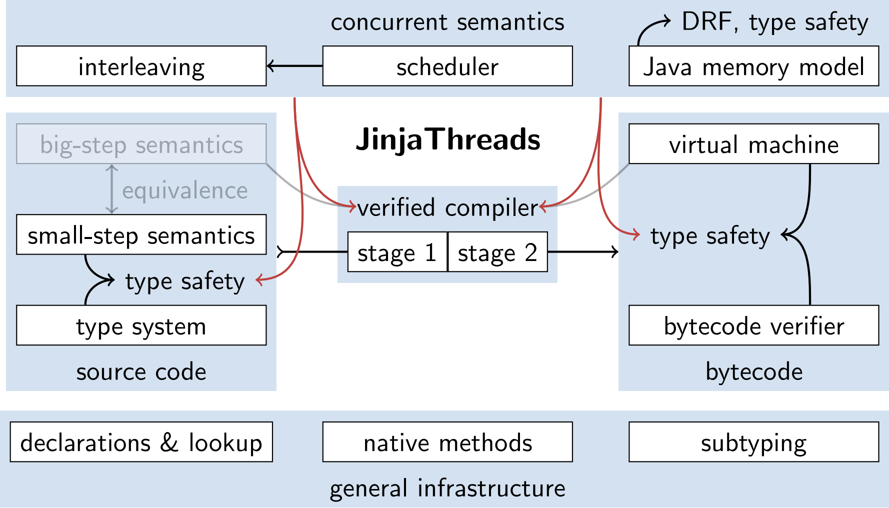
Concurrency

- threads
- synchronisation mechanisms
- Java memory model

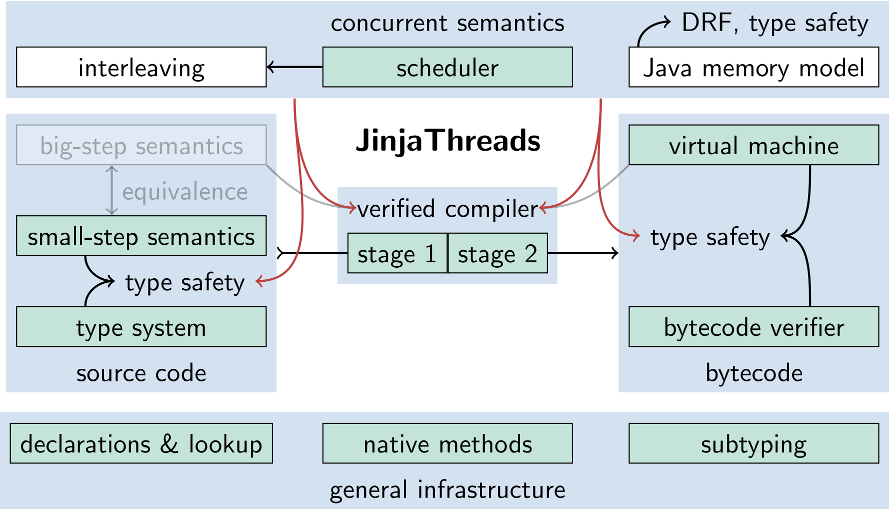
From Jinja to JinjaThreads



From Jinja to JinjaThreads

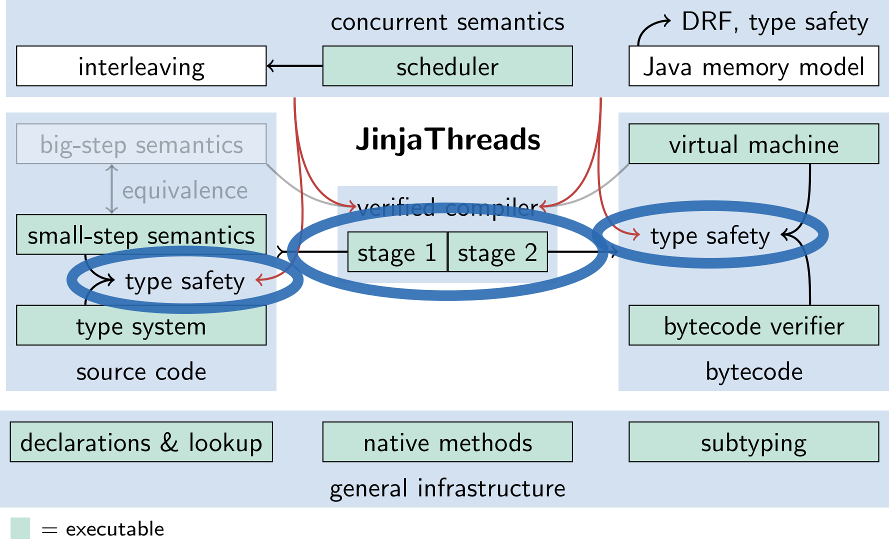


From Jinja to JinjaThreads



🟩 = executable

From Jinja to JinjaThreads



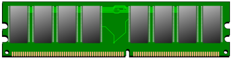
Semantics in layers

Java memory model

set of well-formed
candidate executions

operational
semantics

shared
memory



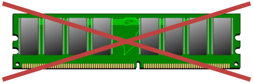
Semantics in layers

Java memory model

set of well-formed
candidate executions

operational
semantics

~~shared memory~~



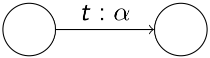
allocation &
type information

Semantics in layers

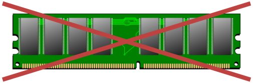
Java memory model

set of well-formed
candidate executions

operational
semantics



~~shared
memory~~



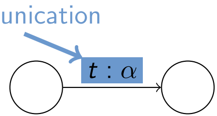
allocation &
type information

Semantics in layers

Java memory model

set of well-formed
candidate executions

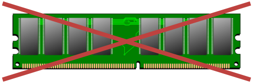
operational
semantics



thread communication

$t : \alpha$

~~shared memory~~



allocation &
type information

Semantics in layers

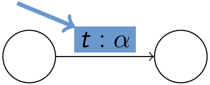
Java memory model

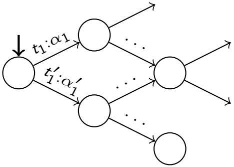
set of well-formed
candidate executions

transition system

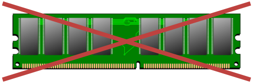
