Karlsruhe Institute of Technology

A unified machine-checked model for multithreaded Java

Andreas Lochbihler

IPD, PROGRAMMING PARADIGMS GROUP, COMPUTER SCIENCE DEPARTMENT

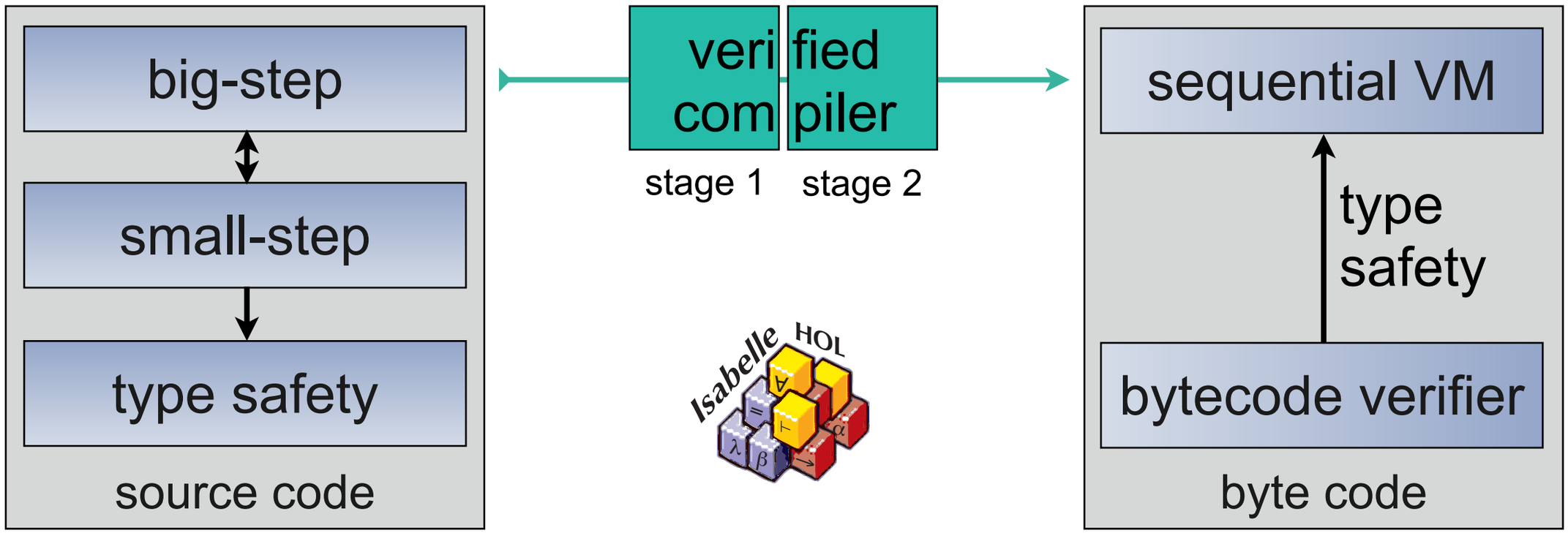

```
text {* The compiler correctness theorem *}

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'"
```

Motivation

- JMM formalisations by Sevcik/Aspinall and Petri/Huisman
  - no connection to operational semantics
- SC formalisations of Java (bytecode)
- Incorrect claims about the JMM
  - supported optimisations
  - litmus tests
- What is intra-thread consistency?
- Memory allocations and initialisations problematic

unified, machine-checked model of multithreaded Java

Jinja [Klein, Nipkow TOPLAS'06]



