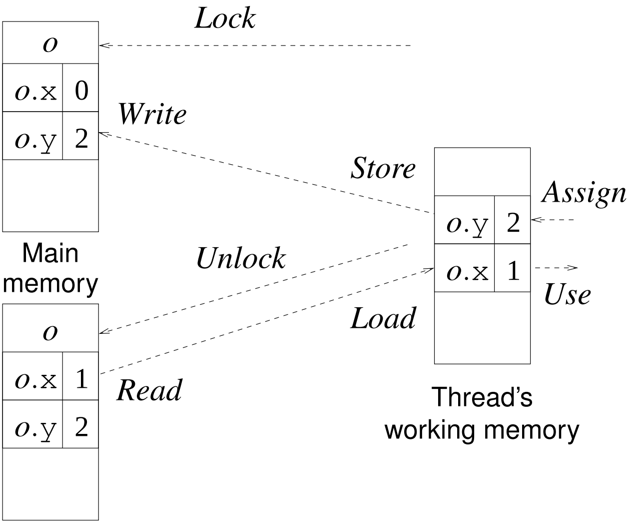
# The Java Memory Model: Operationally, Axiomatically, Denotationally

Alexander Knapp
Universität Augsburg

Joint work with Pietro Cenciarelli, Eleonora Sibilio
Università di Roma “La Sapienza”

# "Old" Java Memory Model: Actions

- Regulating information exchange between thread-local "working" and shared "main" memory



# “Old” Java Memory Model: Constraints

“Java Language Specification” (J. Gosling, B. Joy, G. Steele 1996, 2000)

- “A *Store* action by thread $\theta$ on variable $l$ must intervene between an *Assign* action by $\theta$ of $l$ and a subsequent *Load* action by $\theta$ of $l$. Less formally, a thread is not permitted to lose its most recent assign.”

$$\mathsf{a} : (\mathit{Assign}, \theta, l) \leq \mathsf{l} : (\mathit{Load}, \theta, l) \supset \\ \mathsf{a} : (\mathit{Assign}, \theta, l) \leq \mathsf{s} : (\mathit{Store}, \theta, l) \leq \mathsf{l} : (\mathit{Load}, \theta, l)$$

- “The actions on the master copy of any given variable on behalf of a thread are performed by the main memory in exactly the order that the thread requested.”

$$\mathsf{s} : (\mathit{Store}, \theta, l) \leq \mathsf{l} : (\mathit{Load}, \theta, l) \supset \mathit{writeof}(\mathsf{s}) \leq \mathit{readof}(\mathsf{l})$$

# Drawbacks of the "Old" Java Memory Model (1)

Breaking standard compiler optimsations (J.-W. Maessen, Arvind, X. Shen 2000)

- ☹ Disallowed when `p` and `q` reference the same location:

```
int i = p.x;          int i = p.x;
int j = q.x;    ↛     int j = q.x;
int k = p.x;          int k = i;
```

```
int i = p.x;    ||    p.x = 1;
int j = q.x;    ||    p.x = 2;
int k = p.x;    ||
```

Possible solution: Relaxation of action ordering constraints

# Drawbacks of the “Old” Java Memory Model (2)

Breaking standard concurrency idioms (W. Pugh 1999sqq.)

- ☹ “Double-checked locking”

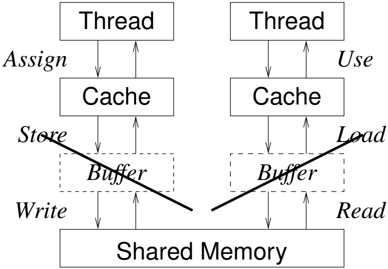
```
class Foo {
  private Helper helper = null;

  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null)          //  another thread
          helper = new Helper();  //  may see helper
      }                                //  uninitialised
    }
    return helper;
  }
}
```

Possible solution: Special actions for constructors and final fields

# Alternative Java Memory Models

- Omitting buffered *Store* and *Load* actions




- Relaxing unsynchronised *Read* actions
  - Pugh's approach (J. Manson, W. Pugh 1999sqq.)
  - Commit–Reconcile–Fence models (J.-W. Maessen, Arvind, X. Shen 2000)
  - Uniform memory model (Y. Yang, G. Gopalakrishnan, G. Lindstrom 2002)