thread communication

operational
semantics





~~shared
memory~~



allocation &
type information

Semantics in layers

Java memory model

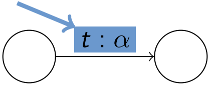
set of well-formed candidate executions

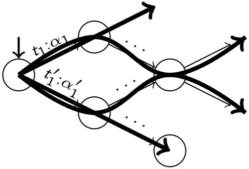
$$\begin{aligned}\{\, &[t_1 : \alpha_1, t_2 : \alpha_2, \ldots],\\ &[t'_1 : \alpha'_1, t'_2 : \alpha'_2, \ldots],\\ &[t''_1 : \alpha''_1, t''_2 : \alpha''_2, \ldots], \ldots \,\}\end{aligned}$$

paths in the transition system

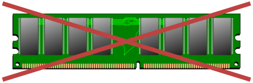
thread communication

operational semantics





~~shared memory~~



allocation & type information

Semantics in layers

Java memory model

legality constraints
pair read and write ops

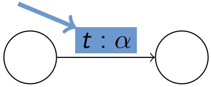
set of well-formed
candidate executions

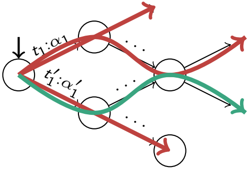
$\{$ ~~$[t_1 : \alpha_1, t_2 : \alpha_2, \ldots]$~~,
$[t'_1 : \alpha'_1, t'_2 : \alpha'_2, \ldots]$, ⟵ legal
~~$[t''_1 : \alpha''_1, t''_2 : \alpha''_2, \ldots]$~~, $\ldots$ $\}$

paths in the
transition system

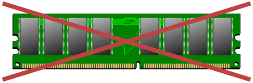
thread communication

operational
semantics





~~shared memory~~



allocation &
type information

type safety = progress + preservation

Let's reuse!

Jinja	Reuse the sequential type safety proof
Lifting	Use same lifting lemmas for source code and bytecode

type safety = progress + preservation

No stuck states

What about deadlocks?

Let's reuse!