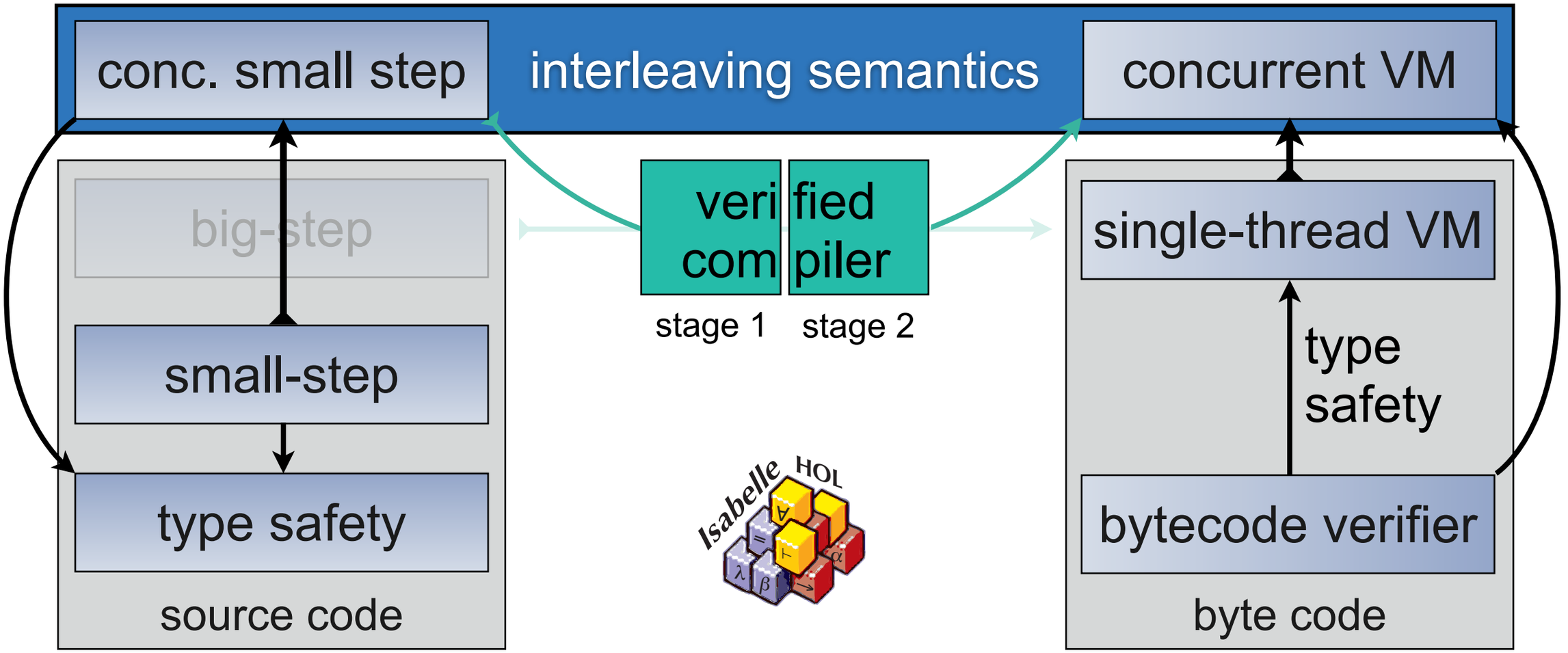
Java features:

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

not modelled:

- reflection & class loading
- interfaces
- threads

JinjaThreads [ESOP'10]



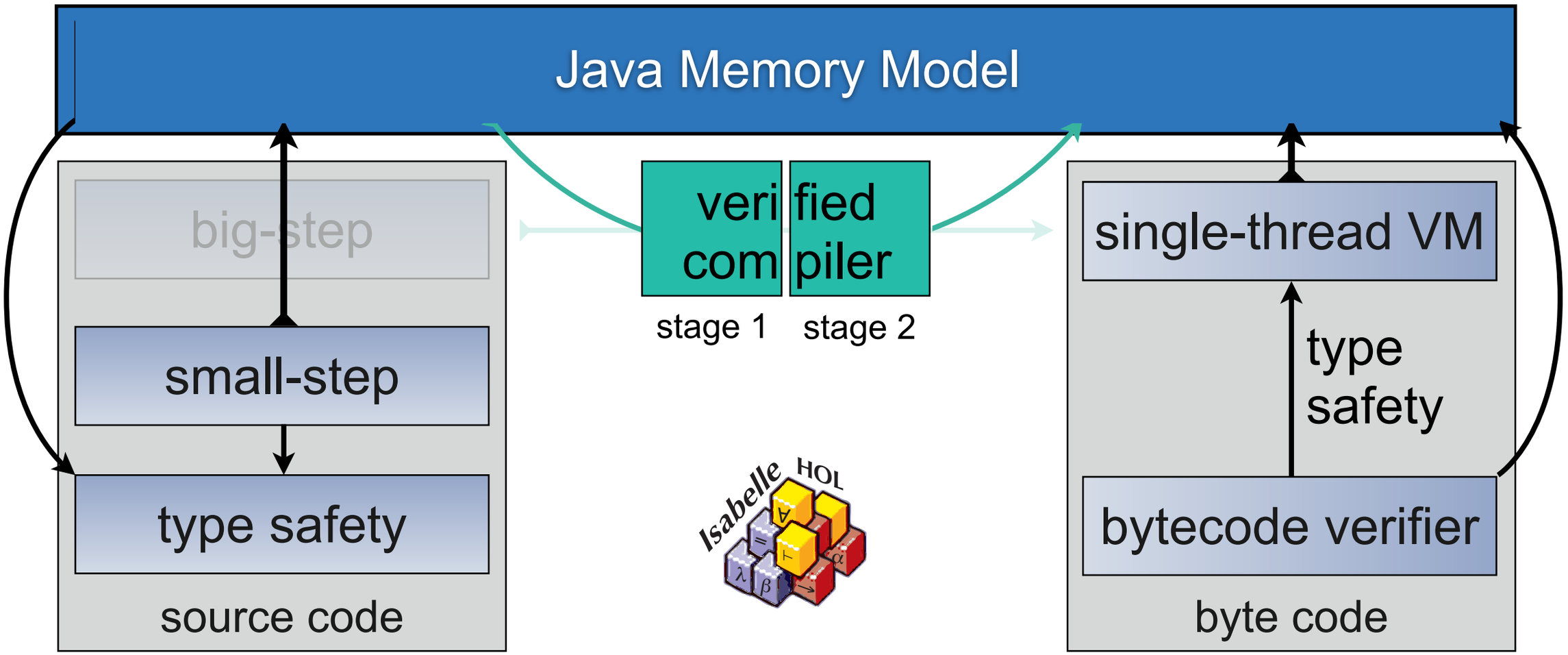
Java concurrency features:

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

not modelled:

- java.util.concurrent
- final fields

JinjaThreads



Prove:

- DRF guarantee
- Type safety
- No thin-air reads
- Compiler correctness

Isolated traces of threads

JMM: Type information and array lengths are not affected.

initially: v = 0; w = null;

r1 = v;	v = 1;	r3 = w;
r2 = new int[r1];		r4 = r3.length;
w = r2;		print r4; // when to print 1?

- r4 = r3.length unobservable
- intra-thread consistency spans threads

Interleaving semantics for types

single-thread semantics

$t \vdash \langle x, T\rangle \xrightarrow{as} \langle x', T'\rangle$

interleaved semantics

$\langle\!\langle \sigma, T\rangle\!\rangle \xrightarrow[as]{t} \langle\!\langle \sigma', T'\rangle\!\rangle$

Interleaving semantics for types

single-thread semantics

$t \vdash \langle x, T\rangle \xrightarrow{as} \langle x', T'\rangle$

interleaving

interleaved semantics

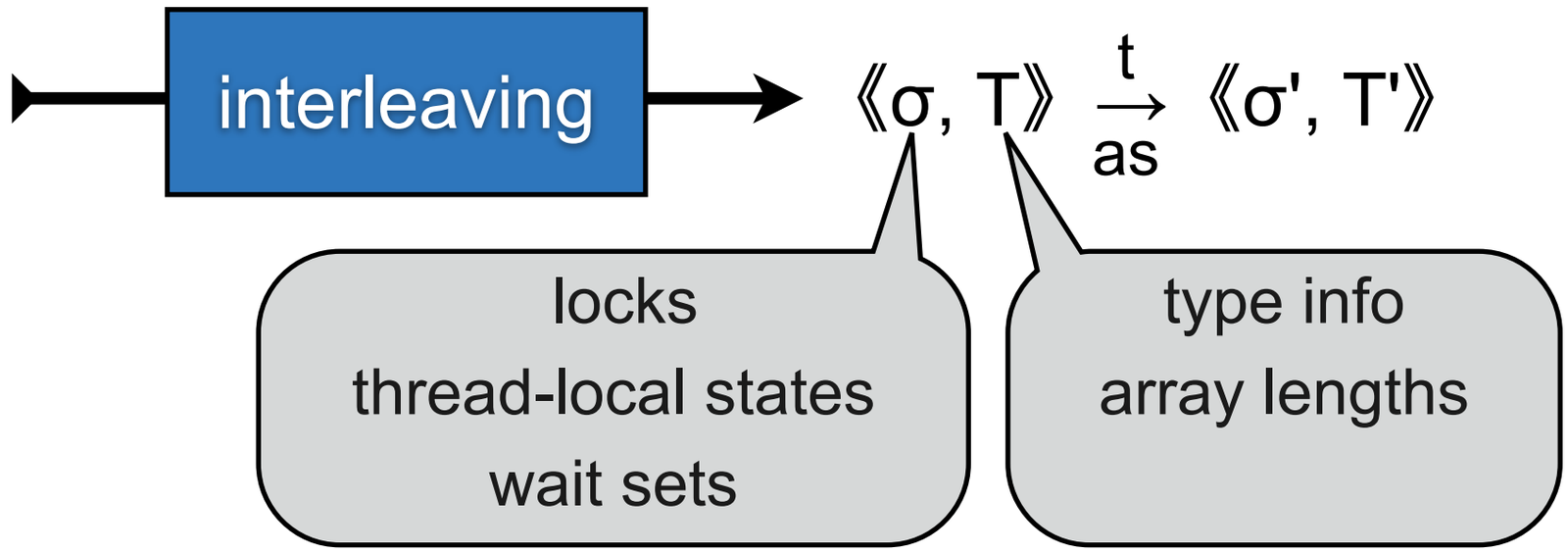
$\langle\!\langle \sigma, T\rangle\!\rangle \xrightarrow[as]{t} \langle\!\langle \sigma', T'\rangle\!\rangle$

