Karlsruhe Institute of Technology

Verifying a Compiler for Java Threads

Andreas Lochbihler

IPD, PROGRAMMING PARADIGMS GROUP, COMPUTER SCIENCE DEPARTMENT


```
text {* The compiler correctness theorem *}

theorem J2JVM_correct:
  fixes P C M vs
  defines "s ≡ J_start_state P C M vs" and "cs ≡ JVM_start_state (J2JVM P) C M vs"
  assumes "wf_J_prog P" "P ⊢ C sees M:Ts→T=(pns,body) in C" "length vs = length pns" "P,start_heap P ⊢ vs [:≤] Ts"
  shows "⟦ red_Tmthr.mthr.Trtrancl3p P s ttas s'; red_mthr.mfinal s' ⟧
       ⟹ ∃ttas'. mexecd_Tmthr.mthr.Trtrancl3p (J2JVM P) cs ttas' (mexception s') ∧
                 bisimulation_base.Tlsim (tlsimJ2JVM P) ttas ttas'"
  and   "⟦ mexecd_Tmthr.mthr.Trtrancl3p (J2JVM P) cs ttas' cs'; exec_mthr.mfinal cs' ⟧
       ⟹ ∃s' ttas. red_Tmthr.mthr.Trtrancl3p P s ttas s' ∧ mexception s' = cs' ∧
                   bisimulation_base.Tlsim (tlsimJ2JVM P) ttas ttas'"
  and   "red_Tmthr.mthr.Tinf_step P s Ttas
         ⟹ ∃Ttas'. mexecd_Tmthr.mthr.Tinf_step (J2JVM P) cs Ttas' ∧ bisimulation_base.Tlsiml (tlsimJ2JVM P) Ttas Ttas'"
  and   "mexecd_Tmthr.mthr.Tinf_step (J2JVM P) cs Ttas'
         ⟹ ∃Ttas. red_Tmthr.mthr.Tinf_step P s Ttas ∧ bisimulation_base.Tlsiml (tlsimJ2JVM P) Ttas Ttas'"
  and   "⟦ red_Tmthr.mthr.Trtrancl3p P s ttas s'; multithreaded_base.deadlock final_expr (mred P) s' ⟧
       ⟹ ∃cs' ttas'. mexecd_Tmthr.mthr.Trtrancl3p (J2JVM P) cs ttas' cs' ∧
                     multithreaded_base.deadlock JVM_final (mexecd (J2JVM P)) cs' ∧ bisimJ2JVM P s' cs' ∧
                     bisimulation_base.Tlsim (tlsimJ2JVM P) ttas ttas'"
```

Related work: formal compiler verification

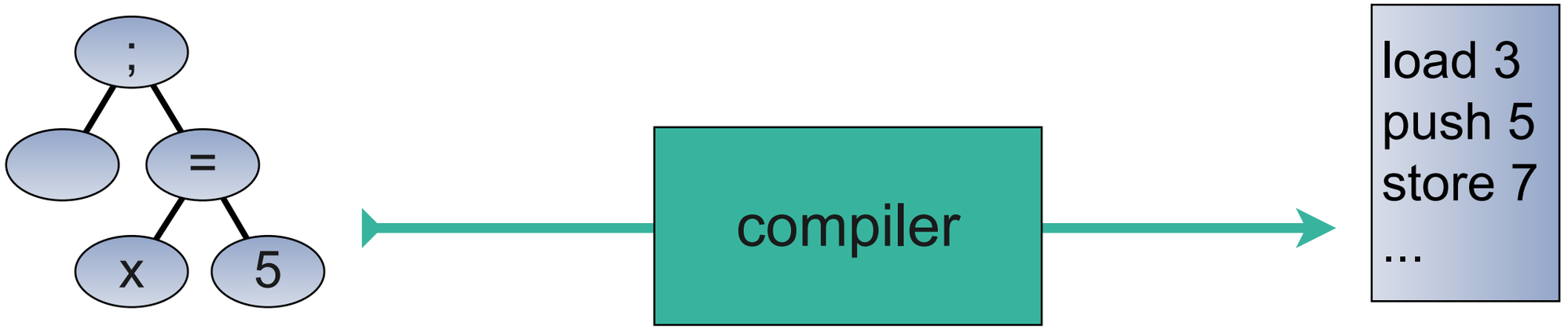
sequential languages

- Verisoft: from C0 to assembler [Leinenbach]
  - single pass, no optimisations
- CompCERT: from Cminor to assembler [Leroy]
  - many stages & optimisations
- Jinja: from Java to byte code [Klein, Nipkow]
  - two passes, no optimisations

concurrent languages

- parallel functional language [Rittri, Wand]
  - pen and paper proofs
- Concurrent Cminor [Appel et al.]
  - focus on separation logic
  - no compiler verification reported

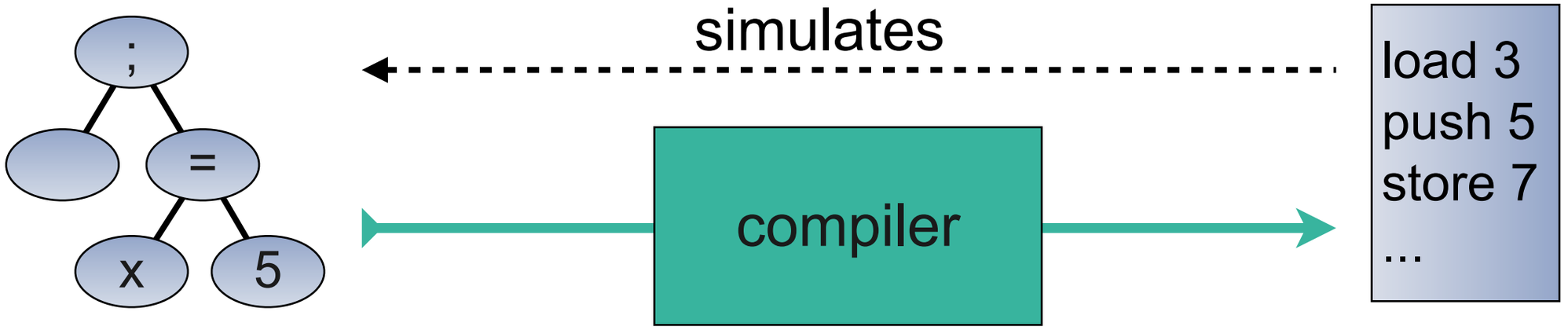
Sequential compiler verification



behaviour:

- result state / trace
- non-termination

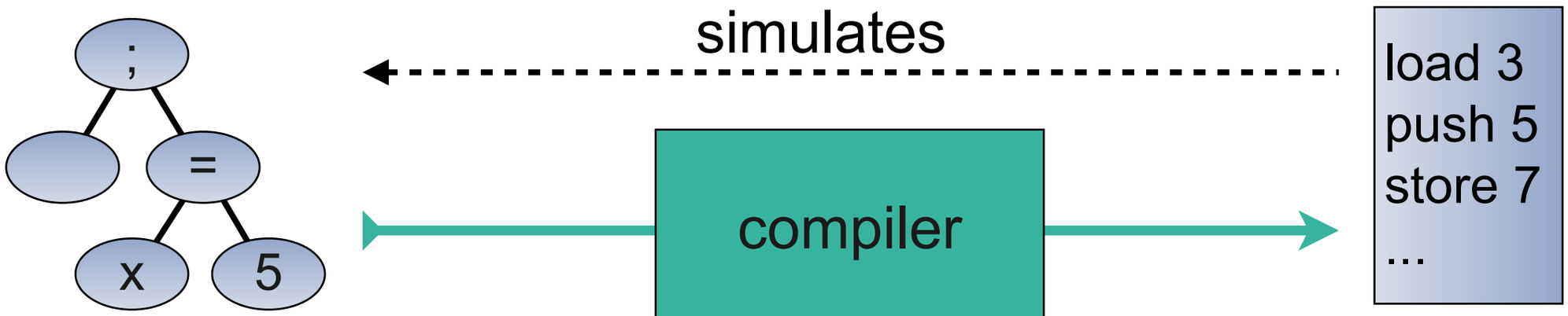
Sequential compiler verification



behaviour:

- result state / trace
- non-termination

Sequential compiler verification

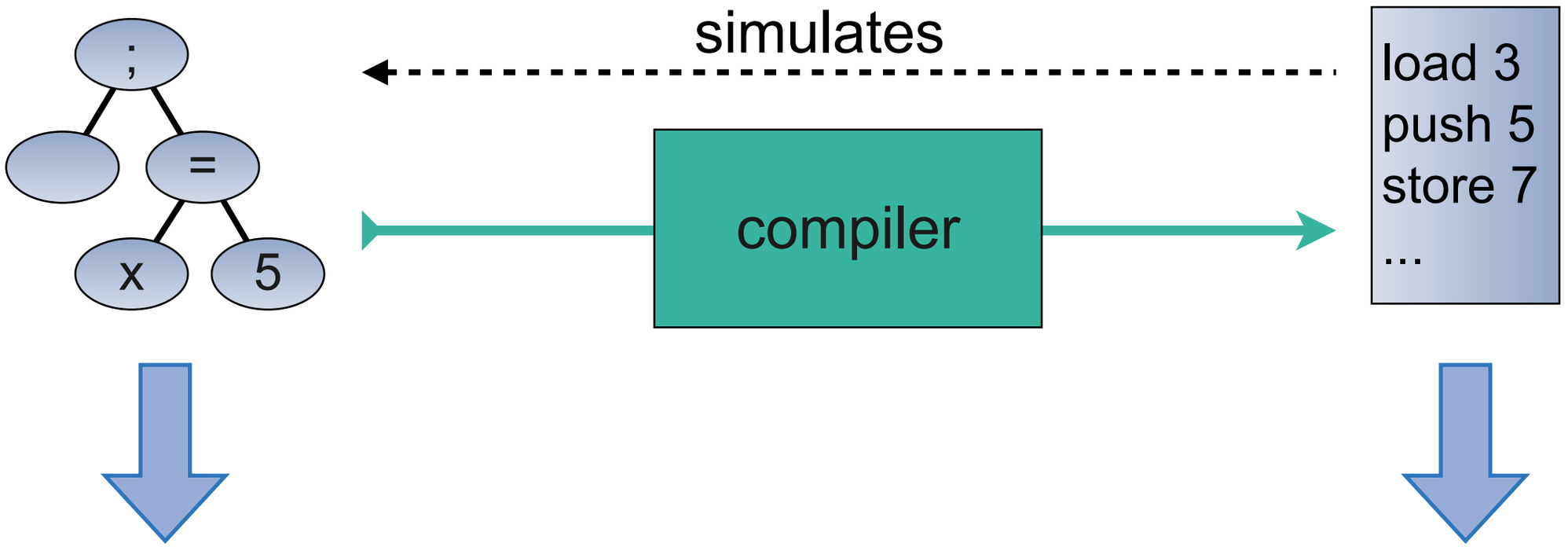


B

behaviour:

- result state / trace
- non-termination

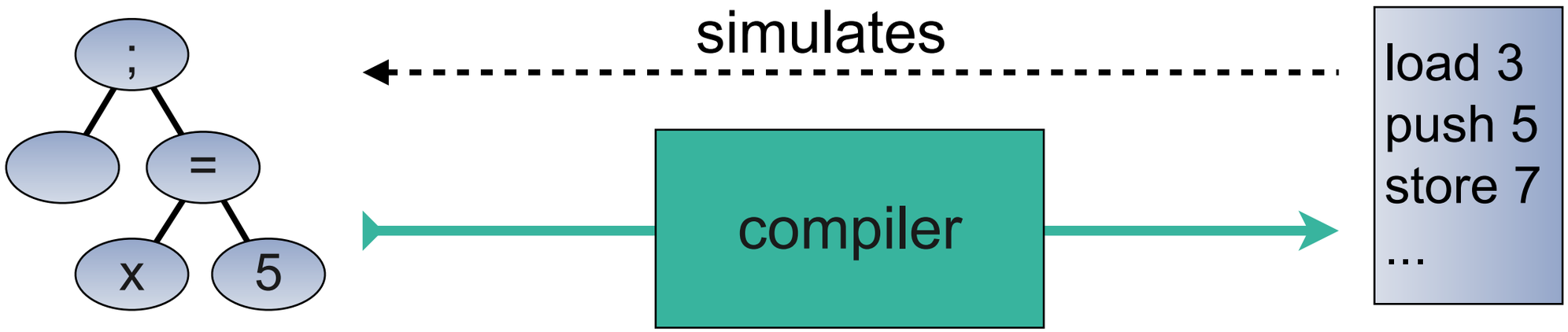
Sequential compiler verification



behaviour:

- result state / trace
- non-termination

Sequential compiler verification



B ≈ B'

behaviour:

- result state / trace
- non-termination

Simulation is not enough

```
synchronized (this) {
    this.x = this.x + 1;
}
```

||

```
synchronized (this) {
    this.x = 2;
}
```

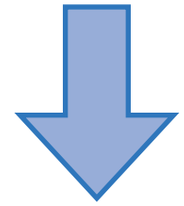
Simulation is not enough

```
synchronized (this) {
    this.x = this.x + 1;
}
```

||

```
synchronized (this) {
    this.x = 2;
}
```

```
this.x = 0
```



```
this.x = 2
this.x = 3
```

Simulation is not enough

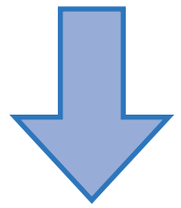
```
synchronized (this) {
    this.x = this.x + 1;
}
```

```
aload 0
dup
astore 1
monitorenter

aload 0
dup
getfield x
iconst_1
iadd
putfield x

aload 1
monitorexit
```

```
synchronized (this) {
    this.x = 2;
}
```

```
aload 0
dup
astore 1
monitorenter

aload 0
iconst_2
putfield x

aload 1
monitorexit
```

```
this.x = 0
```



```
this.x = 2
this.x = 3
```

Simulation is not enough

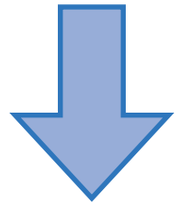
```
synchronized (this) {
    this.x = this.x + 1;
}
```

```
aload 0
dup
astore 1
monitorenter

aload 0
dup
getfield x
iconst_1
iadd
putfield x

aload 1
monitorexit
```

```
synchronized (this) {
    this.x = 2;
}
```

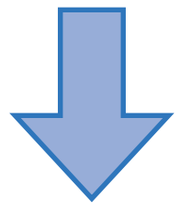
```
aload 0
dup
astore 1
monitorenter

aload 0
iconst_2
putfield x

aload 1
monitorexit
```

```
this.x = 0
```



```
this.x = 2
this.x = 3
```

```
this.x = 0
```



```
this.x = 2
this.x = 3
```

Simulation is not enough

```
synchronized (this) {
    this.x = this.x + 1;
}
```

```
aload 0
dup
astore 1
monitorenter

aload 0
dup
getfield x
iconst_1
iadd
putfield x

aload 1
monitorexit
```

```
aload 1
monitorexit
aload 1
monitorenter
```

```
synchronized (this) {
    this.x = 2;
}
```

```
aload 0
dup
astore 1
monitorenter

aload 0
iconst_2
putfield x

aload 1
monitorexit
```

```
this.x = 0

this.x = 2
this.x = 3

this.x = 0

this.x = 2
this.x = 3
```

Simulation is not enough

```
synchronized (this) {
    this.x = this.x + 1;
}
```

```
aload 0
dup
astore 1
monitorenter

aload 0
dup
getfield x
iconst_1
iadd
putfield x

aload 1
monitorexit
```

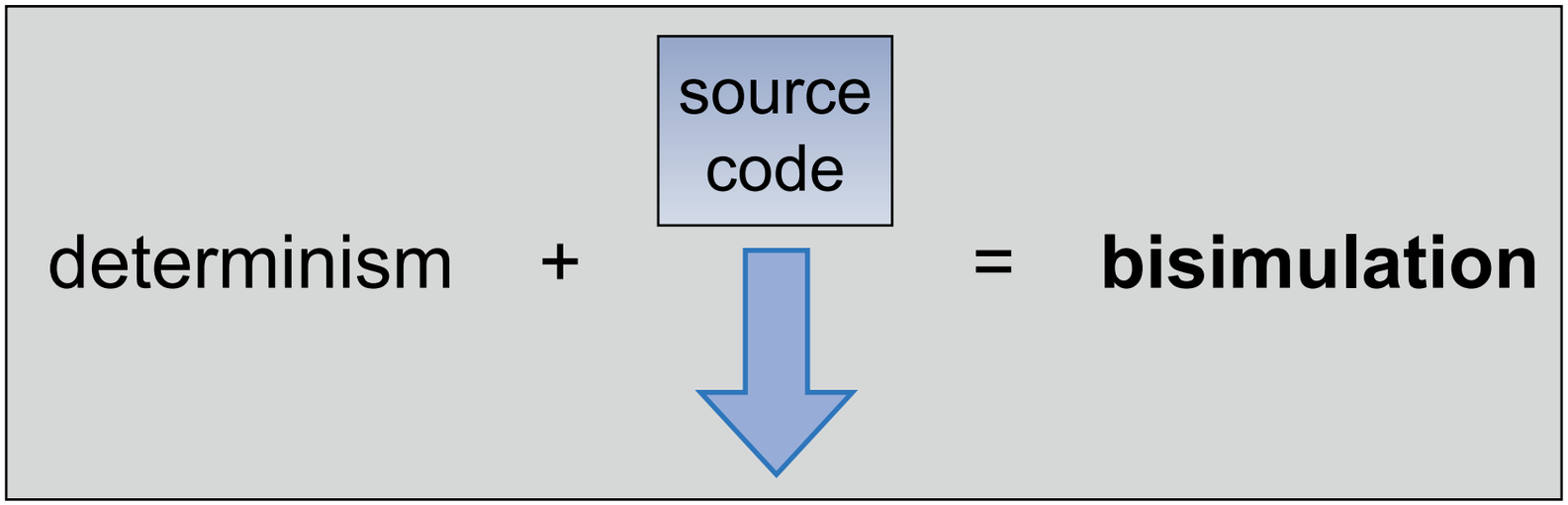
```
aload 1
monitorexit
aload 1
monitorenter
```