Jinja Reuse the sequential type safety proof

Lifting Use same lifting lemmas for source code and bytecode

Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    …
    g.wait();
    …
  }
}
```

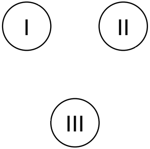
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    …
    g.notify();
    …
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    …
    …
    …
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:			



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Request lock on `f`

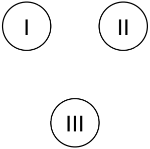
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:			



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    …
    g.wait();
    …
  }
}
```

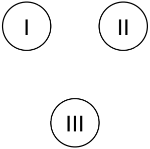
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    …
    g.notify();
    …
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    …
    …
    …
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Thread (II)

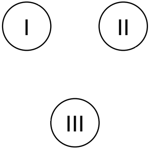
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Request lock on g

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

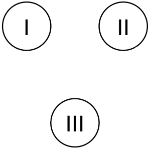
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	II



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Thread (II)

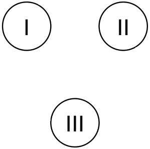
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on e

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	II



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

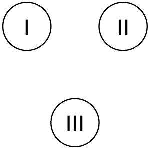
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:	III	I	II



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Request lock on g

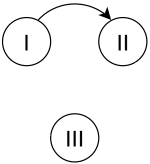
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:	III	I	II



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Request lock on g

Thread (II)

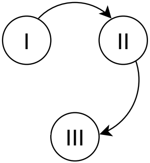
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Request lock on e

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:	III	I	II



Java deadlock example

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    …
    g.wait();
    …
  }
}
```

Request lock on g

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    …
    g.notify();
    …
  }
}
```

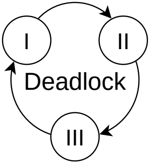
Request lock on e

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    …
    …
    …
  }
}
```

Request lock on f

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:	III	I	II



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

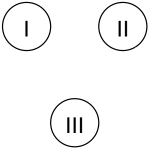
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:			



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Request lock on `f`

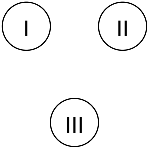
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:			



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

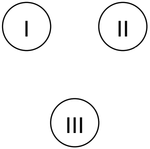
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Request lock on g

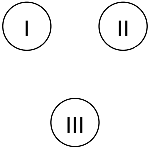
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

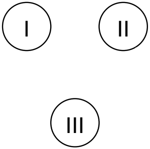
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	I



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ●●●
    g.wait();
    …
  }
}
```

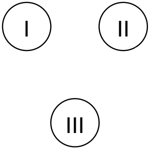
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    …
    g.notify();
    …
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    …
    …
    …
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{}
Locked by:		I	I



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

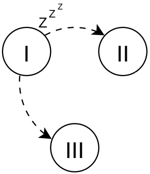
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:		I	



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

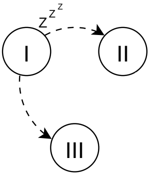
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on e

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:		I	



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

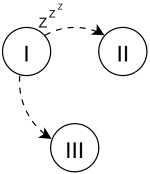
Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

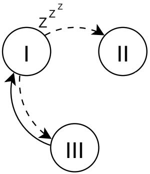
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `f`

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

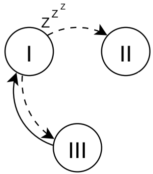
Request lock on g

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `f`

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

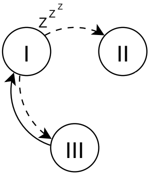
```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

Request lock on `f`

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	II



Java deadlock example with monitors

Thread (I)

```
synchronized (f) {
  synchronized (g) {
    ...
    g.wait();
    ...
  }
}
```

Wait on notify

Thread (II)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

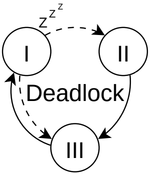
Request lock on e

Thread (III)