# Overview

- "New" Java memory model
- Axioms for the Java memory model
  - Configuration structures
  - Configuration theories
  - Application to Java
- Operational semantics
- Towards denotational semantics

# “New” Java Memory Model: Overview (1)

JSR-133; J. Manson, W. Pugh, S. V. Adve 2005; “Java Language Specification” (J. Gosling, B. Joy, G. Steele, G. Bracha 2005)

- Causality-based model partially captured by happens-before consistency

  Happens-before: program and synchronisation order

  | x == y == 0 | |
  |---|---|
  | `r1 = x;` | `r2 = y;` |
  | `y = 1;` | `x = 1;` |
  | `r1 == r2 == 1` possible | |

  A *Read* $r$ of a variable $v$ is allowed to observe a *Write* $w$ to $v$ if
  - $r$ does not happen-before $w$; and
  - there is no *Write* $w'$ such that $w$ happens-before $w'$ and $w'$ happens-before $r$.

# "New" Java Memory Model: Overview (2)

- Commitment-based verification scheme for preventing "out-of-thin-air" results

| x == y == 0 | |
|---|---|
| `r1 = x;`<br>`if (r1 != 0)`<br>`  y = 1;` | `r2 = y;`<br>`if (r2 != 0)`<br>`  x = 1;` |
| only `r1 == r2 == 0` possible | |

Verification of an execution

$$E = (P, A, \underbrace{\leq_{po}}_{\text{prog. ord.}}, \underbrace{\leq_{so}}_{\text{sync. ord.}}, \underbrace{W}_{\text{write seen}}, \underbrace{V}_{\text{value written}}, \underbrace{\leq_{sw}}_{\text{sync. with}}, \underbrace{\leq_{hb}}_{\text{happens-before}})$$

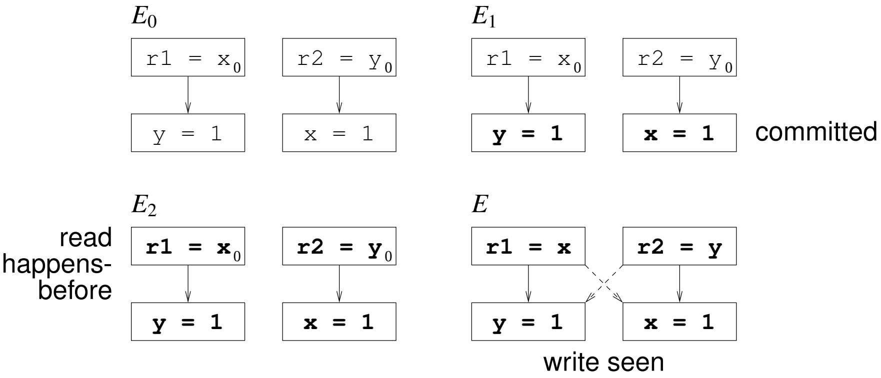
by committing actions $(C_i)_{i \in I} \subseteq A$ through executions

$$E_i = (P, A_i, \leq_{po,i}, \leq_{so,i}, W_i, V_i, \leq_{sw,i}, \leq_{hb,i})$$

# “New” Java Memory Model: Example

| x == y == 0 | |
|---|---|
| r1 = x; | r2 = y; |
| y = 1; | x = 1; |

r1 == r2 == 1 possible



# Problems of the "New" Java Memory Model

- Independent statements cannot necessarily be exchanged
  - in contrast to claim by J. Manson, W. Pugh, S. V. Adve (POPL'05)

x == y == z == 0

| Thread 1 | Thread 2 |
|---|---|
| `r1 = x;` | `r3 = z;` |
| `r2 = y;` | `if (r3 == 1) {` |
| `if (r1 == 1 && r2 == 1)` | `  x = 1; // order` |
| `  z = 1;` | `  y = 1; // matters` |
| | `}` |
| | `else` |
| | `  y = 1;` |
| | `  x = 1;` |
| | `}` |

- Integration into operational semantics
  - Guessing of final execution
  - Connection between actions and program

# Configuration Structures

Configuration structure $(E, \mathscr{C})$ with $\mathscr{C} \subseteq \wp E$, $\mathscr{C} \neq \emptyset$