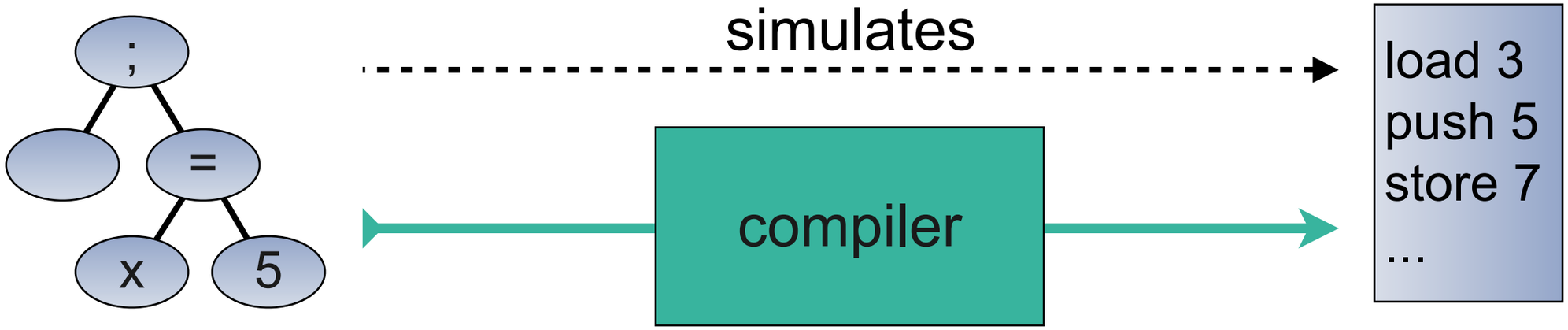
```
synchronized (this) {
    this.x = 2;
}
```

```
aload 0
dup
astore 1
monitorenter

aload 0
iconst_2
putfield x

aload 1
monitorexit
```

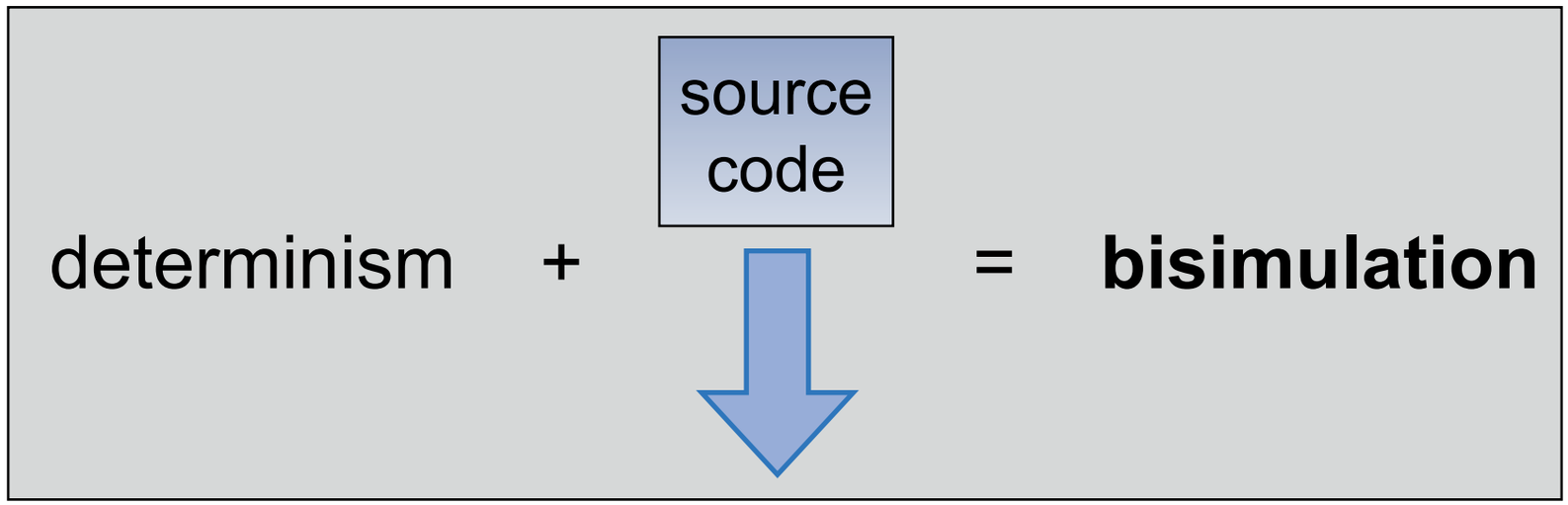
```
this.x = 0
this.x = 2
this.x = 3
this.x = 0
this.x = 2
this.x = 3
this.x = 1
```

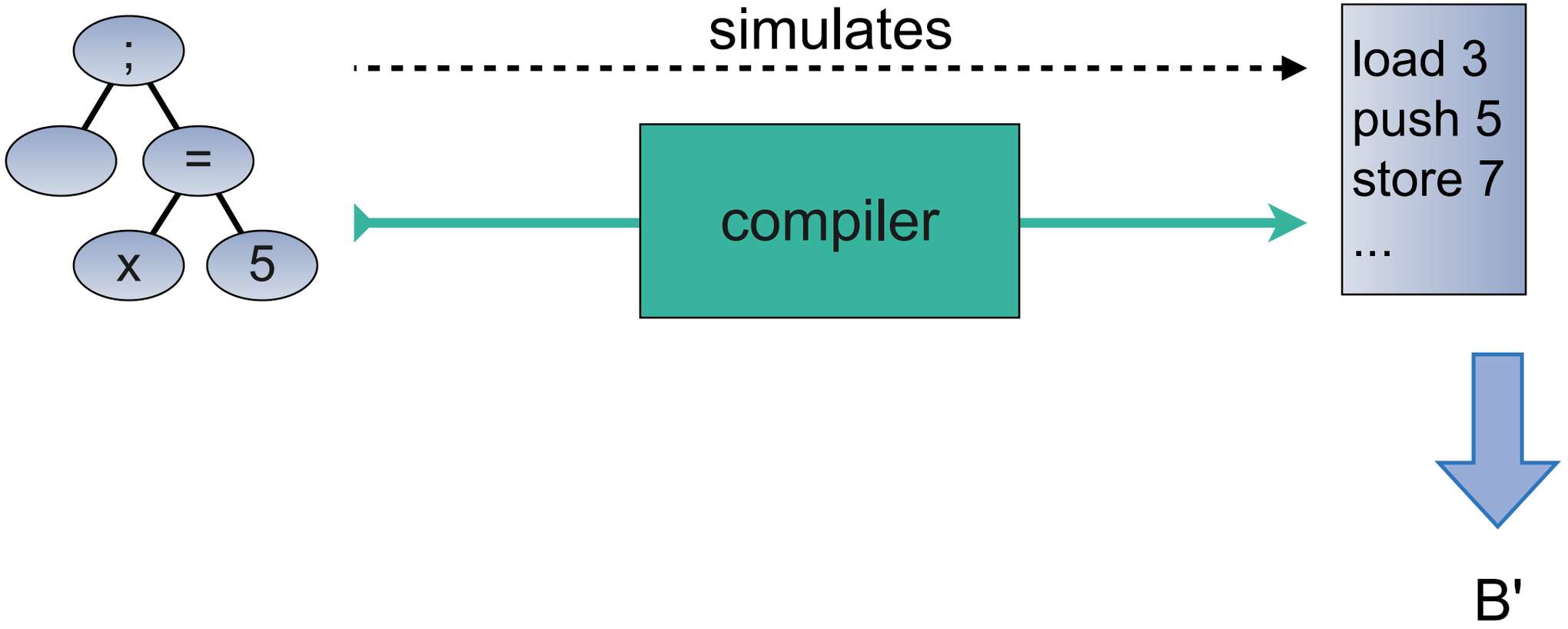
Why does it work for sequential languages?





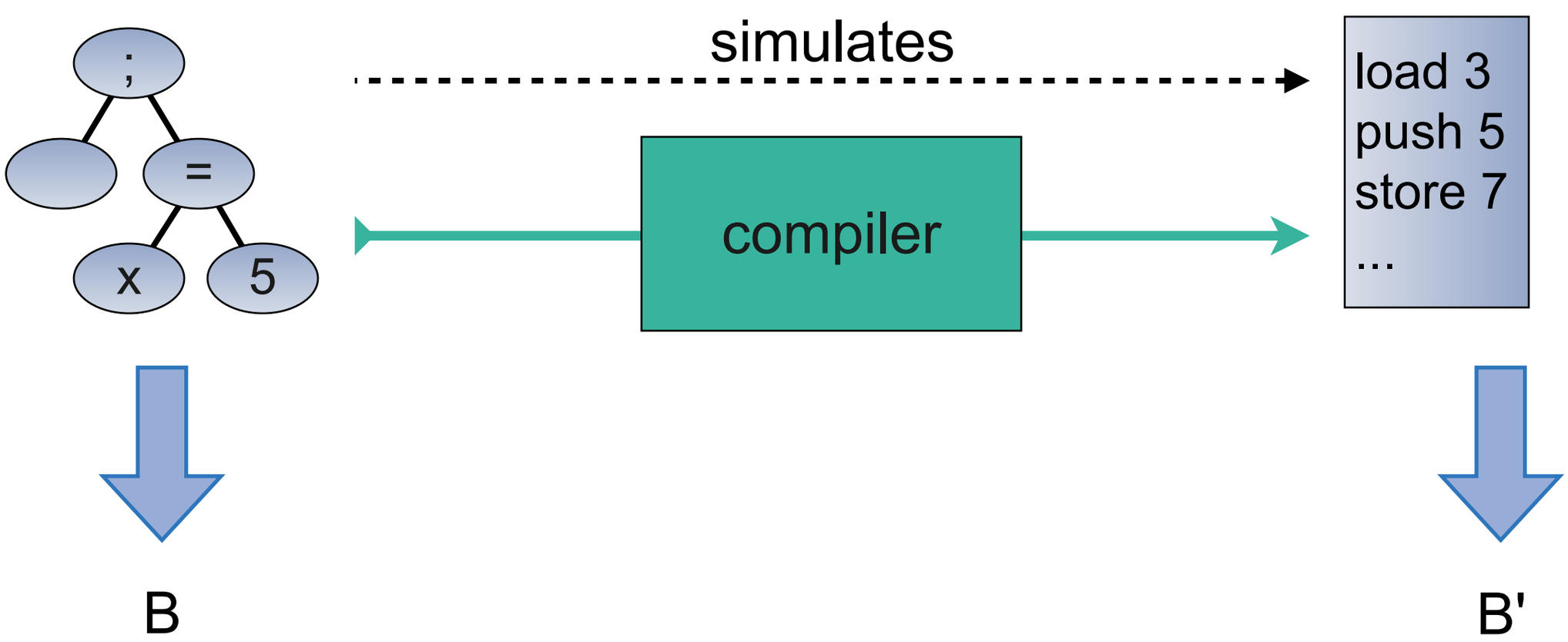
Why does it work for sequential languages?





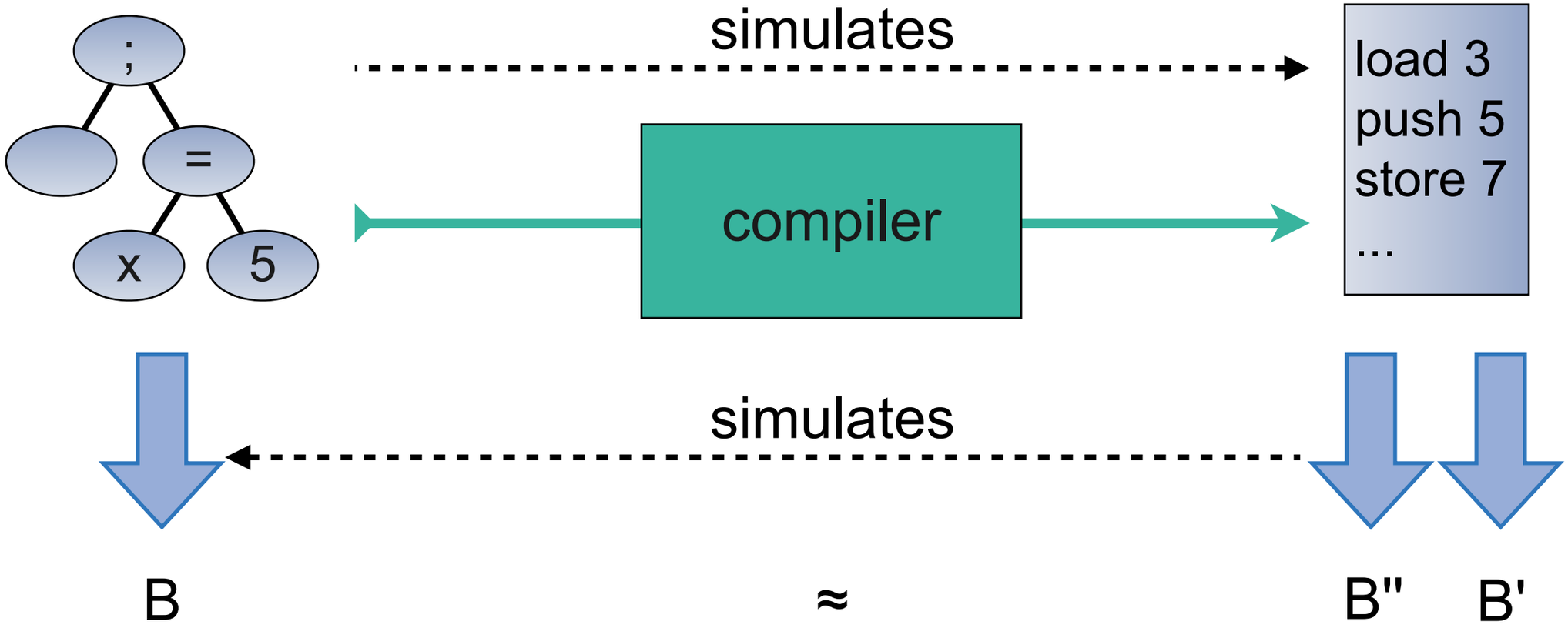
Why does it work for sequential languages?

determinism + source code = **bisimulation**



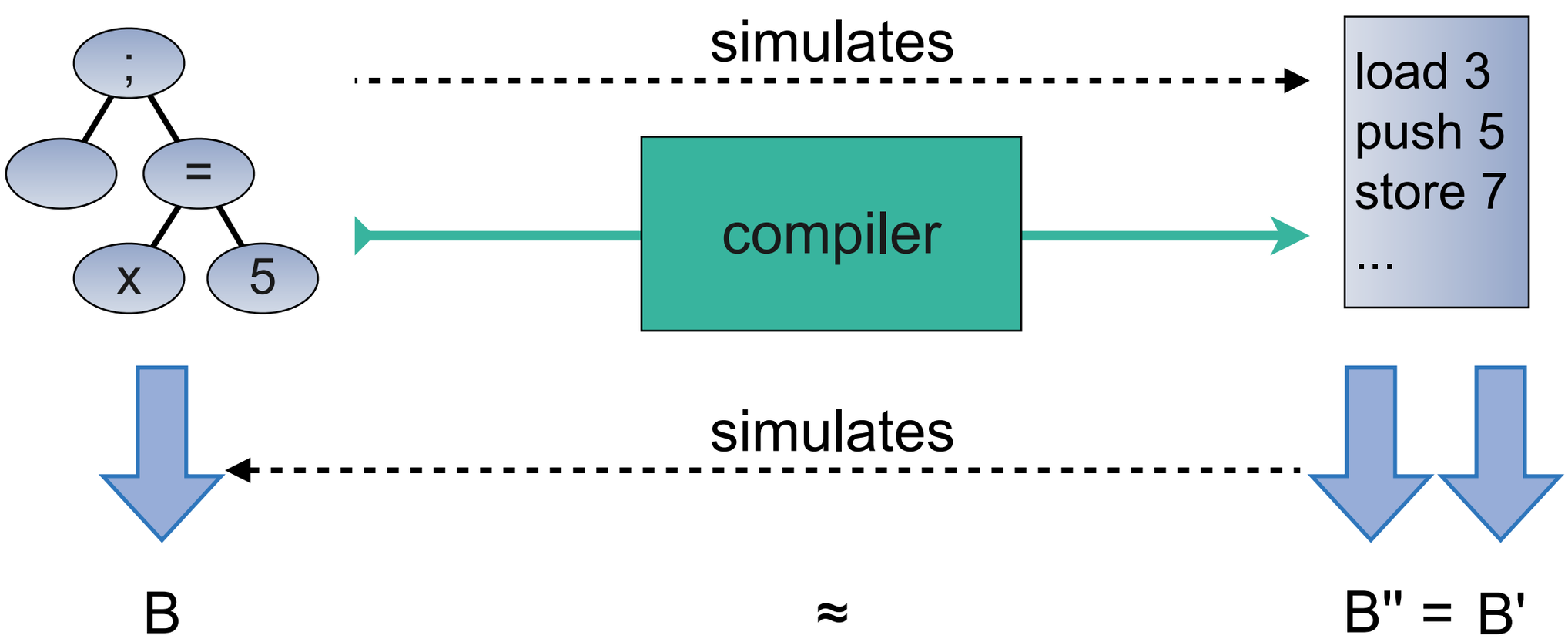
Why does it work for sequential languages?