```
synchronized (e) {
  synchronized (f) {
    ...
    ...
    ...
  }
}
```

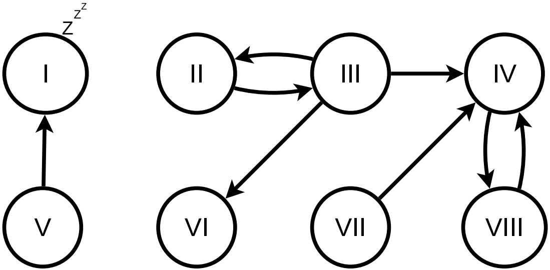
Request lock on f

Objects	e	f	g
Wait set:	{}	{}	{I}
Locked by:	III	I	II



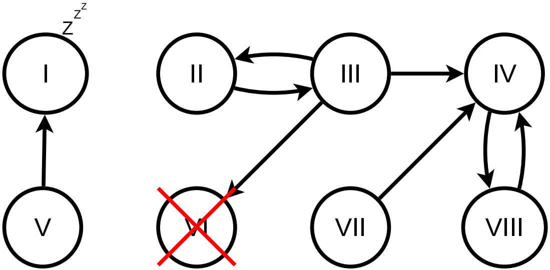
Deadlock computation

Deadlock as a greatest fixpoint:



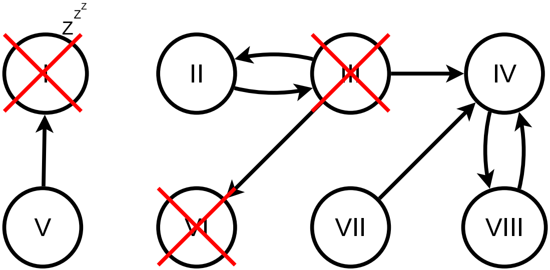
Deadlock computation

Deadlock as a greatest fixpoint:



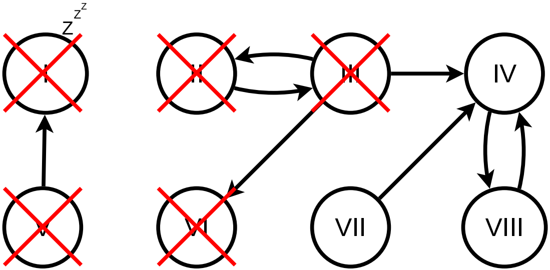
Deadlock computation

Deadlock as a greatest fixpoint:



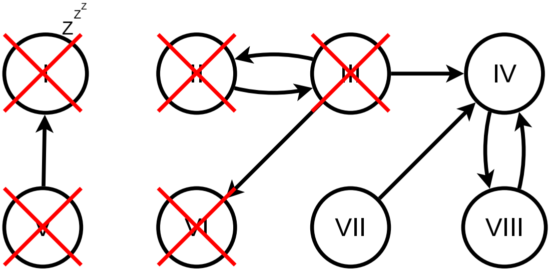
Deadlock computation

Deadlock as a greatest fixpoint:



Deadlock computation

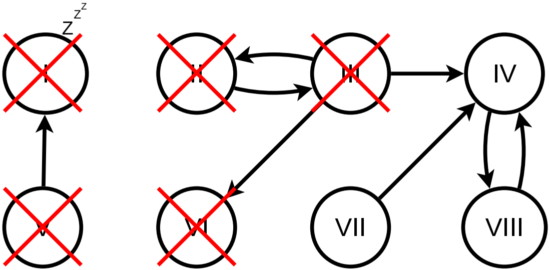
Deadlock as a greatest fixpoint:



Threads in deadlock: IV, VII, VIII

Deadlock computation

Deadlock as a greatest fixpoint:

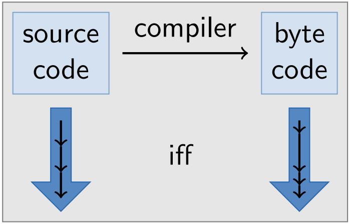


Threads in deadlock: IV, VII, VIII

Formalised as a coinductive definition, independent of the language!

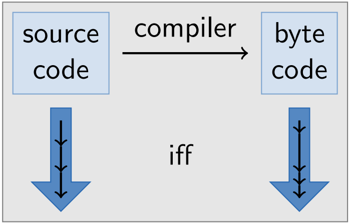
Compiler correctness

Compiler correctness



Compiler correctness

Compiler correctness

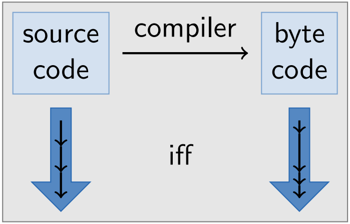


Trace behaviour:

- result state
- non-termination
- deadlock
- I/O

Compiler correctness

Compiler correctness



⟸

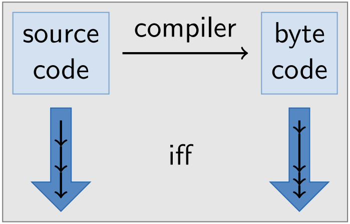
Delay bisimulation ≈ with divergence

Trace behaviour:

- result state
- non-termination
- deadlock
- I/O

Compiler correctness

Compiler correctness



⟸

Delay bisimulation ≈ with divergence

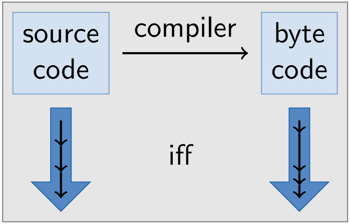
$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d, "\tau"] & s_2 \arrow[d, dashed, "\tau", "*"'] \\
s_1' \arrow[r, dash, dashed, "\approx"] & s_2'