- Events $E$ in a concurrent system
- Configuration $C \in \mathscr{C}$ partial, concurrent computation
- Subconfigurations $\mathscr{C}(C) = \{D \in \mathscr{C} \mid D \subseteq C\}$

satisfying for each $C \in \mathscr{C}$

- Coincidence-freedom: $a \neq b \in C \supset \exists D \in \mathscr{C}(C) \,.\, a \in D \iff b \notin D$
  - ensures partial order of events in a configuration

    $a \leq_C b \iff \forall D \in \mathscr{C}(C) \,.\, b \in D \supset a \in D$
- Finiteness: $a \in C \supset \exists D \in \mathscr{C}(C) \,.\, a \in D \wedge |D| < \infty$
  - ensures finite causes
- Monotonicity: $\forall D \in \mathscr{C}(C) \,.\, a \leq_D b \supset a \leq_C b$
  - ensures preservation of event order over extensions

(introduced by G. Plotkin, R. van Glabbeek 1995)

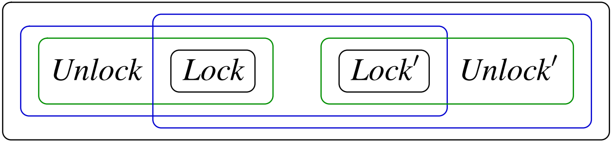
# Stable Configuration Structures

- In stable configuration structures, causality can be faithfully represented by partial orders.

Configuration structure $(E, \mathscr{C})$ stable

- Connectedness: $\forall \emptyset \neq C \in \mathscr{C} \,.\, \exists a \in C \,.\, C \setminus \{a\} \in \mathscr{C}$
  - implies coincidence freeness
- Closed under non-empty bounded unions and intersections
  - $A, B \in \mathscr{C}$ bounded, if $A, B \in \mathscr{C}(C)$ for some $C \in \mathscr{C}$

But: Too strong a requirement




not stable

# Configuration Theories

- Logic for configuration structures
- Sequents of the form

  $$\rho : C_1, \ldots, C_m \Rightarrow D_1, \ldots, D_n \qquad (C_i,\, D_j \text{ partial orders, } \rho_{ij} : C_i \hookrightarrow D_j)$$

  $C_i$ premises (conjunctive), $D_j$ conclusions (disjunctive)
- Interpretation: Partial orders $C_i$ are combined and extended by $\rho$ into partial orders $D_j$

(introduced by P. Cenciarelli 2002)

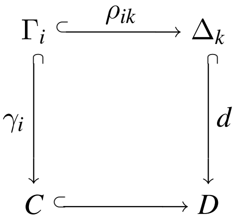
But restrict interpretation to computations

- Computation of $C \in \mathscr{C}$: maximal stable sub-configuration structure $\mathscr{D} \subseteq \mathscr{C}(C)$ with $C \in \mathscr{D}$

# Configuration Theories: Satisfaction

$(E, \mathscr{C}) \models \rho : \Gamma \Rightarrow \Delta$

- if $\Gamma$ can be interpreted in a computation (of) $C \in \mathscr{C}$
- then there is a computation (of) $D \in \mathscr{C}$ with $C \in \mathscr{C}(D)$ such that a $\Delta_k$ can be consistently interpreted in $D$



# Application to Java: Axioms (1)

- Ordering

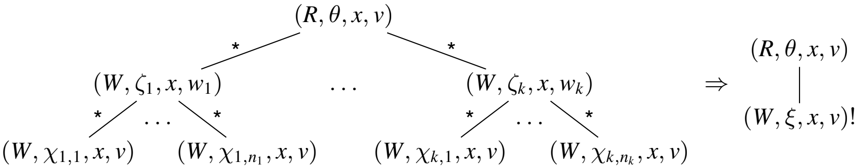
$$a \quad b \Rightarrow \begin{array}{c} a \\ | \\ b \end{array} , \begin{array}{c} b \\ | \\ a \end{array} \qquad \text{if } a \succ b$$

where $a$ affects $b$ if

  - $(W, \theta, x) \succ (\theta, x)$
  - $(\theta, x) \succ (U, \theta)$
  - $(L, \theta) \succ (\theta, x)$
  - $(L, \theta) \succ (U, \theta)$
  - $(\theta, m) \succ (L, \zeta, m)$