determinism + source code = **bisimulation**

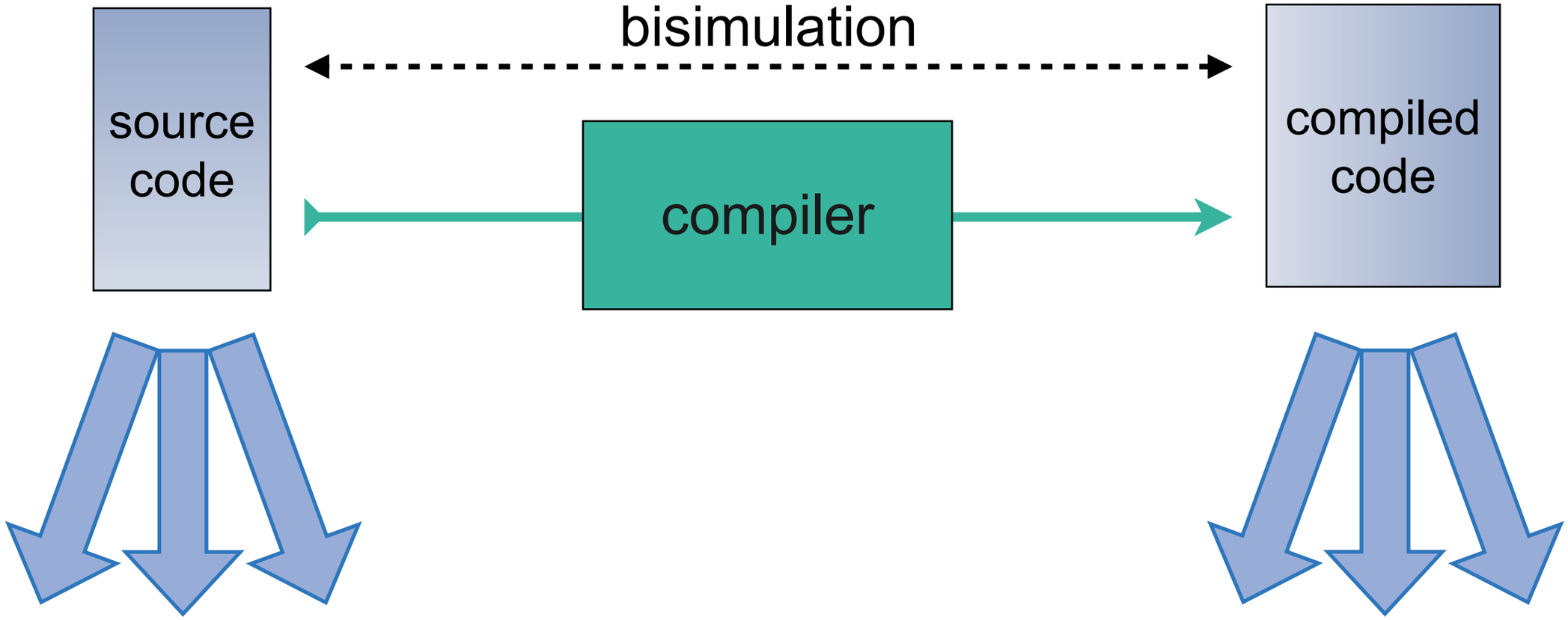


Why does it work for sequential languages?

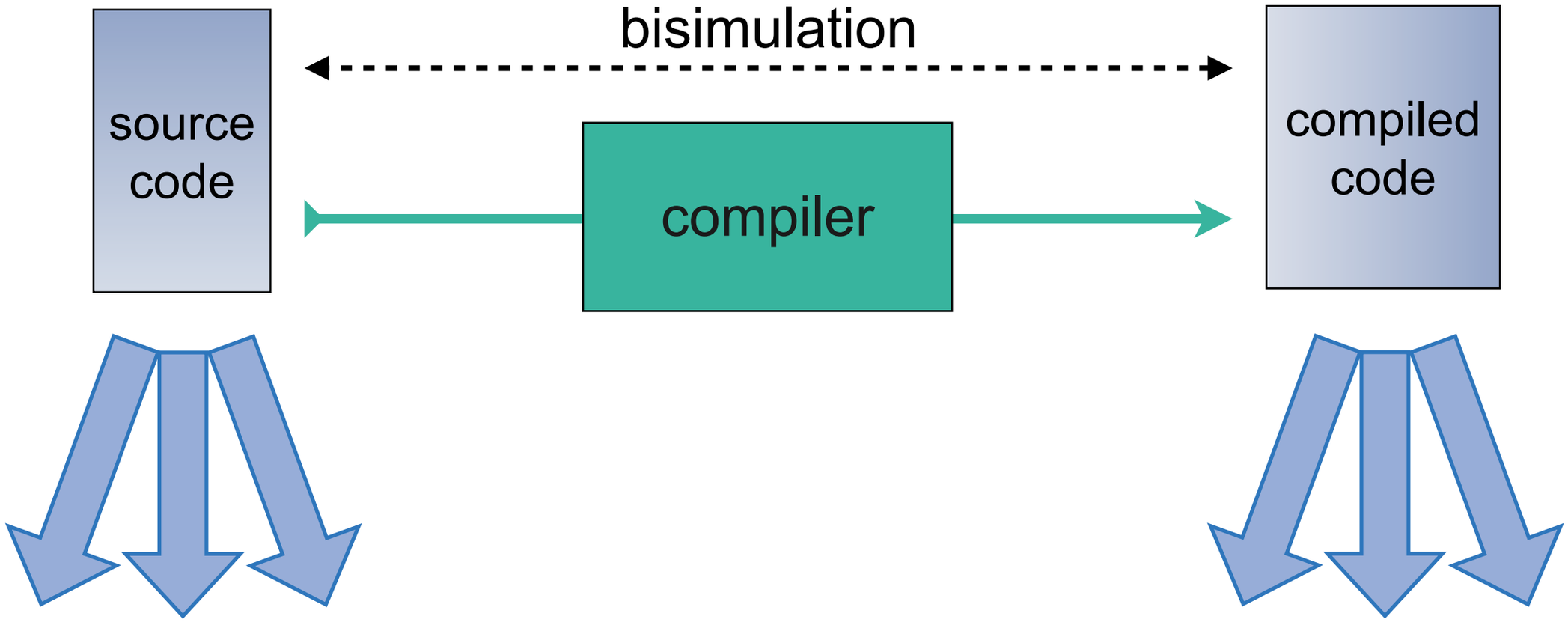
determinism + source code = **bisimulation**



Shared memory concurrency



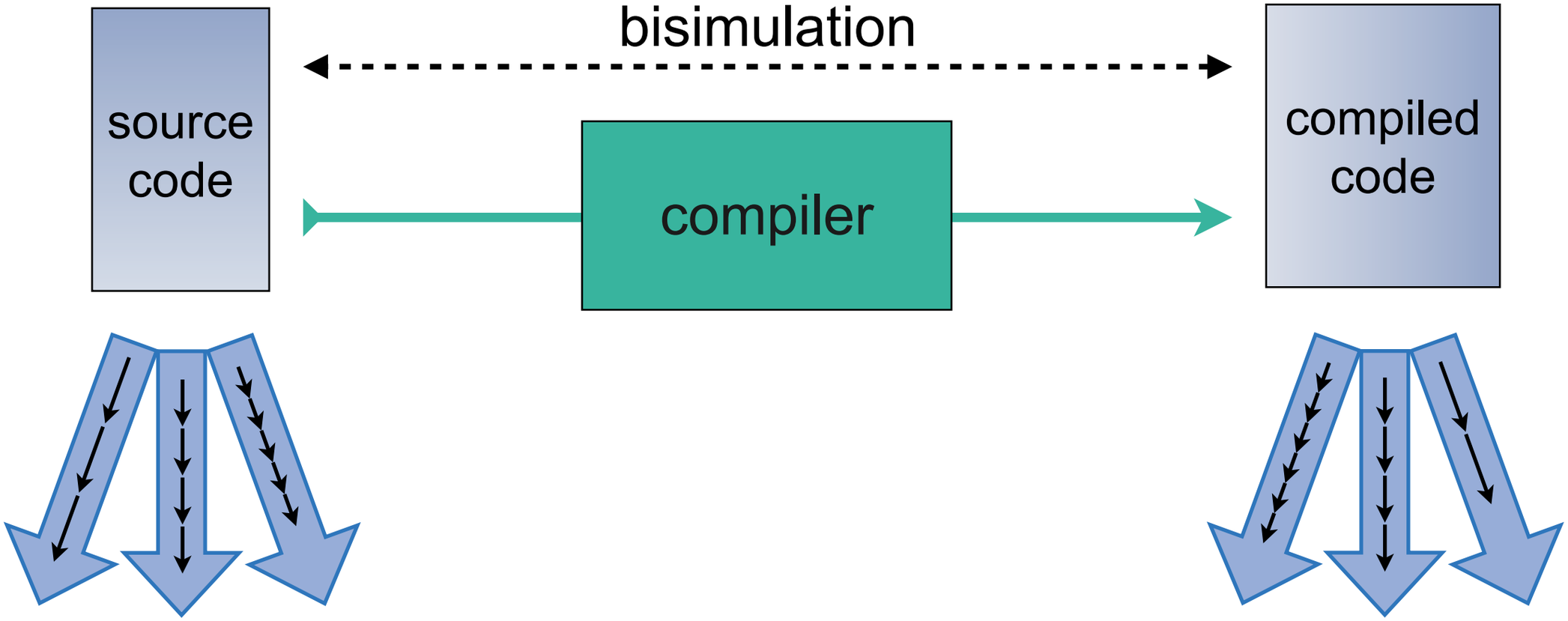
Shared memory concurrency



behaviour:

- result state / trace
- non-termination
- deadlock

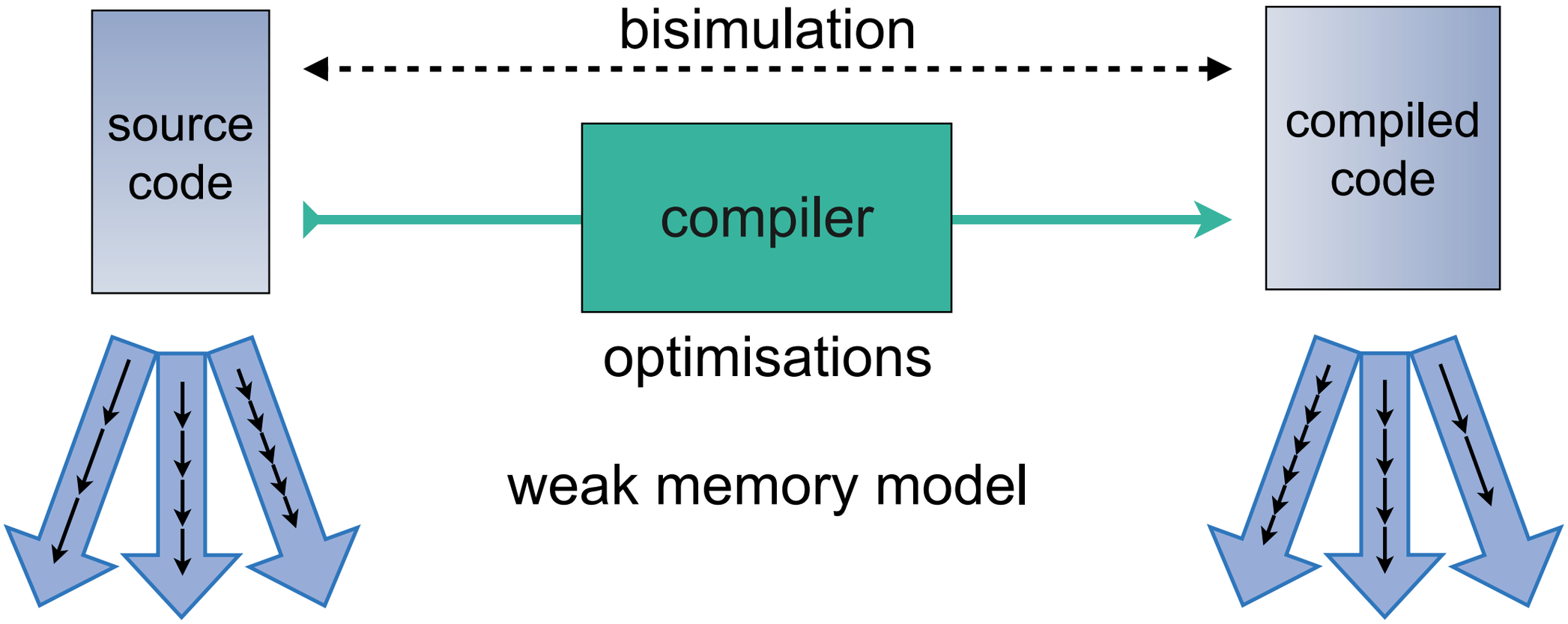
Shared memory concurrency



behaviour:

- result state / trace
- non-termination
- deadlock

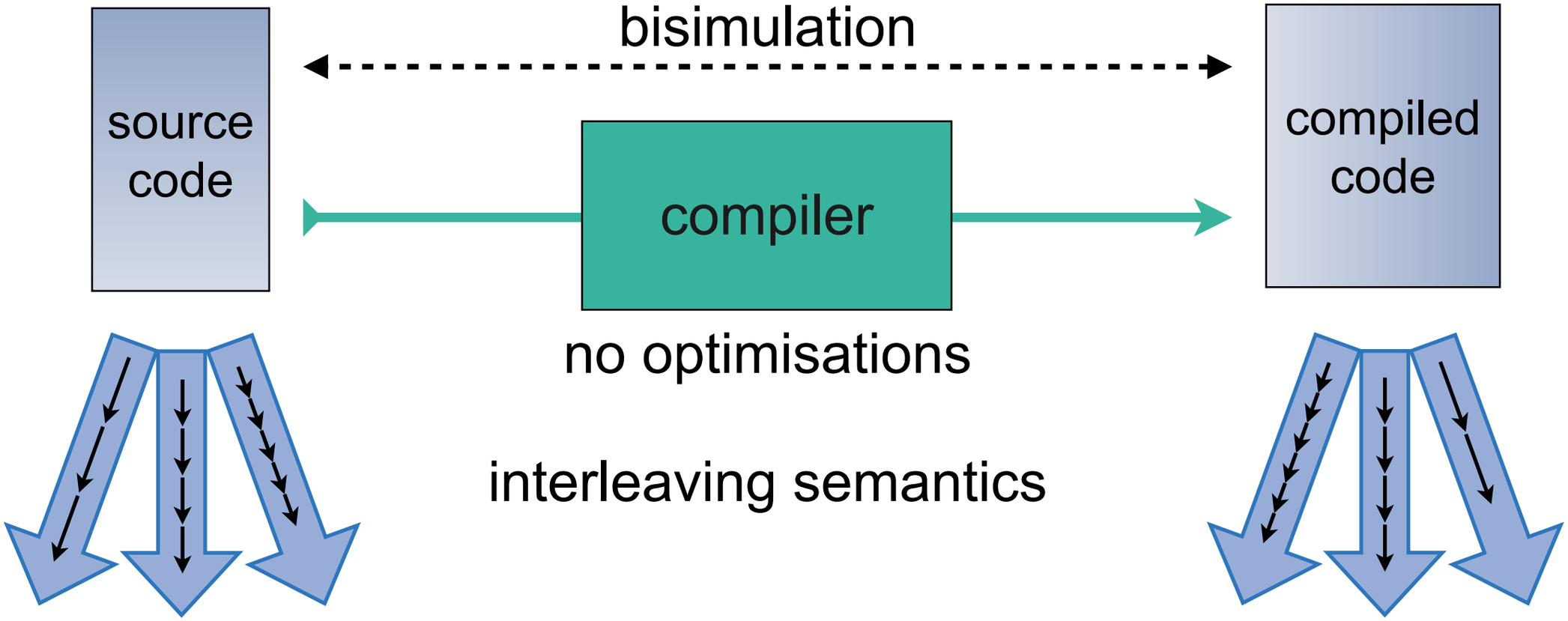
Shared memory concurrency



behaviour:

- result state / trace
- non-termination
- deadlock

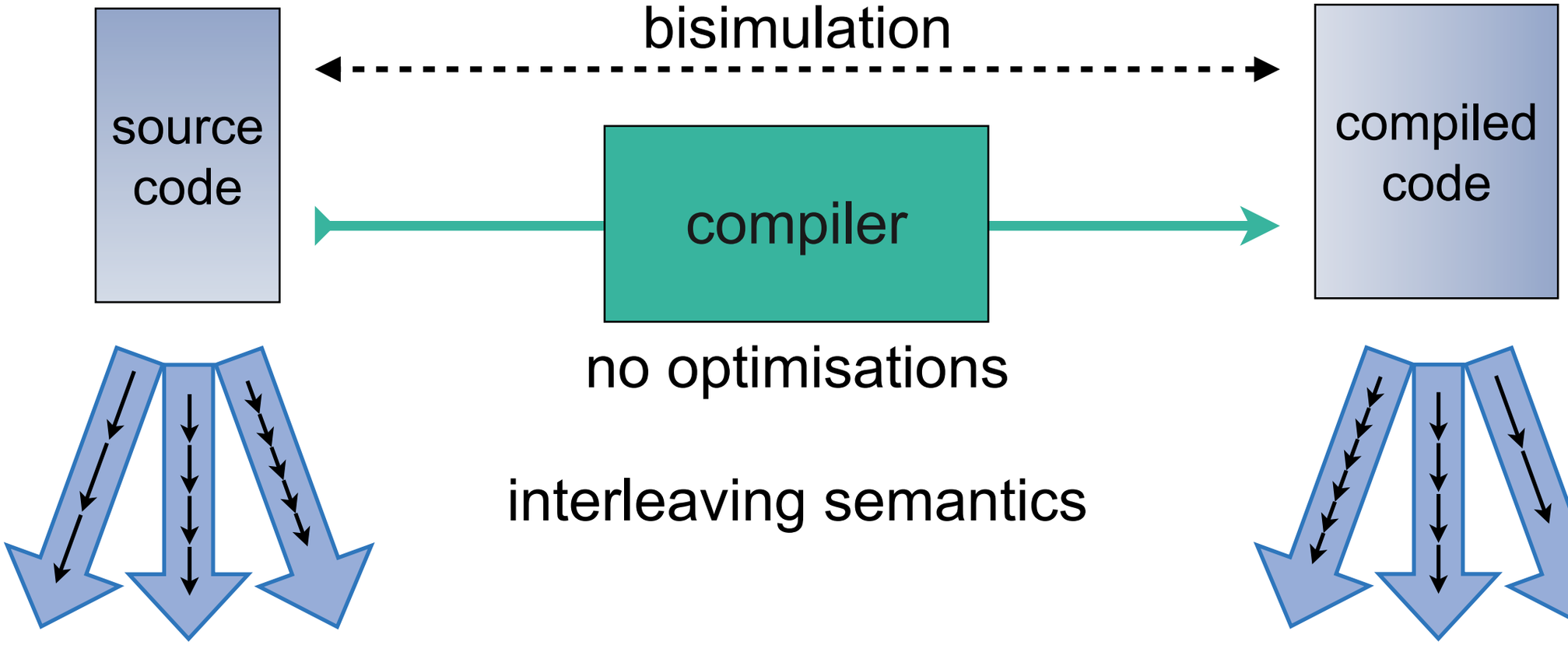
Shared memory concurrency



behaviour:

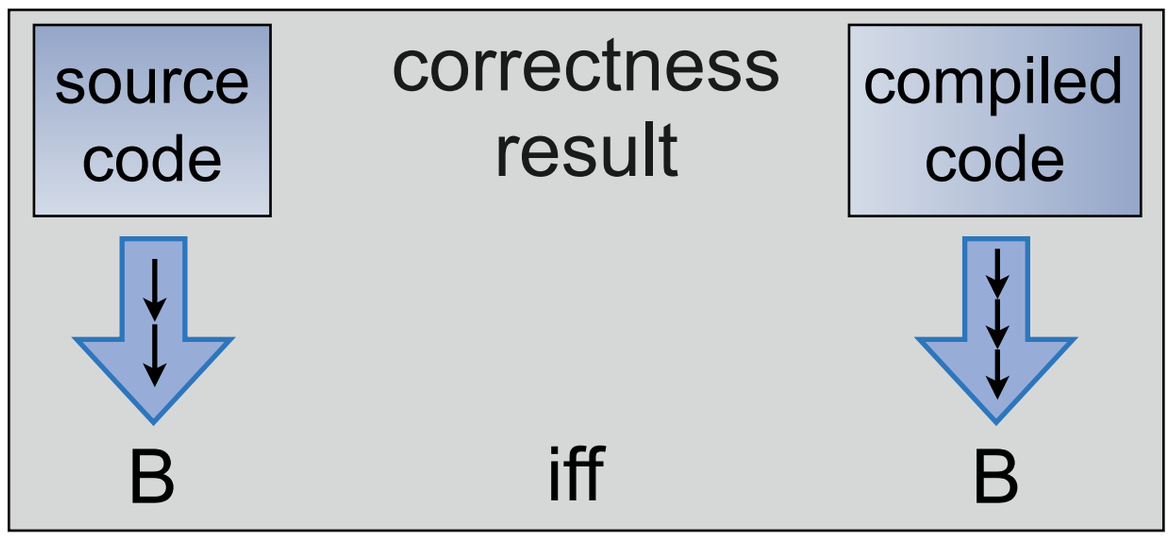
- result state / trace
- non-termination
- deadlock

Shared memory concurrency

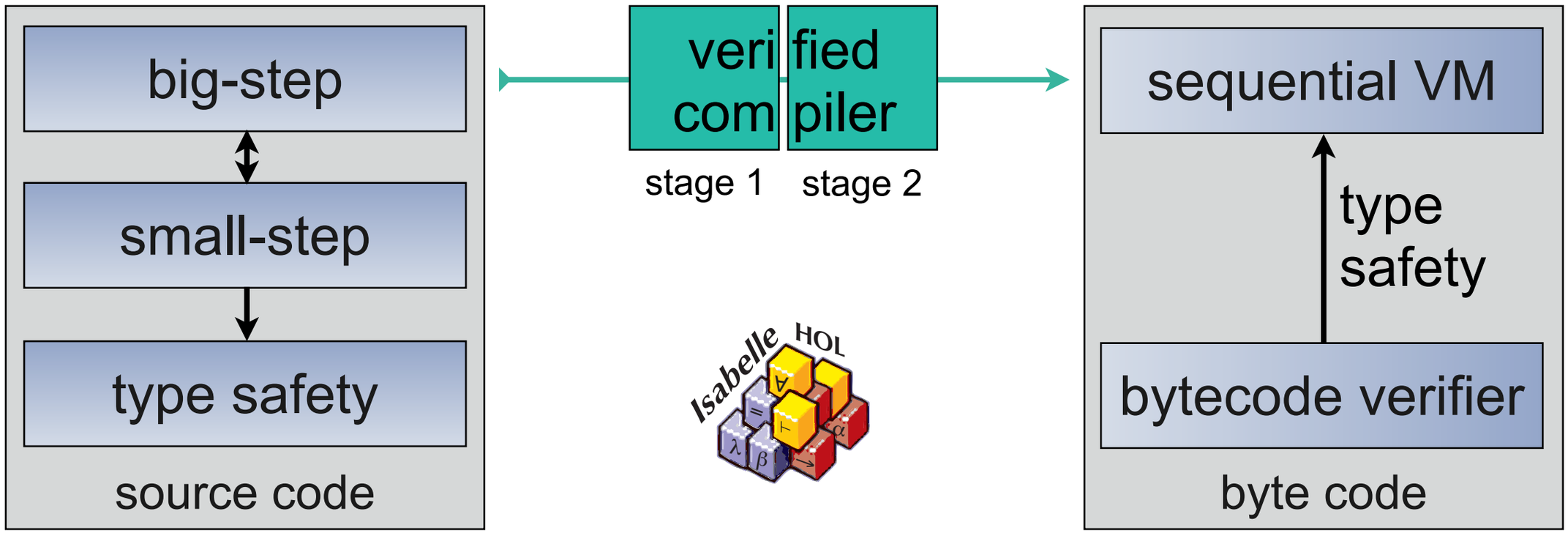


behaviour:

- result state / trace
- non-termination
- deadlock



Jinja [Klein, Nipkow '06]



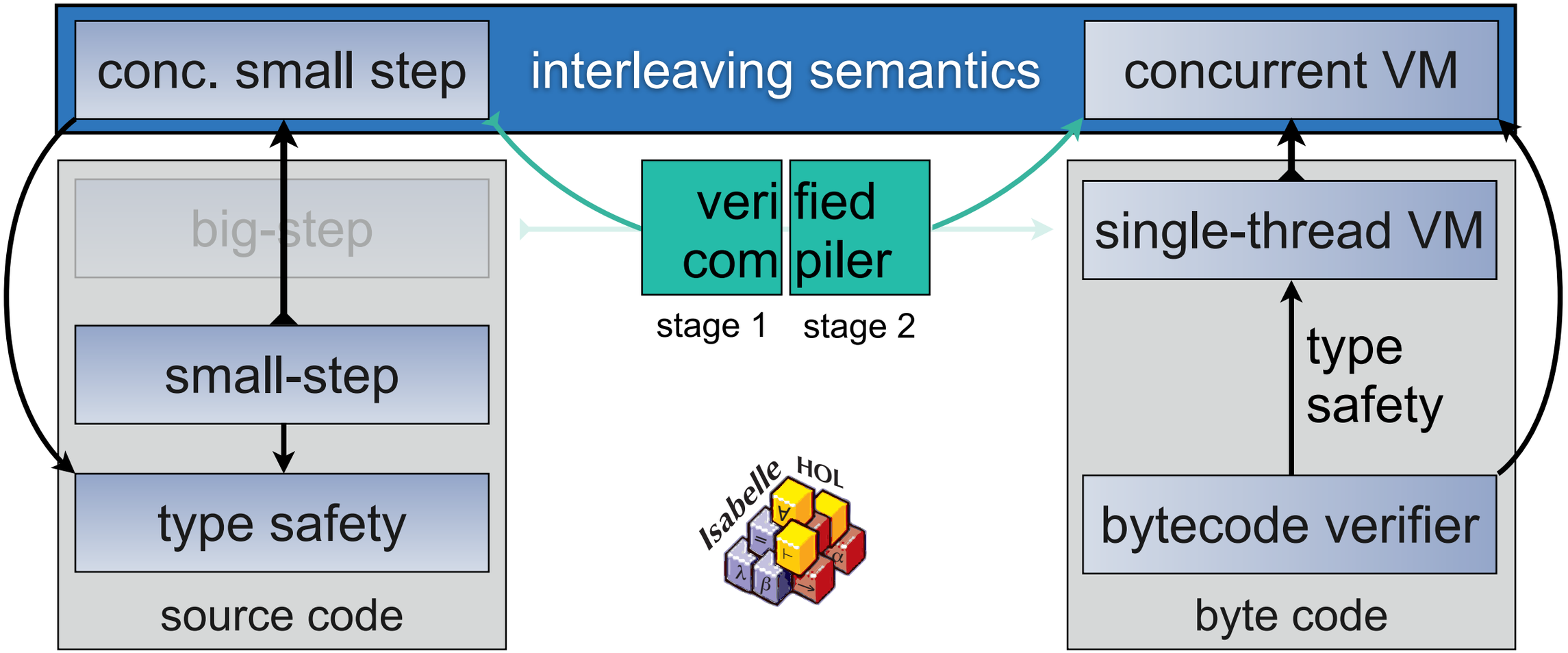
Java features:

- classes, objects & fields
- inheritance & late binding
- exceptions
- imperative features

not modelled:

- reflection & class loading
- interfaces
- threads

JinjaThreads



Java concurrency features:

- arbitrary thread creation
- synchronisation
- join
- wait / notify

not modelled:

- (thread interruption)
- java.util.concurrent

Interleaving small-step semantics

single-thread semantics

$t \vdash \langle x, h\rangle \xrightarrow{ta} \langle x', h'\rangle$

multithreaded semantics

$\langle\!\langle \sigma, h\rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h'\rangle\!\rangle$

Interleaving small-step semantics

single-thread semantics

$t \vdash \langle x, h\rangle \xrightarrow{ta} \langle x', h'\rangle$

multithreaded semantics

$\langle\!\langle \sigma, h\rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h'\rangle\!\rangle$

locks
thread-local states
wait sets

Interleaving small-step semantics

single-thread semantics

$t \vdash \langle x, h\rangle \xrightarrow{ta} \langle x', h'\rangle$

interleaving

multithreaded semantics

$\langle\!\langle \sigma, h\rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h'\rangle\!\rangle$

new thread x
lock l / unlock l
wait w / notify w / ...

locks
thread-local states
wait sets

Interleaving small-step semantics

single-thread semantics

$t \vdash \langle x, h\rangle \xrightarrow{ta} \langle x', h'\rangle$

interleaving

multithreaded semantics

$\langle\!\langle \sigma, h\rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h'\rangle\!\rangle$

new thread x
lock l / unlock l
wait w / notify w / ...

locks
thread-local states
wait sets

$$\frac{h\ a = \text{Obj}\ C\ fs \qquad P \vdash C \leq \text{Thread} \qquad P \vdash C \text{ sees run}() = body}{t \vdash \langle (\text{addr}\ a).\text{start}(), h\rangle \xrightarrow{\text{[NewThread body]}} \langle \text{Unit}, h\rangle}$$

Interleaving small-step semantics

single-thread semantics

$t \vdash \langle x, h\rangle \xrightarrow{ta} \langle x', h'\rangle$

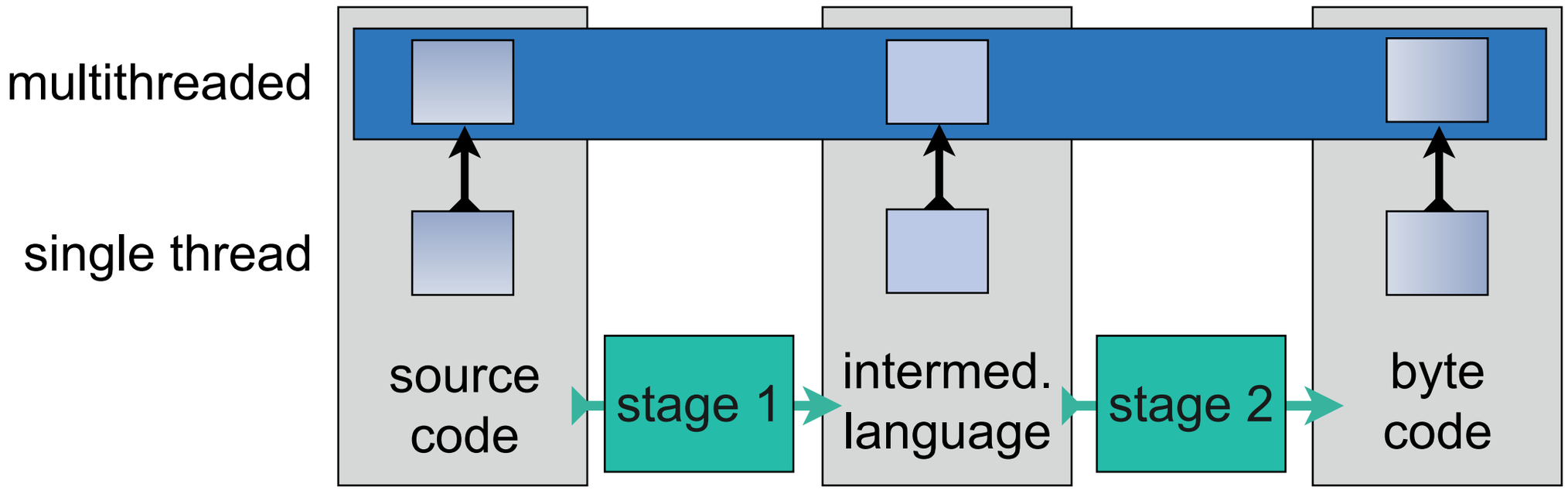
new thread x
lock l / unlock l
wait w / notify w / ...

interleaving

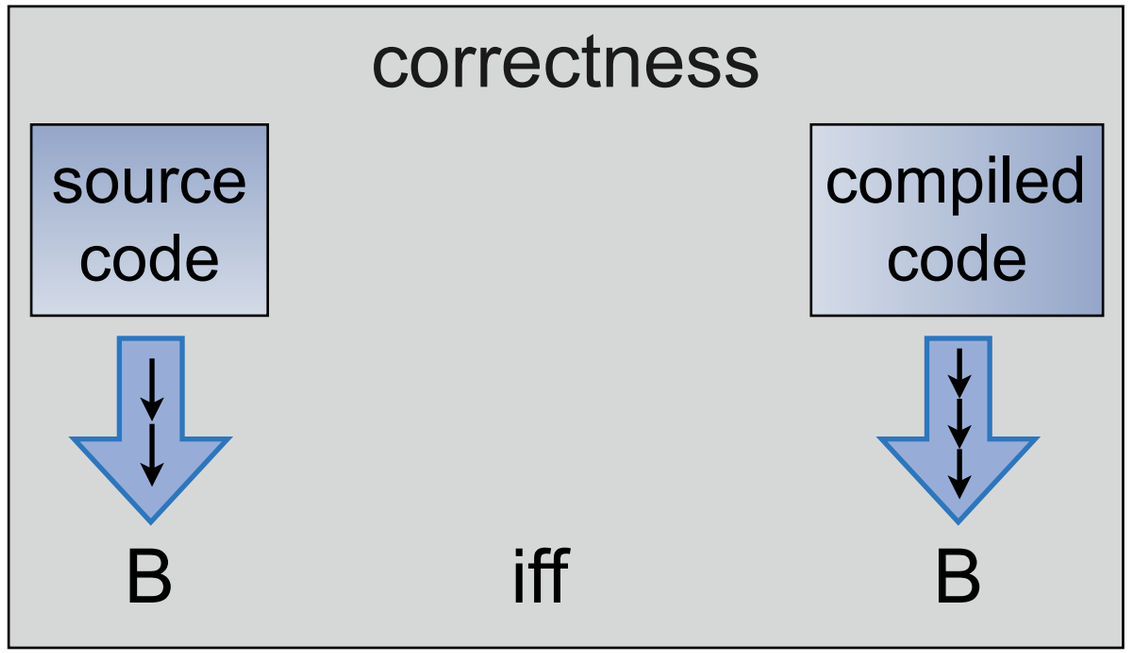
multithreaded semantics

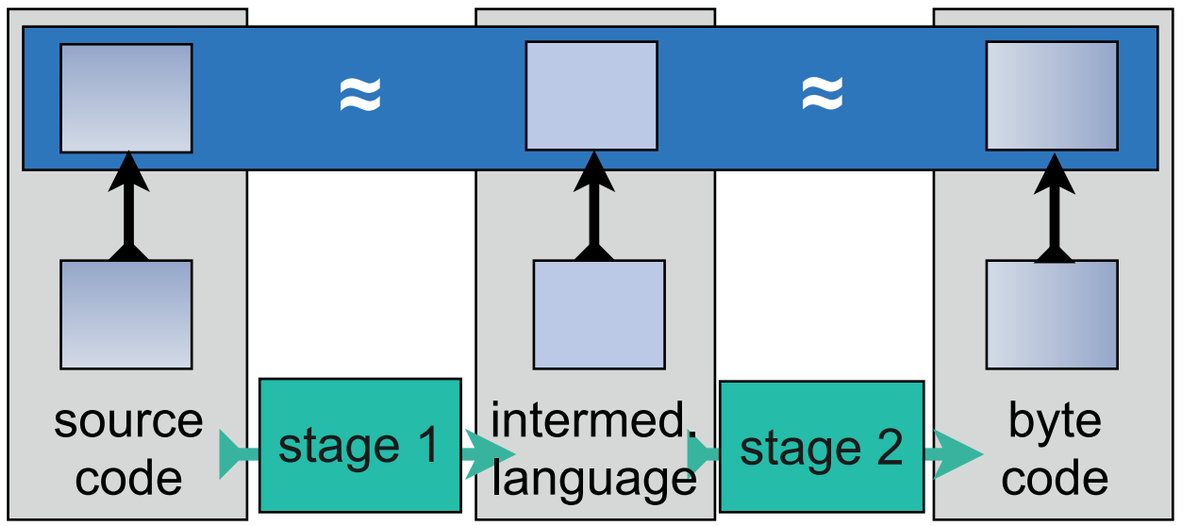
$\langle\!\langle \sigma, h\rangle\!\rangle \xrightarrow[ta]{t} \langle\!\langle \sigma', h'\rangle\!\rangle$