type info
array lengths

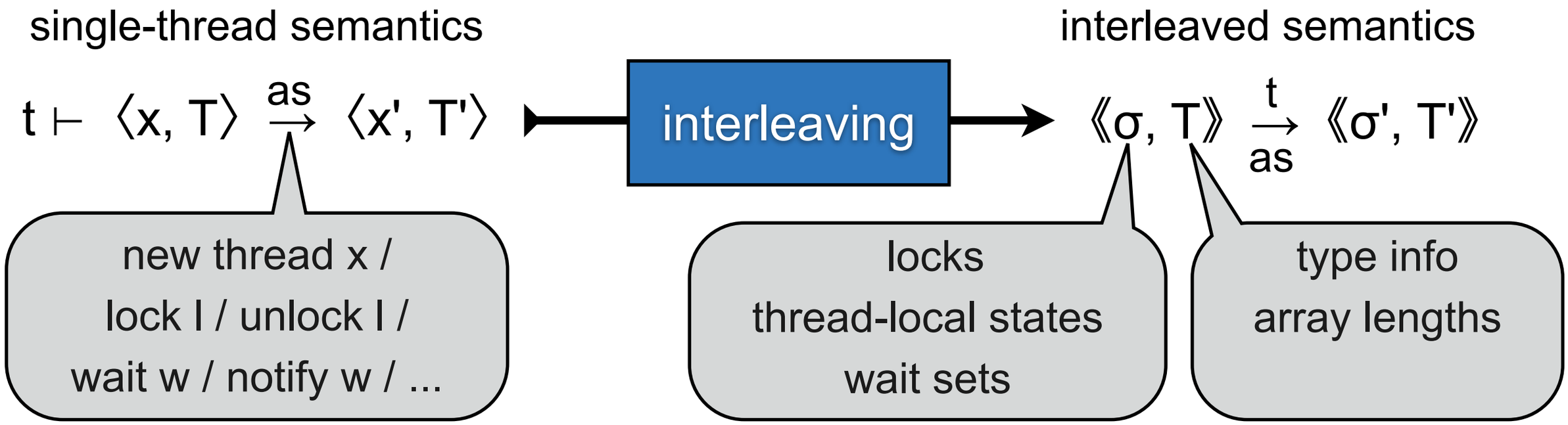
Interleaving semantics for types

single-thread semantics

$t \vdash \langle x, T\rangle \xrightarrow{as} \langle x', T'\rangle$



Interleaving semantics for types



Interleaving semantics for types

single-thread semantics

$t \vdash \langle x, T\rangle \xrightarrow{as} \langle x', T'\rangle$

interleaved semantics

$\langle\!\langle \sigma, T\rangle\!\rangle \xrightarrow[as]{t} \langle\!\langle \sigma', T'\rangle\!\rangle$

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

Interleaving semantics for types

single-thread semantics

$$t \vdash \langle x, T\rangle \xrightarrow{as} \langle x', T'\rangle$$

interleaving

interleaved semantics

$$\langle\!\langle \sigma, T\rangle\!\rangle \xrightarrow[as]{t} \langle\!\langle \sigma', T'\rangle\!\rangle$$