# Application to Java: Axioms (2)

- Reading from “shared memory”
  - Values read from synchronised threads are most recent




if $v \neq w_i$ for all $1 \leq i \leq k$

- Locking and unlocking


if $\theta \neq \zeta$

# Integration with Operational Semantics

- Integration of Java configurations into operational semantics
  - Prescient extension of Java configuration
  - Validate guess by executing program and confirming events
- Java configurations represent mainly happens-before
  - Relaxation of ordering on different variables in a thread
  - Dependency of *Read* on *Write* added
- But: Not enough to capture causality

`x == y == 0`

| r1 = x; | r2 = y; |
|---|---|
| **if** (r1 != 0) | **if** (r2 != 0) |
| y = 1; | x = 1; |

only `r1 == r2 == 0` possible

  - Additionally record dependencies of *Write* on *Read*
  - Confirm dependencies when validating a configuration

# Tagged Java Configurations

- Dependencies set of *Read* events
- Tagged Java configuration $(C, t)$
  - Java configuration $C$
  - tagging $t : \{e \mid e : (W) \in C\} \to \mathbb{B}$ $t(e) = tt \Leftrightarrow$ "prescient"
- Extending a tagged Java configuration $\eta \oplus A$
  - conservative extension of order and tagging
  - if $A = (W)$, new event tagged as "prescient"
- Confirming a *Write* $\eta \downarrow_\delta (W)$
  - prescient $e : (W) \in \eta$ with all previous *Write*s non-prescient
  - with $d \leq e$ for all $d \in \delta$
  - make $e$ non-prescient

# A Simple Java Fragment

$$
\begin{aligned}
\textit{D-Term} &::= \textit{D-Stm} \mid \textit{D-Expr} \\
\textit{D-Stm} &::= \textit{Stm}\ \textit{Dep} \\
\textit{D-Expr} &::= \textit{Expr}\ \textit{Dep} \\
\textit{Stm} &::= \texttt{;} \mid \textit{Var} = \textit{D-Expr}\ \texttt{;} \mid \textit{D-Stm}\ \textit{Stm} \\
&\quad \mid \texttt{if}\ \texttt{(}\ \textit{D-Expr}\ \texttt{)}\ \textit{D-Stm}\ \texttt{else}\ \textit{D-Stm} \\
&\quad \mid \texttt{synchronized}\ \texttt{(}\ \textit{Mon}\ \texttt{)}\ \textit{D-Stm} \\
&\quad \mid \textit{synchronized}\ \texttt{(}\ \textit{Mon}\ \texttt{)}\ \textit{D-Stm} \\
\textit{Expr} &::= \textit{Val} \mid \textit{Lit} \mid \textit{Var} \mid \textit{Expr}\ \textit{BOp}\ \textit{Expr}
\end{aligned}
$$

- `x = 1;` becomes $(\texttt{x} = (\texttt{1})_{\emptyset}\ \texttt{;})_{\emptyset}$

# Operational Semantics

[var] $(\theta, (x)_\delta), \eta \to (\theta, (v)_{\delta,(Read,\theta,x,v)}), \eta \oplus (Read, \theta, x, v)$

[assign2] $(\theta, (x = (v)_{\delta_0} \texttt{;})_\delta), \eta \to (\theta, (\texttt{;})_\delta), \eta \downarrow_{\delta,\delta_0} (Write, \theta, x, v)$

[if4]
$$\frac{(s_1)_{\delta,\delta_1}, \eta \to (s_1')_{\delta,\delta_1'}, \eta' \quad (s_2)_{\delta,\delta_2}, \eta \to (s_2')_{\delta,\delta_2'}, \eta'}{(\texttt{if (} (v)_{\delta_0} \texttt{)} \ (s_1)_{\delta_1} \ \texttt{else} \ (s_2)_{\delta_2})_\delta, \eta \to (\texttt{if (} (v)_{\delta_0} \texttt{)} \ (s_1')_{\delta_1'} \ \texttt{else} \ (s_2')_{\delta_2'})_\delta, \eta'}$$