locks
thread-local states
wait sets

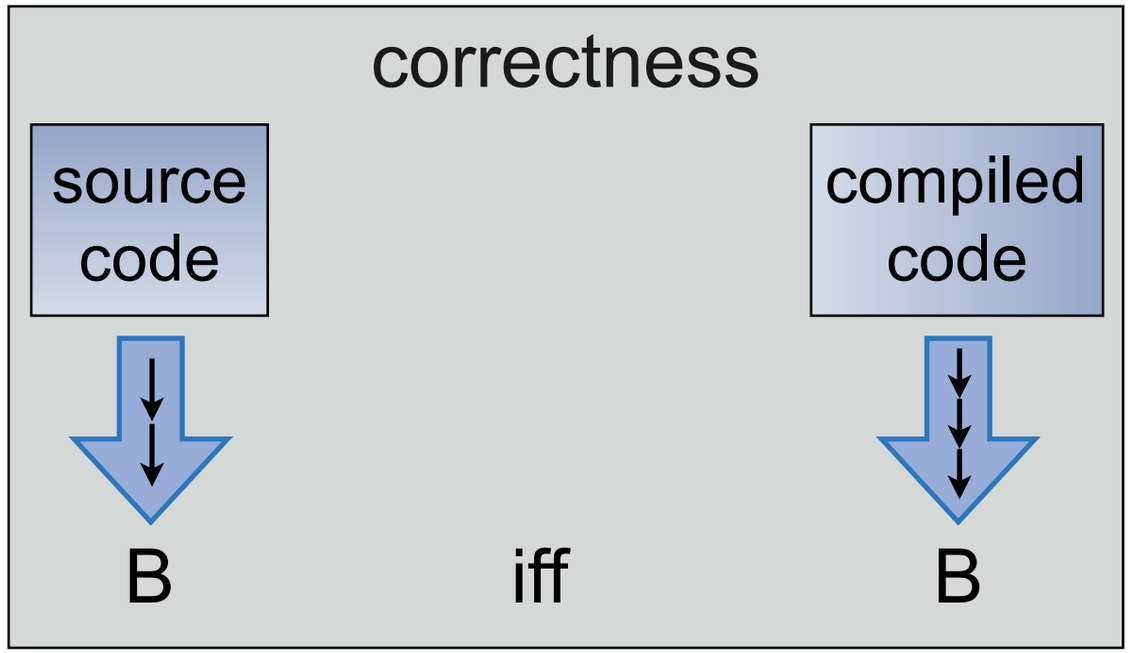


The correctness proof

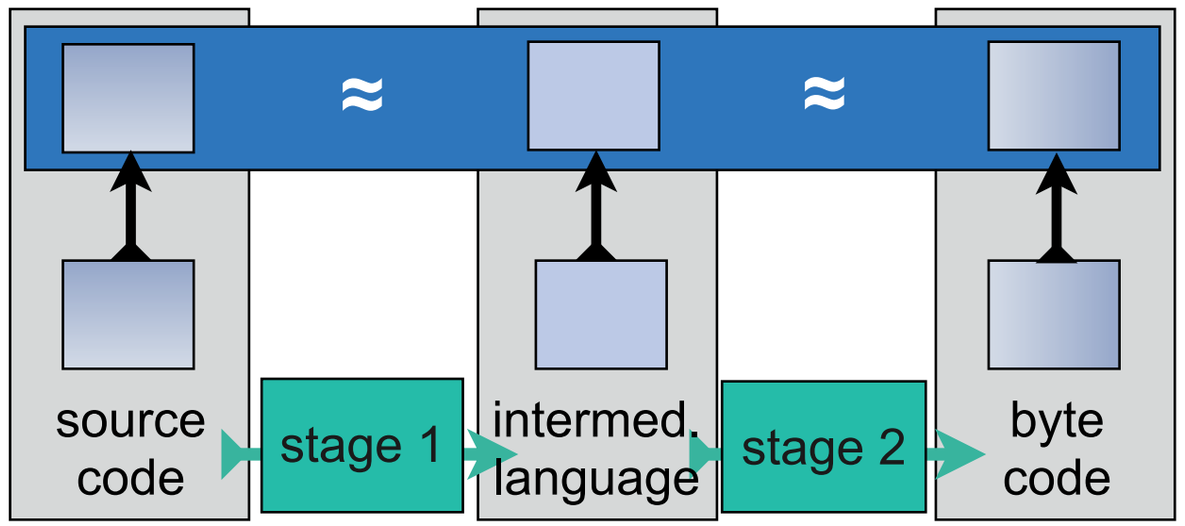




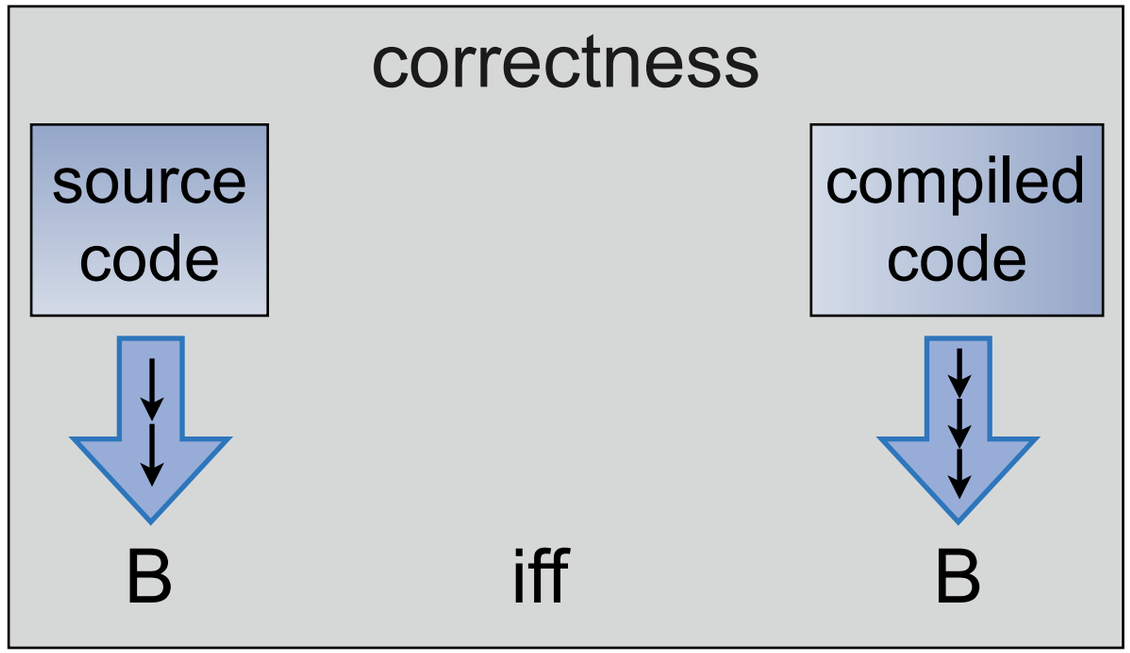
The correctness proof



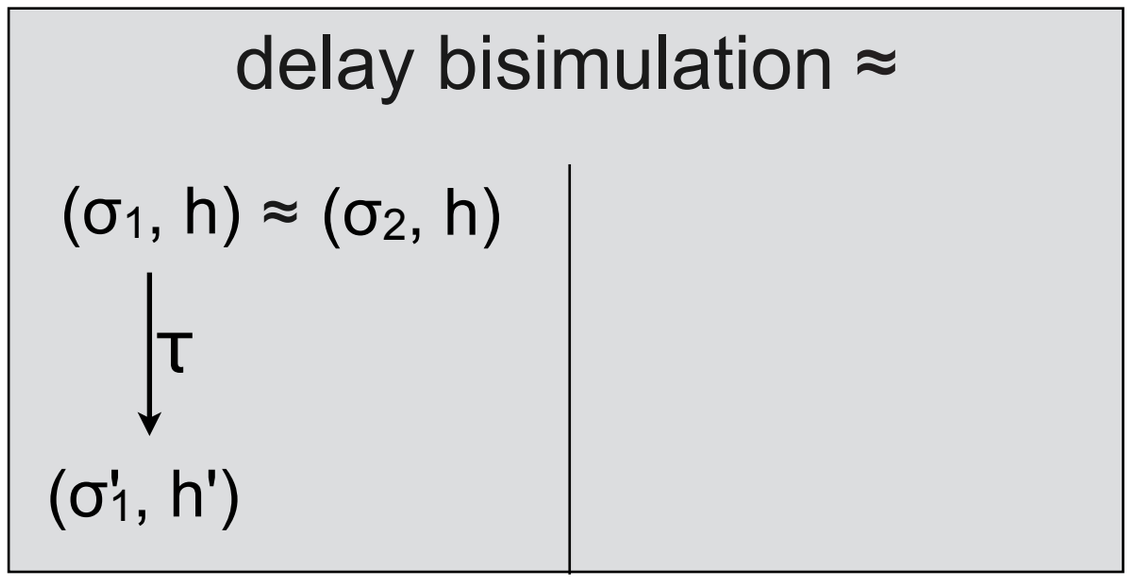
⇑

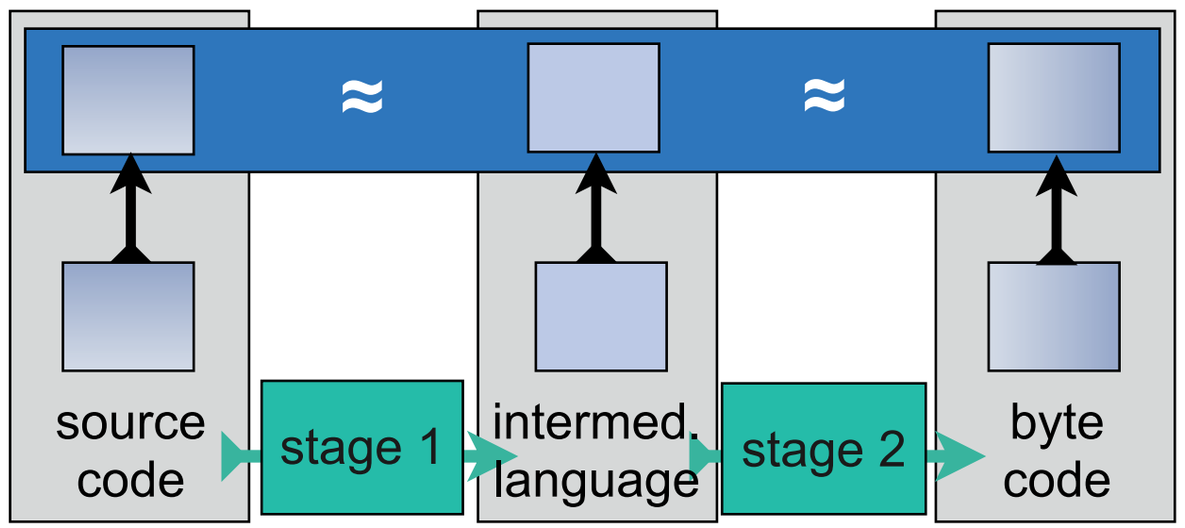


The correctness proof

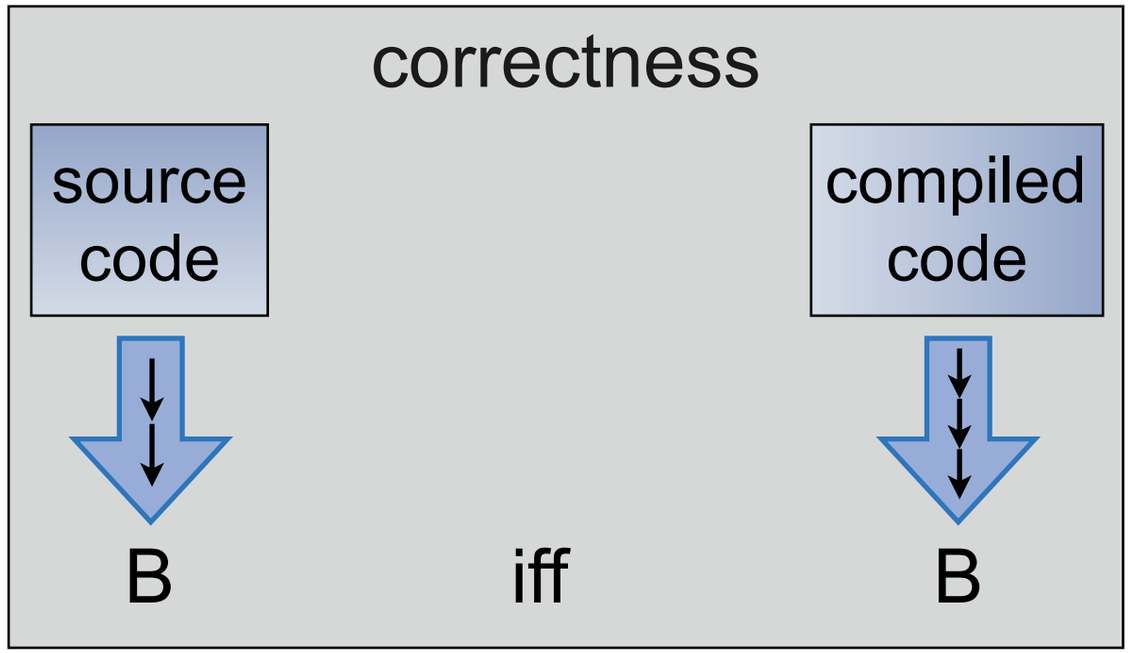


⇑

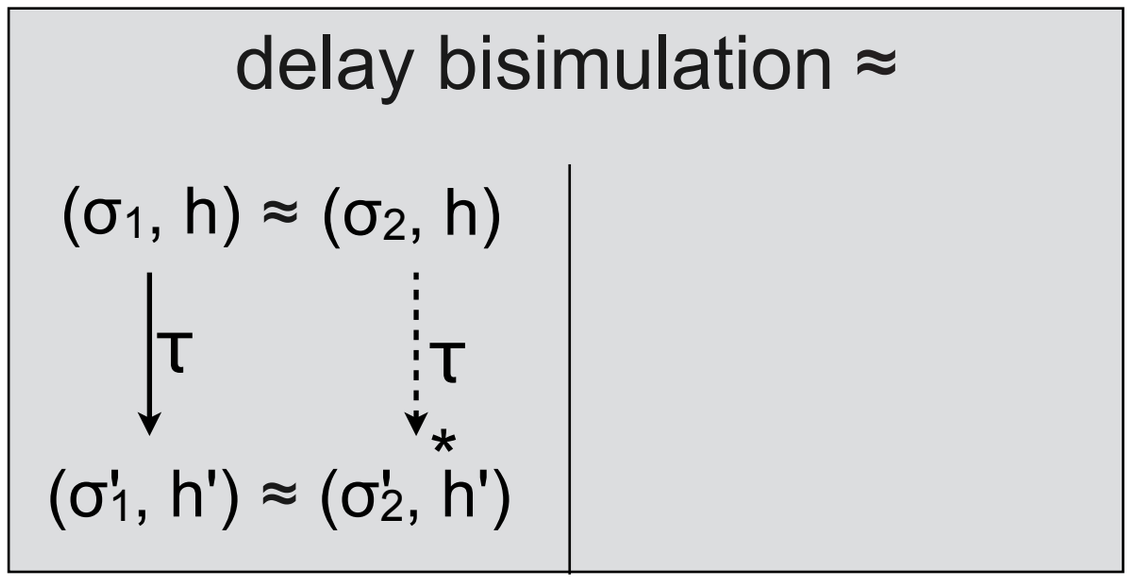


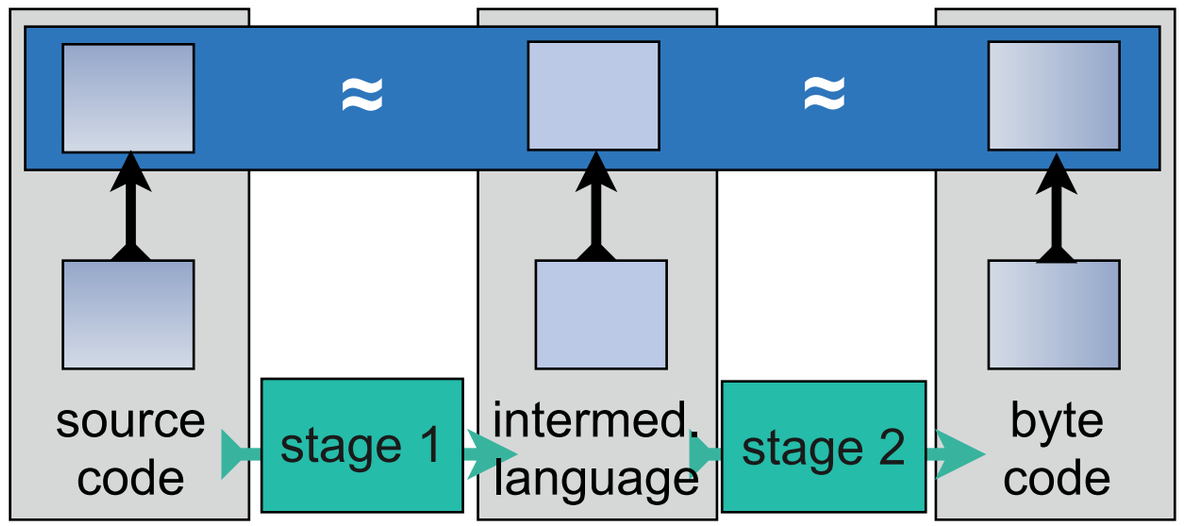


The correctness proof

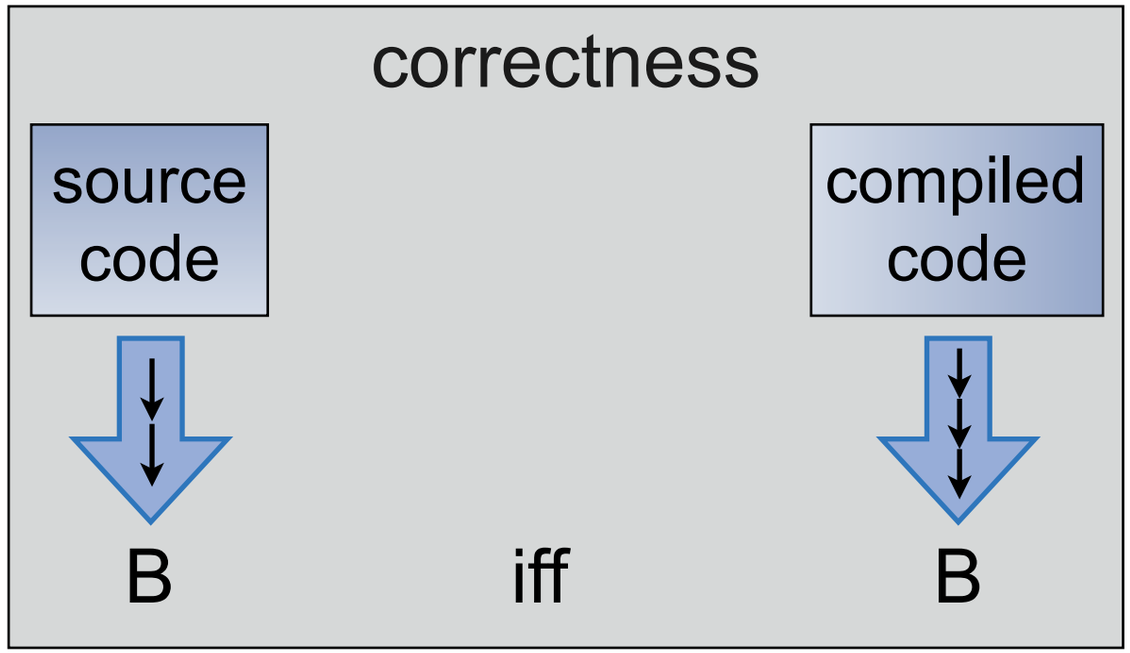


⇑

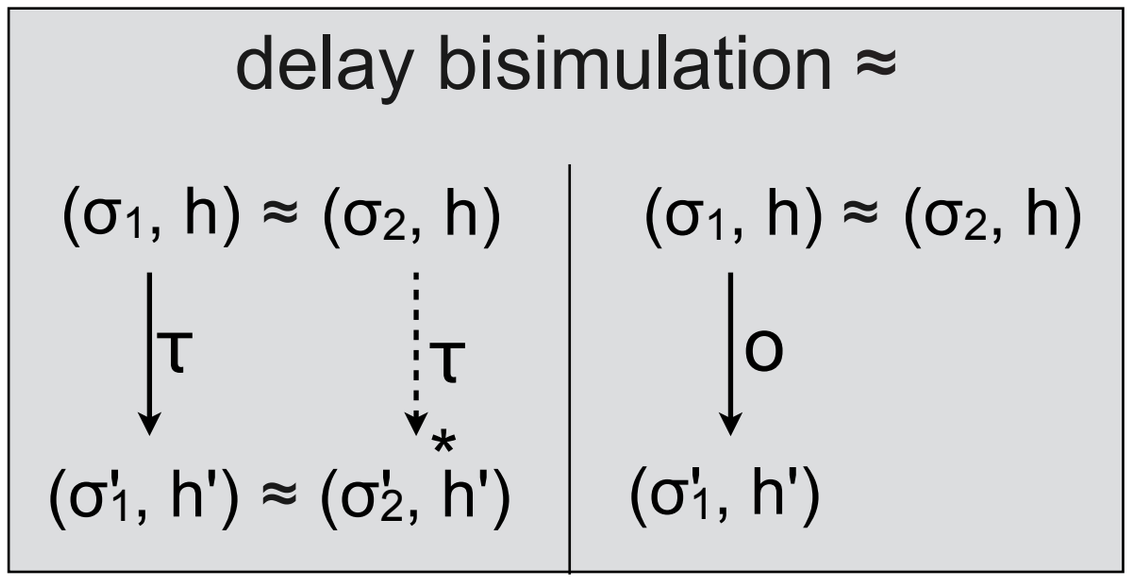


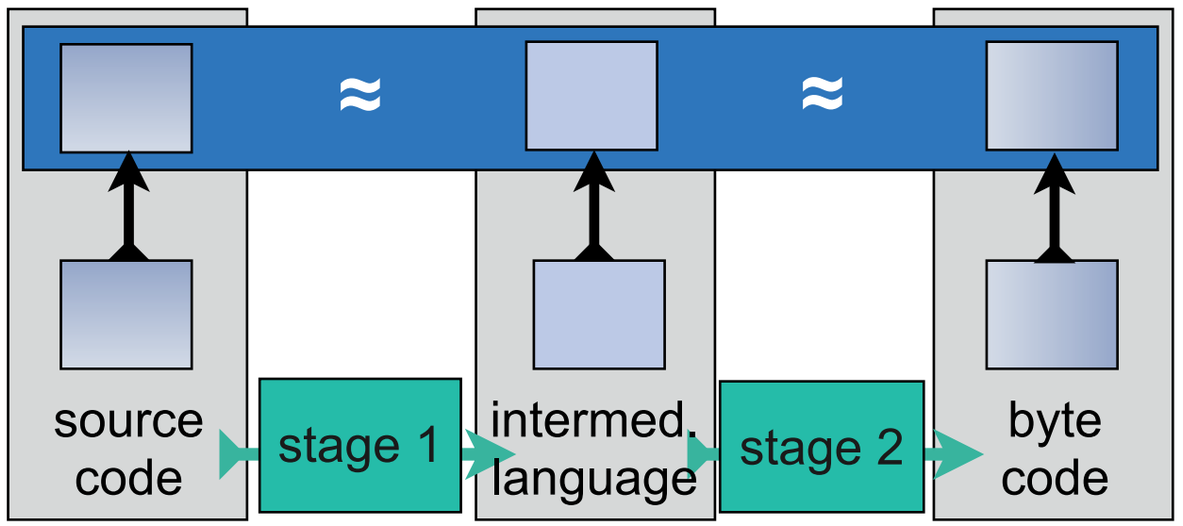


The correctness proof

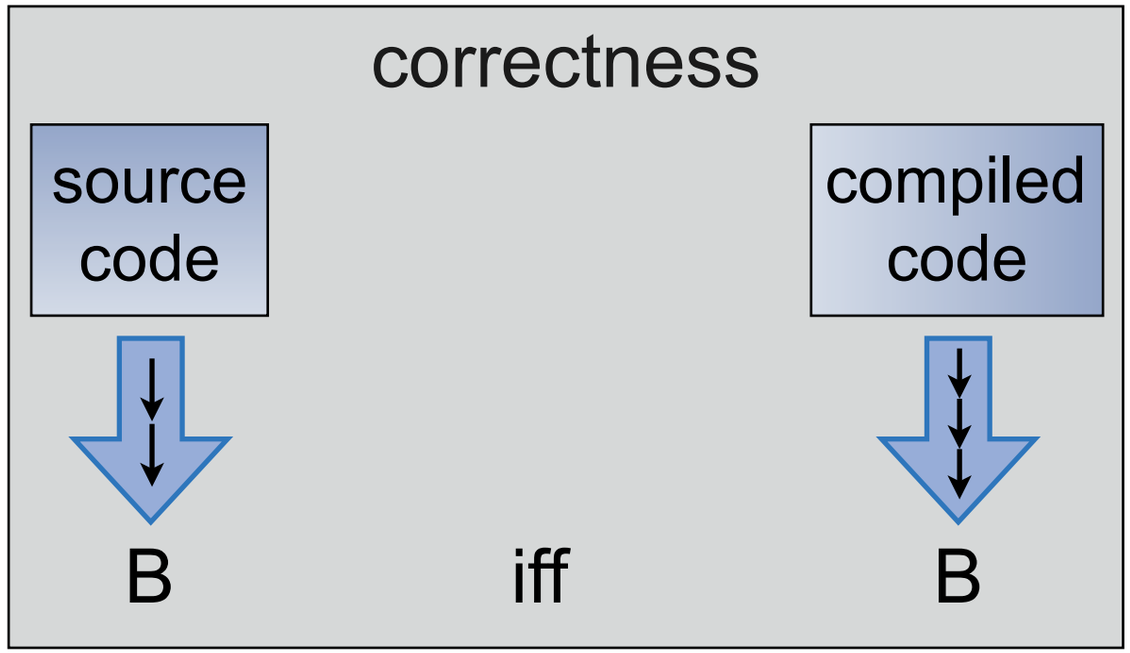