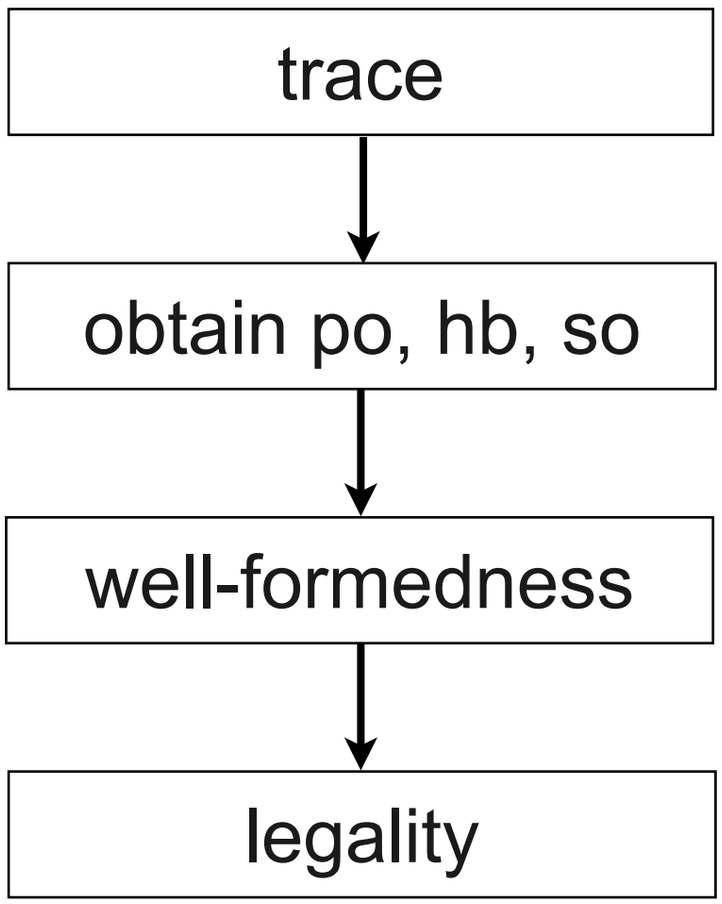
$$\frac{typeof_T\ a = Class\ C \qquad P \vdash C \leq Thread \qquad P \vdash C\ sees\ run() = body}{t \vdash \langle (addr\ a).start(), T\rangle \xrightarrow{[NewThread\ body]} \langle Unit, T\rangle}$$

$$\frac{\langle\!\langle \sigma, T\rangle\!\rangle \not\rightarrow}{\langle\!\langle \sigma, T\rangle\!\rangle \downarrow []} \qquad \frac{\langle\!\langle \sigma, T\rangle\!\rangle \xrightarrow[as]{t} \langle\!\langle \sigma', T'\rangle\!\rangle \qquad \langle\!\langle \sigma', T'\rangle\!\rangle \downarrow E}{\langle\!\langle \sigma, T\rangle\!\rangle \downarrow obs_t(as) : E}$$

trace E: $\langle\!\langle \sigma, T\rangle\!\rangle \Downarrow E := \exists E'.\ \langle\!\langle \sigma, T\rangle\!\rangle \downarrow E' \wedge E = concat(E')$

intra-thread consistency: program = maximal traces of interleaving

Axiomatic JMM



Deviations:

- no thread divergence actions
- thread interruption via volatile field
- ordinality of so and po
  - synchronisation order ω+ω
  - program order ω+ω
- no ssw edges and legality constraint 8

initialisations:

- happen before all other actions
- location type may depend on read values

```
v = 1;  |  r1 = (v == 1 ? new int[1] : new bool[1]);
        |  r2 = r1[0];                          // read 0 or false
```

DRF guarantee

Proof outline for correctly synchronized programs:

- If each read sees a write that happens before it, execution is SC.
  - If not, find **first** violating read r,
  - obtain **SC completion** from r on, and
  - show that r and the writes are part of an **hb data race**.
- by induction: justifying executions are SC.

SC completions

- SC defined w.r.t. happens-before
- traces coinductive
- coinductive characterisation of SC prefixes
  - allocation precedes read access

- construct SC completion via corecursion
  - cut-and-update property for thread semantics
  - requires type safety
  - restrict reads to read only type-correct values

disallows reordering with object creation:

r1 = x; y = new Object();	r2 = y; x = r2;

r1 == y?

Summary

- Unified model for multithreaded Java (bytecode)
- in Isabelle/HOL
- usable for proving metatheoretic results

Future work

- remedy type restriction
- type safety
- correctness of the bytecode verifier and compiler