\end{tikzcd}$$

Trace behaviour:

- result state
- non-termination
- deadlock
- I/O

Compiler correctness

Compiler correctness



$\Longleftarrow$

Delay bisimulation $\approx$ with divergence

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d, "\tau"] & s_2 \arrow[d, dashed, "\tau *"] \\
s_1' \arrow[r, dashed, dash, "\approx"] & s_2'
\end{tikzcd}$$

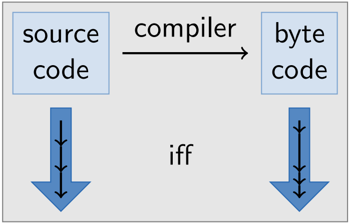
$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[dd, "\alpha_1"] & s_2 \arrow[d, dashed, "\tau *"] \\
{} \arrow[r, dashed, dash, "\approx"] & {} \arrow[d, dashed, "\alpha_2"] \\
s_1' \arrow[r, dashed, dash, "\approx"] & s_2'
\end{tikzcd}$$

Trace behaviour:

- result state
- non-termination
- deadlock
- I/O

Compiler correctness

Compiler correctness



source code

compiler

byte code

iff

$\Longleftarrow$

Delay bisimulation $\approx$ with divergence

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d, "\tau"] & s_2 \arrow[d, dashed, "\tau *"] \\
s_1' \arrow[r, dash, dashed, "\approx"] & s_2'
\end{tikzcd}$$

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[dd, "\alpha_1"] & s_2 \arrow[d, dashed, "\tau *"] \\
{} \arrow[r, dash, dashed, "\approx"] & {} \arrow[d, dashed, "\alpha_2"] \\
s_1' \arrow[r, dash, dashed, "\approx"] & s_2'
\end{tikzcd}$$

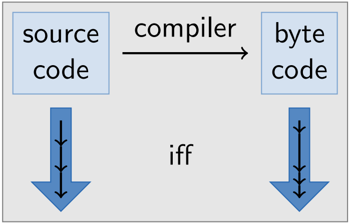
$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d, "\tau"] & s_2 \arrow[d, dashed, "\tau"] \\
\infty & \infty
\end{tikzcd}$$

Trace behaviour:

- result state
- non-termination
- deadlock
- I/O

Compiler correctness

Compiler correctness



source code

compiler

byte code

iff

⟸

Delay bisimulation ≈ with divergence