⇑

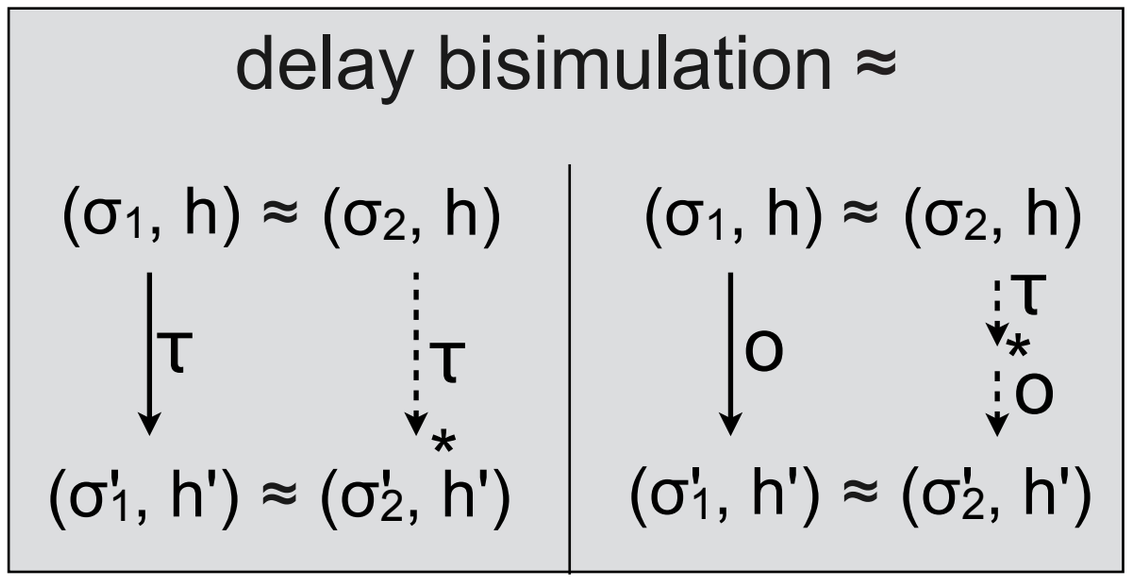


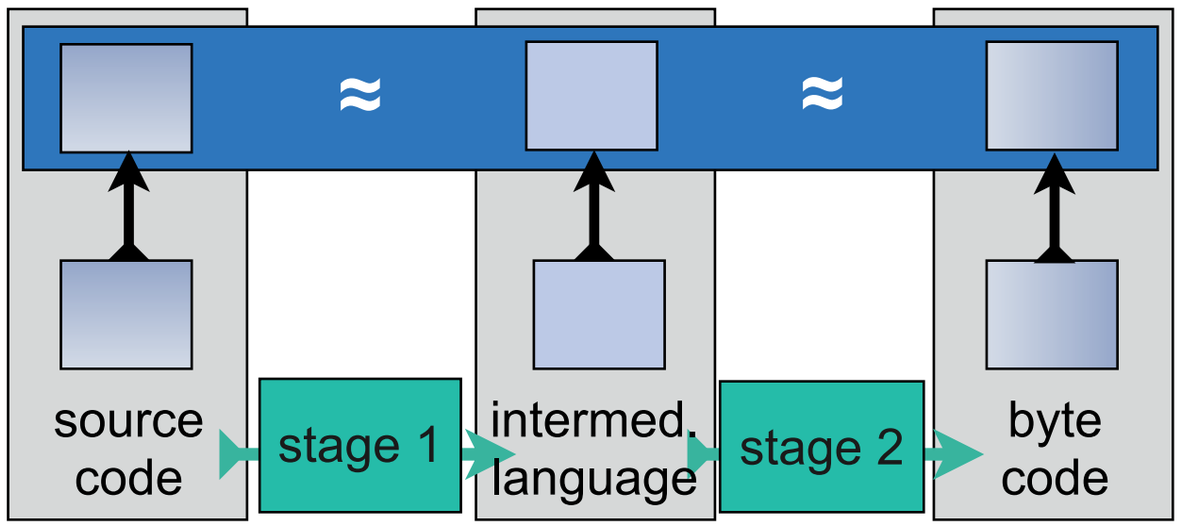


The correctness proof

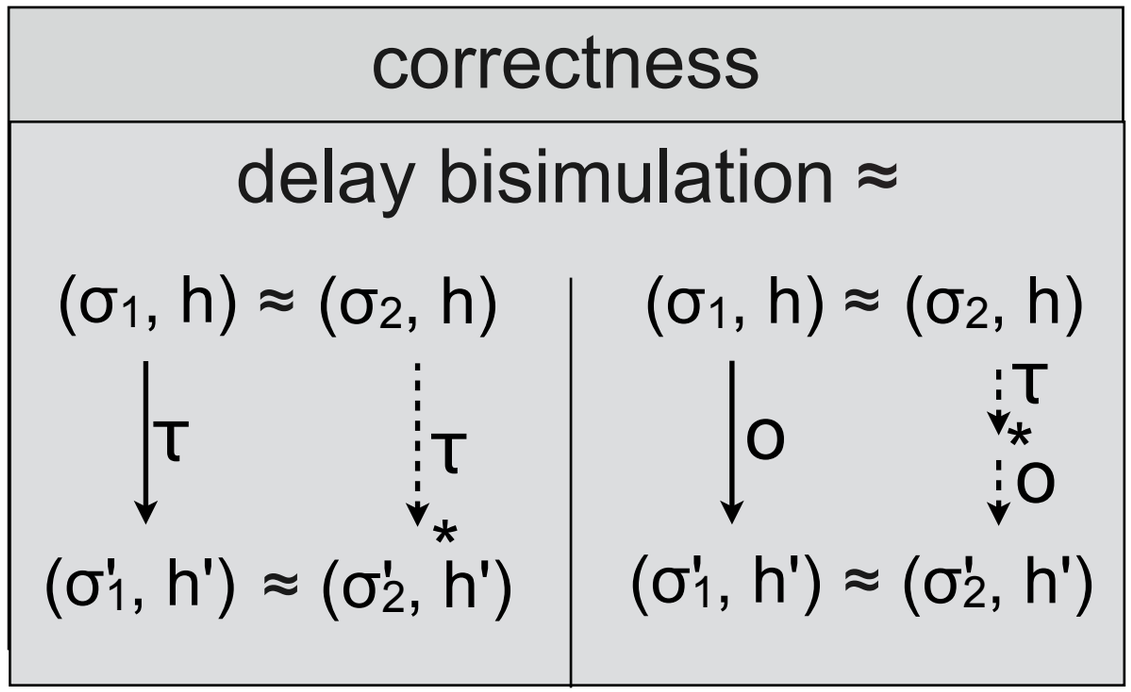


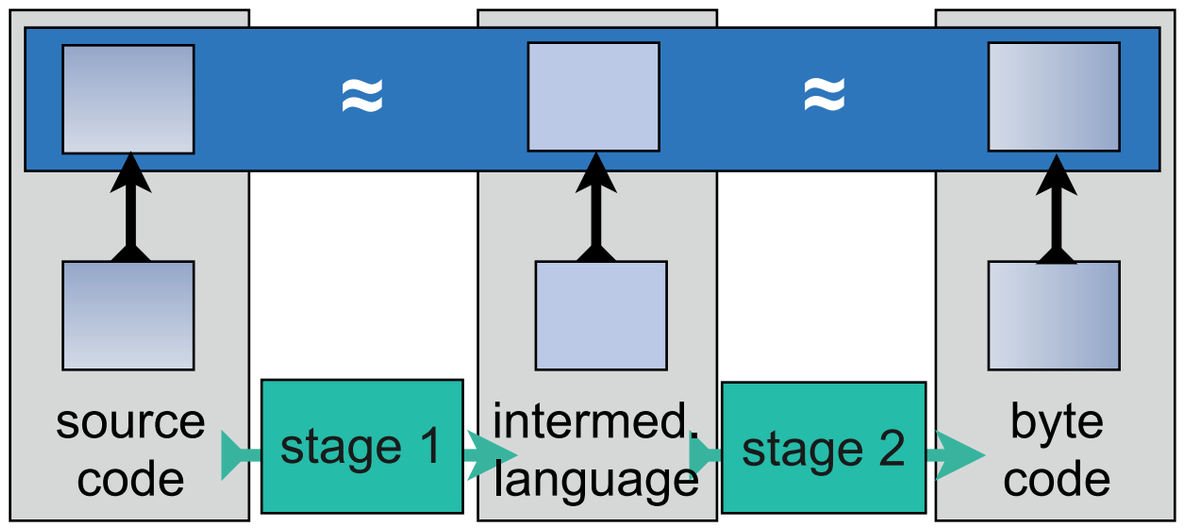
⇑



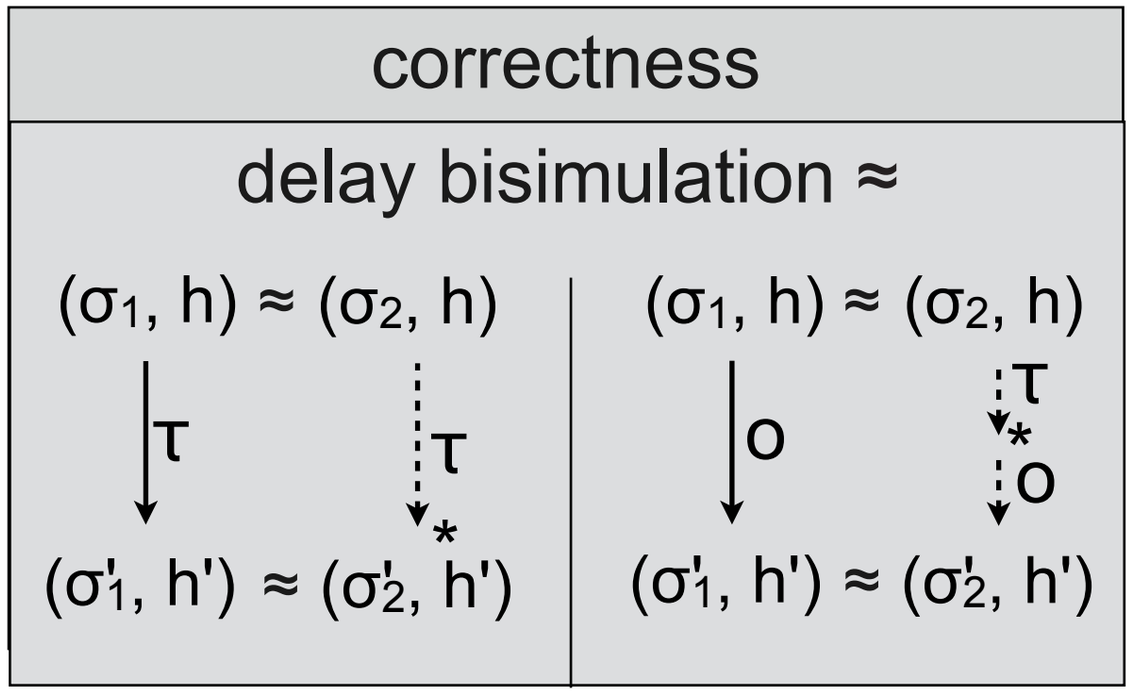


The correctness proof



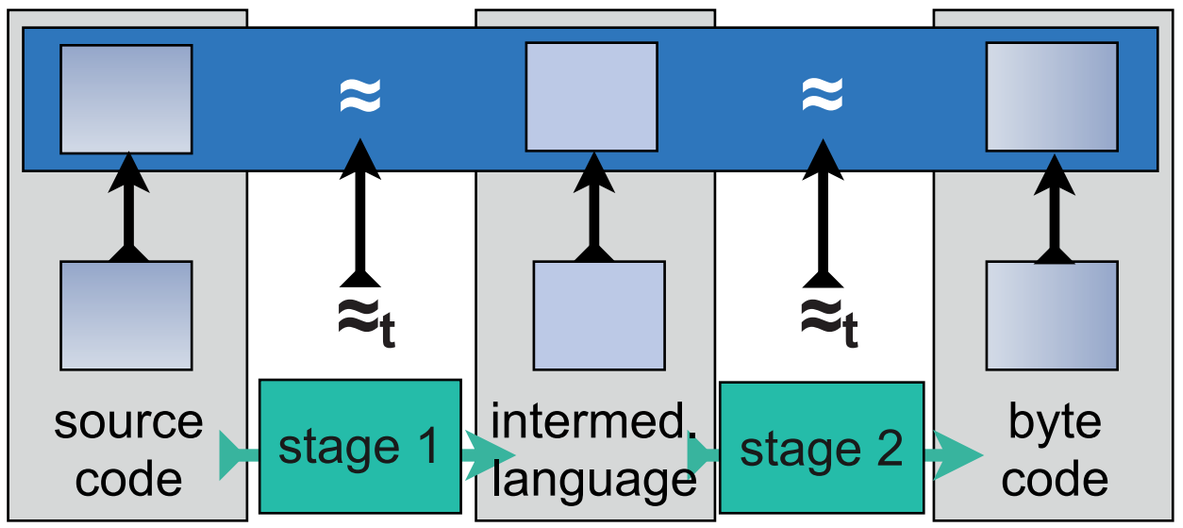


The correctness proof

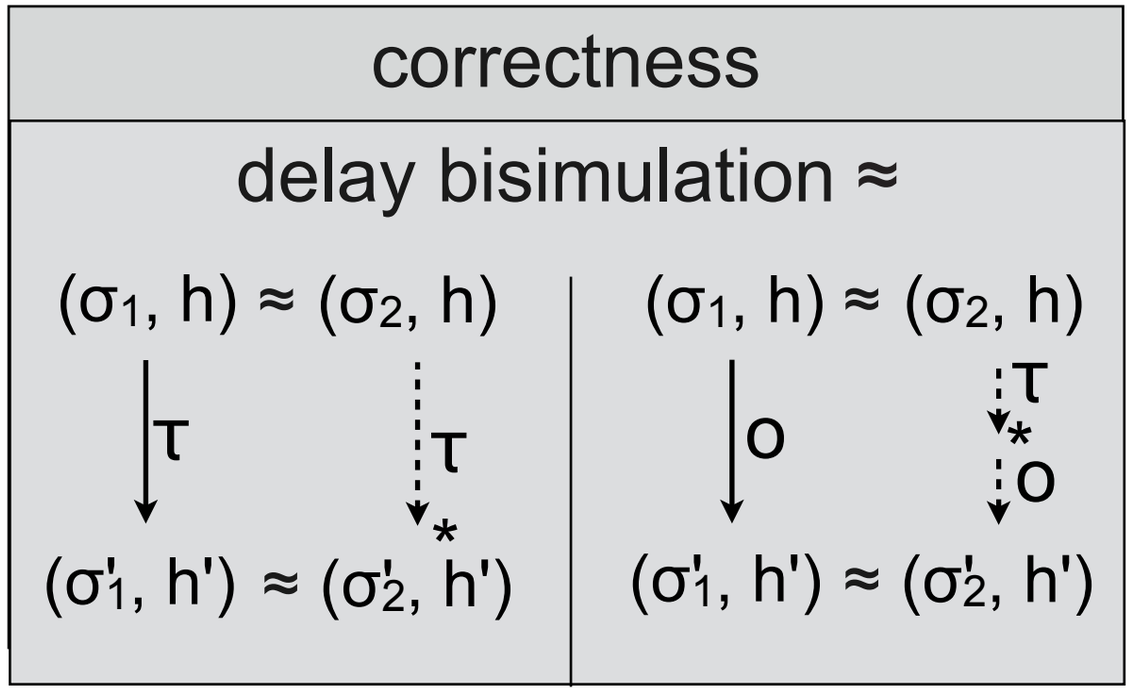


Observable steps

- heap access
- synchronisation
- thread creation
- method calls



The correctness proof



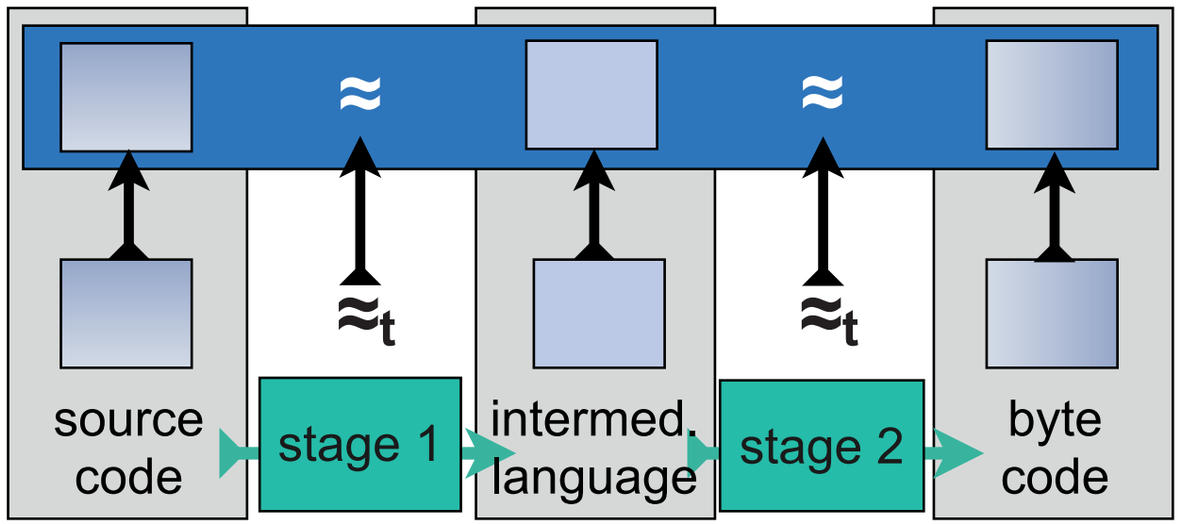
define $(\sigma_1, h) \approx (\sigma_2, h)$:

- locks and wait sets of σ_1 and σ_2 are the same
- thread-local states x_1 and x_2 satisfy:

$$(x_1, h) \approx_t (x_2, h)$$

Observable steps

- heap access
- synchronisation
- thread creation
- method calls



The correctness proof

correctness

delay bisimulation ≈

$(\sigma_1, h) \approx (\sigma_2, h)$ | $(\sigma_1, h) \approx (\sigma_2, h)$

τ ⋮ τ | o ⋮ τ^*

(σ'_1, h')

define $(\sigma_1, h) \approx (\sigma_2, h)$:

- locks and wait sets of σ_1 and σ_2 are the same
- thread-local states x_1 and x_2 satisfy:

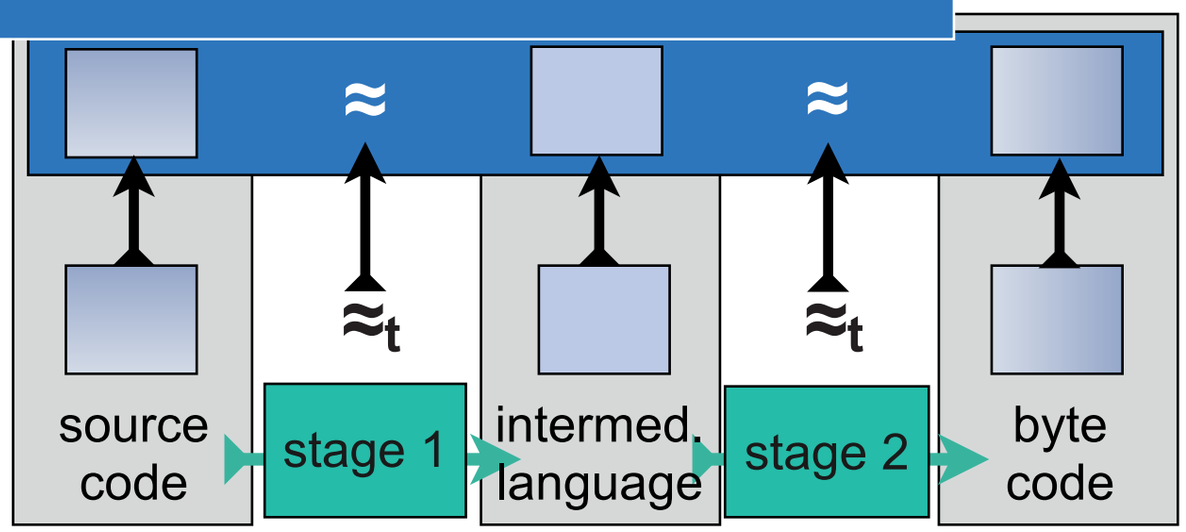
$$(x_1, h) \approx_t (x_2, h)$$

Theorem

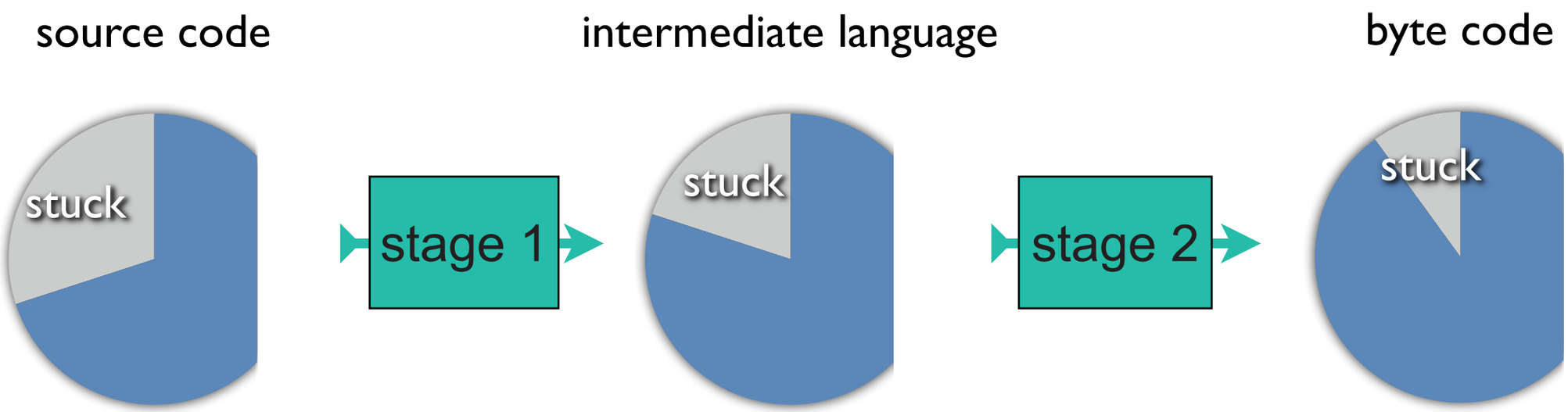
If $\approx_t$ is a single-thread delay bisimulation,
then $\approx$ is a multithreaded delay bisimulation.