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d, "\tau"] & s_2 \arrow[d, dashed, "\tau", "*" near end] \\
s_1' \arrow[r, dash, dashed, "\approx"] & s_2'
\end{tikzcd}$$

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[dd, "\alpha_1"] & s_2 \arrow[d, dashed, "\tau", "*" near end] \\
{} \arrow[r, dash, dashed, "\approx"] & {} \arrow[d, dashed, "\alpha_2"] \\
s_1' \arrow[r, dash, dashed, "\approx"] & s_2'
\end{tikzcd}$$

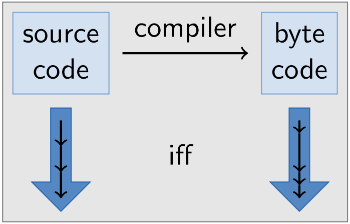
$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d, "\tau"] & s_2 \arrow[d, dashed, "\tau"] \\
\infty & \infty
\end{tikzcd}$$

Observable:
- I/O
- memory access
- allocation
- synchronisation

Trace behaviour:
- result state
- non-termination
- deadlock
- I/O

Compiler correctness

Compiler correctness



⟸

Delay bisimulation ≈ with divergence

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[d, "\tau"] & s_2 \arrow[d, dashed, "\tau *"] \\
s_1' \arrow[r, dashed, dash, "\approx"] & s_2'
\end{tikzcd}$$

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\approx"] \arrow[dd, "\alpha_1"] & s_2 \arrow[d, dashed, "\tau *"] \\
{} \arrow[r, dashed, dash, "\approx"] & {} \arrow[d, dashed, "\alpha_2"] \\
s_1' \arrow[r, dashed, dash, "\approx"] & s_2'
\end{tikzcd}$$

Trace behaviour:

- result state
- non-termination
- deadlock
- I/O

Lifting theorem:

single-threaded d. bisim.

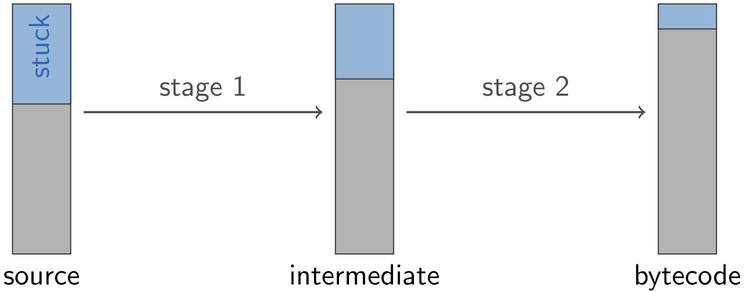
⇓

multi-threaded d. bisim.

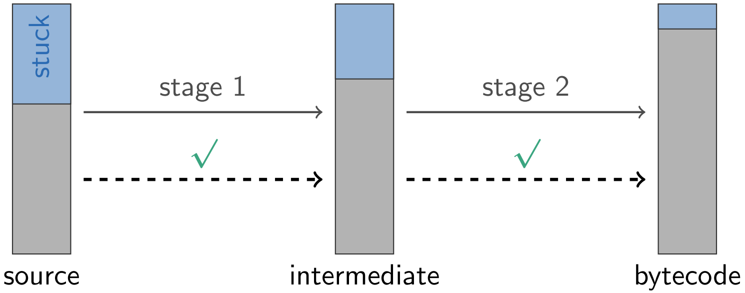
Observable:

- I/O
- memory access
- allocation
- synchronisation

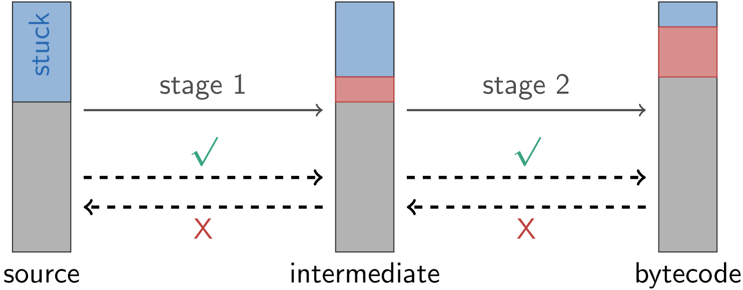
Stuck programs



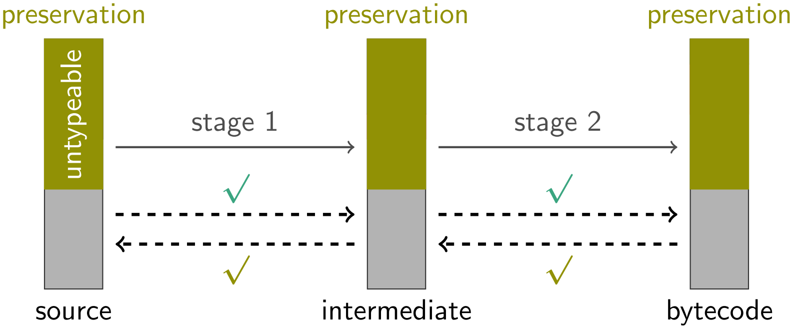
Stuck programs