Observable steps

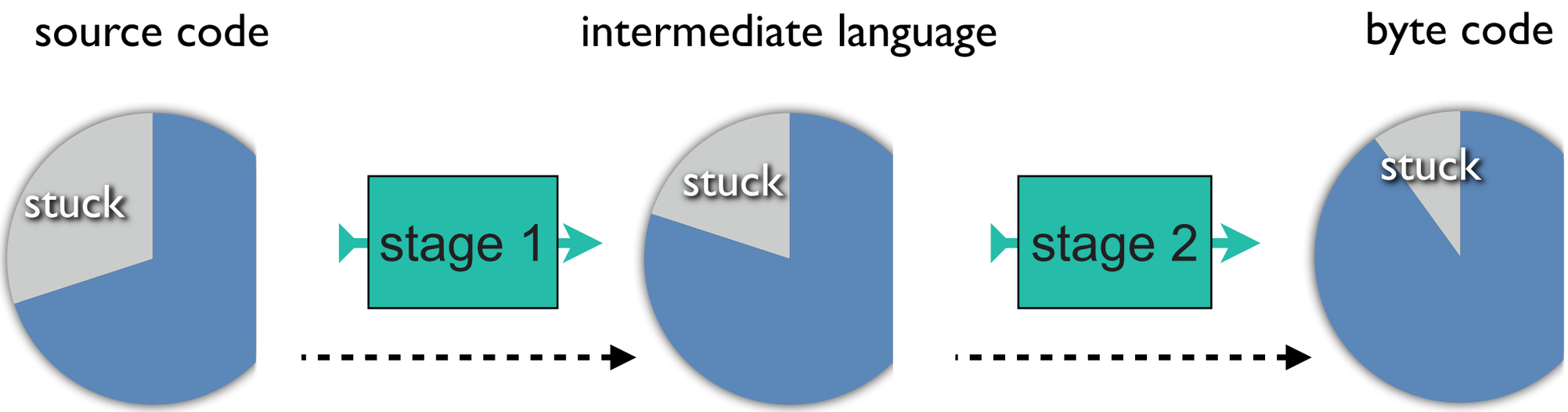
- heap access
- synchronisation
- thread creation
- method calls



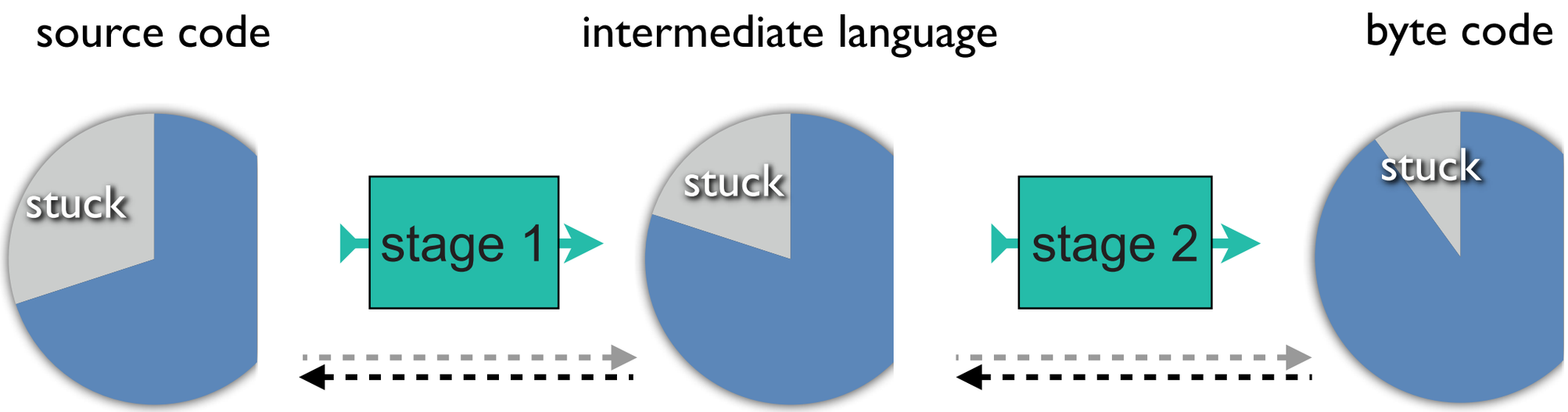
Stuck programs



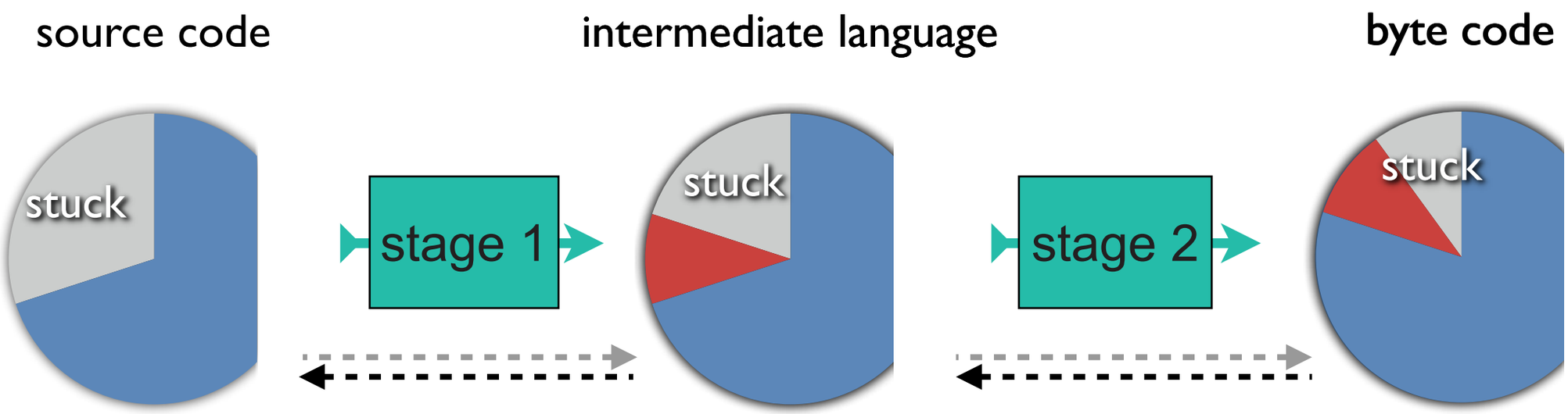
Stuck programs



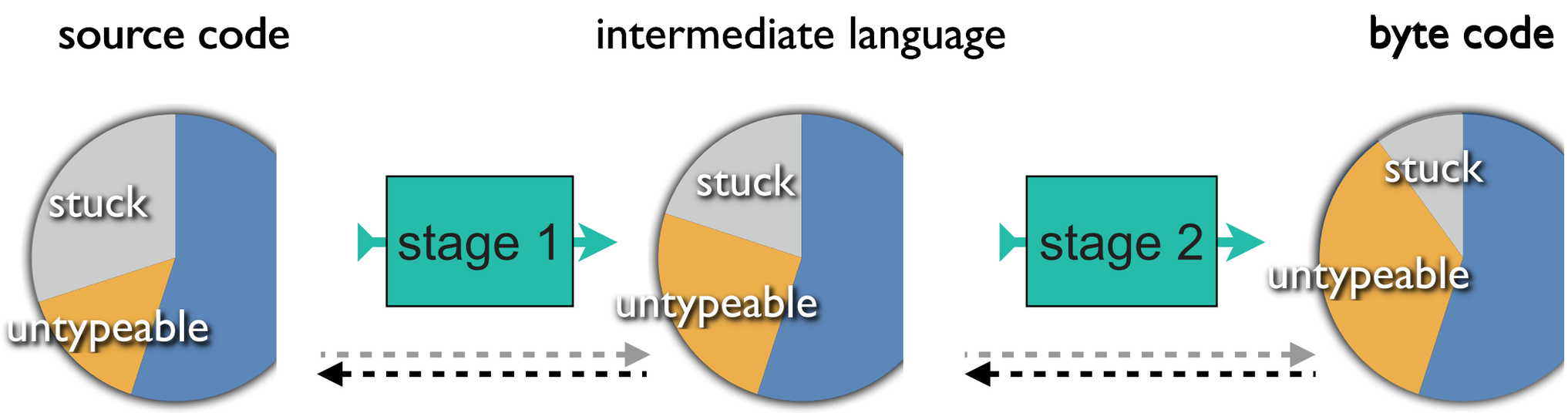
Stuck programs



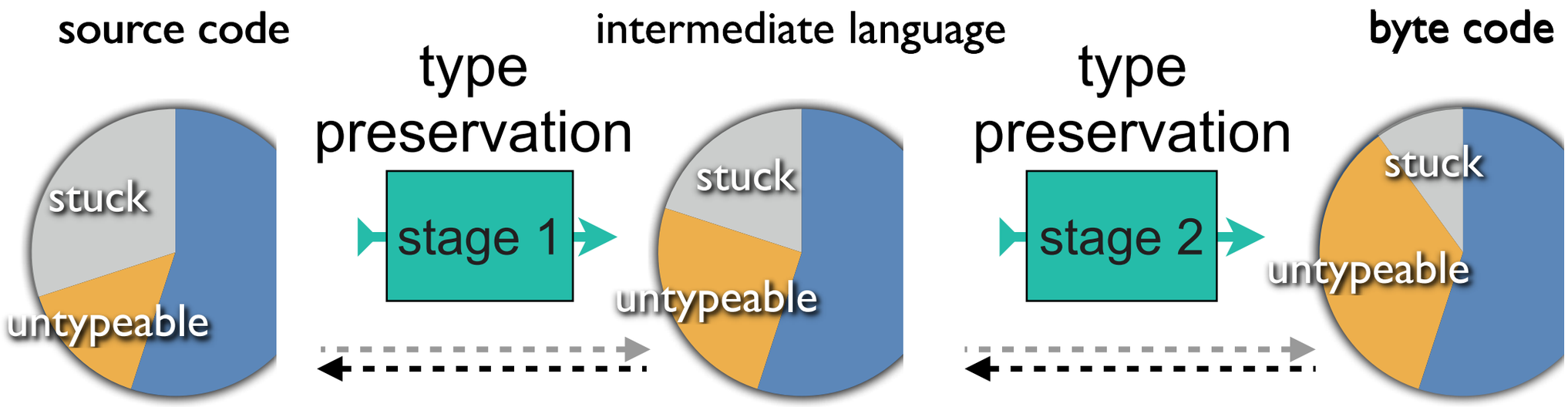
Stuck programs



Stuck programs

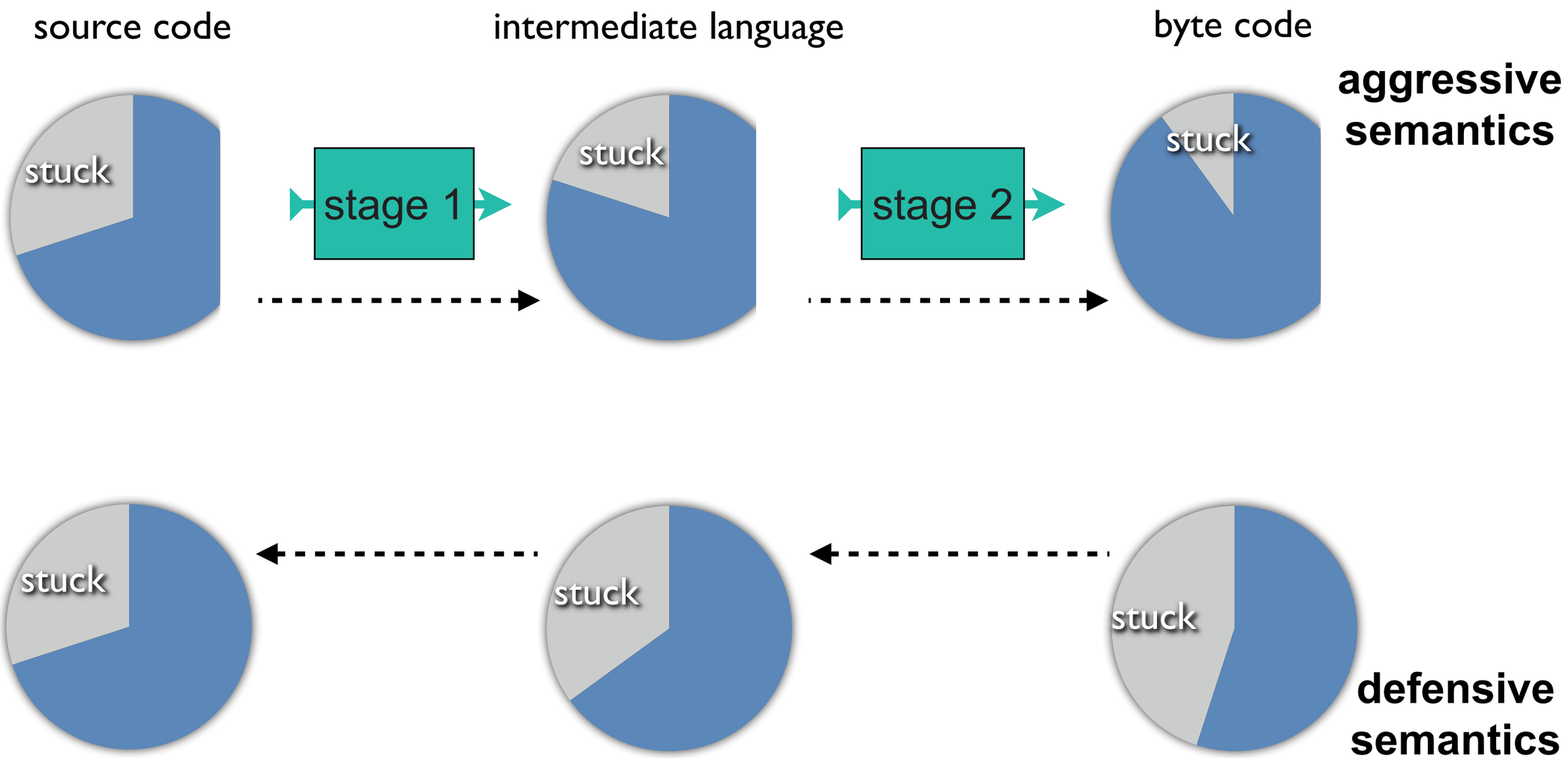


Stuck programs

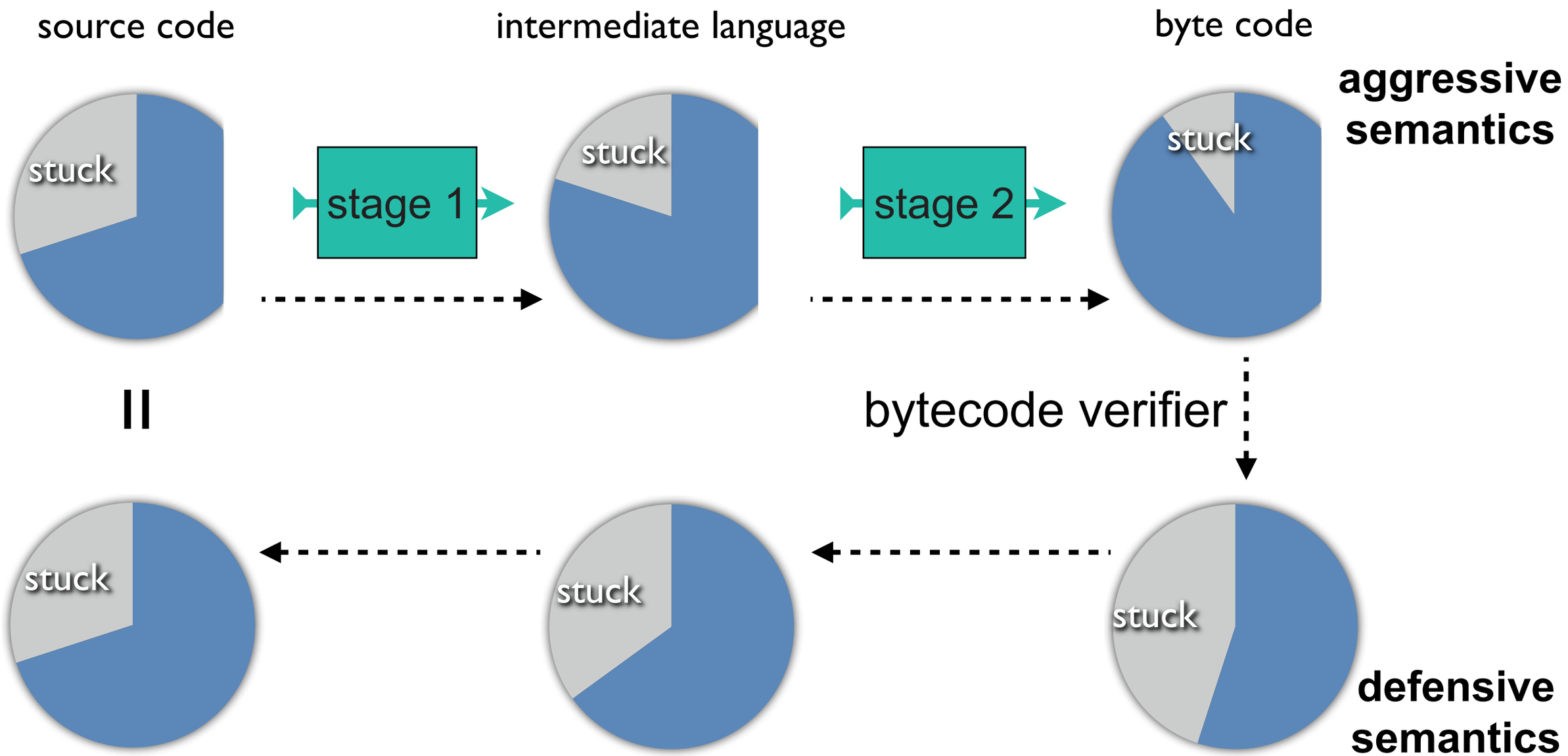


type safety | type safety | type safety

Stuck programs

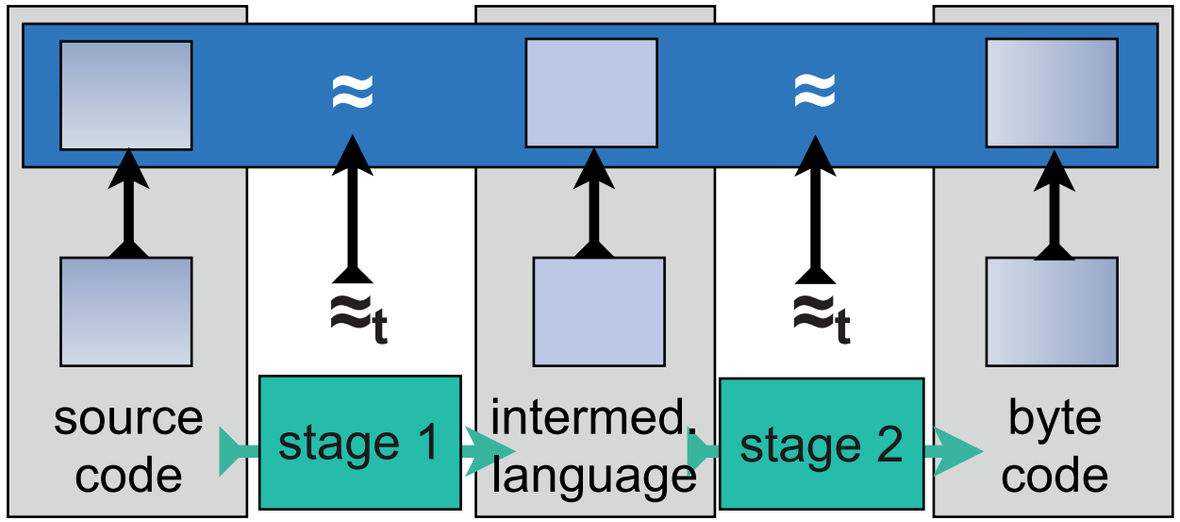


Stuck programs



Summary

- formal semantics for Java threads
- verified a simple compiler to byte code



Future work

- (deadlock & nontermination)
- Java memory model
- optimisations