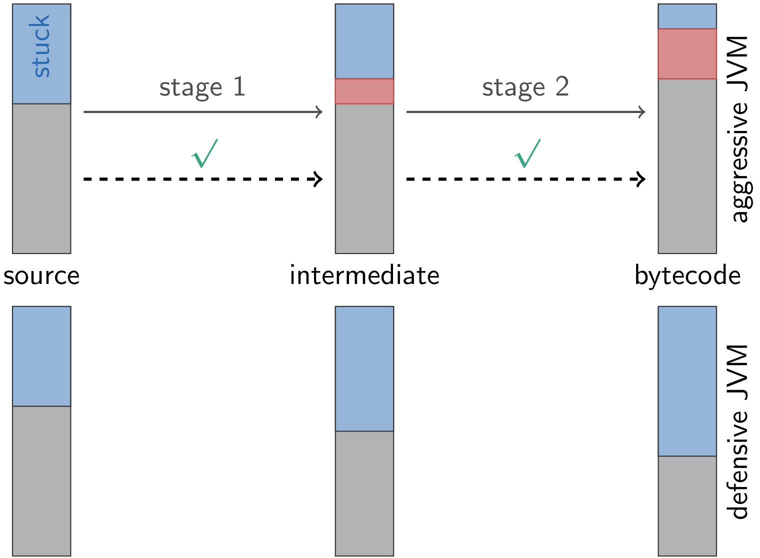
Stuck programs



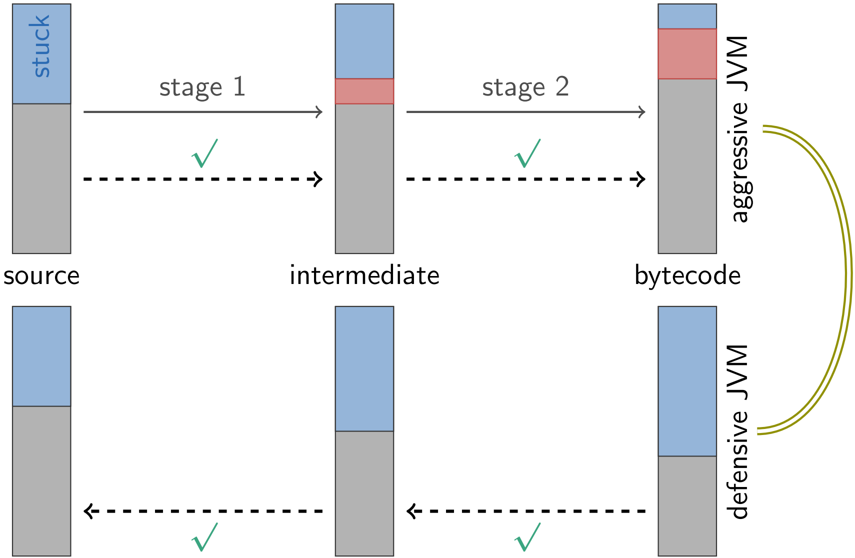
Stuck programs



Stuck programs



Stuck programs



Not covered in this talk

Java Memory Model formalised

- DRF guarantee: No data races $\Longrightarrow$ only interleaving behaviours
- Consistency: Every interleaving allowed
- Type safety even with data races
  if addresses are partitioned by type

Not covered in this talk

Java Memory Model formalised

- DRF guarantee: No data races $\Longrightarrow$ only interleaving behaviours
- Consistency: Every interleaving allowed
- Type safety even with data races
  if addresses are partitioned by type

Validation via code generation

- Executable interpreter, JVM, bytecode verifier, compiler
- Unverified converter from Java to abstract syntax
- Validated with 230+ test cases

Isabelle features

Used a lot:

- locales
- (co)inductive
- datatype, primrec, fun
- code generator
- nitpick
- auto & fastforce

Isabelle features

Used a lot:

- locales
- (co)inductive
- datatype, primrec, fun
- code generator
- nitpick
- auto & fastforce

Would have been great:

- Eisbach
- type parameters in locales
- codatatype & primcorec
- coinduction method
- refactoring support

Isabelle features

Used a lot:

- locales
- (co)inductive
- datatype, primrec, fun
- code generator
- nitpick
- auto & fastforce

Would have been great:

- Eisbach
- type parameters in locales
- codatatype & primcorec
- coinduction method
- refactoring support

Statistics:

- 89409 lines of code (10k empty lines)
- 567 definition, 101 (co)inductive, 124 primrec, 169 fun
- 4045 theorem statements
- 51 min AFP build time (factor 4.03, 64bit)