[syn1] $(\theta, \texttt{synchronized (} m \texttt{)} \ p), \eta \to (\theta, \mathit{synchronized} \ \texttt{(} m \texttt{)} \ p), \eta \oplus (Lock, \theta, m)$

[syn3] $(\theta, \mathit{synchronized} \ \texttt{(} m \texttt{)} \ (\texttt{;})_{\delta_0}), \eta \to (\theta, \texttt{;}), \eta \oplus (Unlock, \theta, m)$

[pre] $T, \eta \to T, \eta \oplus (W)$

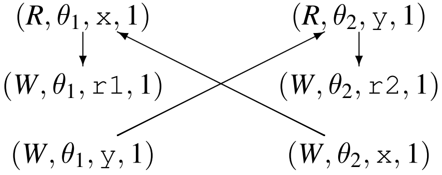
# Operational Semantics: Example (1)

`x == y == 0`

| | |
|---|---|
| `r1 = x;` | `r2 = y;` |
| `y = 1;` | `x = 1;` |

`r1 == r2 == 1` possible

- Configuration to be confirmed



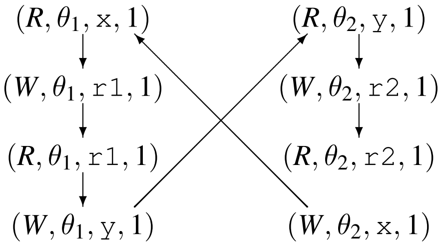
# Operational Semantics: Example (2)

`x == y == 0`

| | |
|---|---|
| `r1 = x;` | `r2 = y;` |
| **`if`** `(r1 == 1)` | **`if`** `(r2 == 1)` |
| `  y = 1;` | `  x = 1;` |
| | **`else`** |
| | `  x = 1;` |

`r1 == r2 == 1` possible

- Configuration to be confirmed



# Operational Semantics: Correctness

- From computation $\vec{\gamma} = (T_0, \eta_0) \to \cdots \to (T_n, \eta_n)$ construct well-formed execution

$$exec(\vec{\gamma}) = (T, |\eta_n|, po(\vec{\gamma}), so(\vec{\gamma}), W(\vec{\gamma}), V(\vec{\gamma}), sw(\vec{\gamma}), hb(\vec{\gamma}))$$

- Construct validating sequence $(X(\vec{\gamma})_i, C(\vec{\gamma})_i)_{0 \leq i \leq n}$ by inductively committing minimal events of $\eta_i \setminus C(\vec{\gamma})_i$

# Operational Semantics: Incompleteness (1)

- In order to handle

```
x == y == z == 0
```

```
r1 = x;
r2 = y;
if (r1 == 1 && r2 == 1)
  z = 1;
```

```
r3 = z;
if (r3 == 1) {
  x = 1;
  y = 1;
}
else
  y = 1;
  x = 1;
}
```

- use

$$\frac{(s_1)_{\delta,\delta_1}, \eta \longrightarrow^+ (s_1')_{\delta,\delta_1'}, \eta' \quad (s_2)_{\delta,\delta_2}, \eta \longrightarrow^+ (s_2')_{\delta,\delta_2'}, \eta'}{\begin{array}{l}(\texttt{if ( } (v)_{\delta_0} \texttt{ ) } (s_1)_{\delta_1} \texttt{ else } (s_2)_{\delta_2})_\delta, \eta \longrightarrow \\ \qquad (\texttt{if ( } (v)_{\delta_0} \texttt{ ) } (s_1')_{\delta_1'} \texttt{ else } (s_2')_{\delta_2'})_\delta, \eta'\end{array}}$$

# Operational Semantics: Incompleteness (2)

- What about

|  | x == y == 0 |
|---|---|
| `r3 = x;`<br>`if (r3 == 0)`<br>`  x = 42;`<br>`r1 = x;`<br>`y = r1;` | `r2 = y;`<br>`x = r2;` |

`r1 == r2 == r3 == 42` possible

- "A compiler could determine that the only values ever assigned to `x` are $0$ and $42$. From that, the compiler could deduce that, at the point where we execute `r1 = x`, either we had just performed a write of $42$ to `x`, or we had just read `x` and seen the value $42$. In either case, it would be legal for a read of `x` to see the value $42$." (J. Manson, W. Pugh, S. V. Adve 2005)

# Towards a Denotational Semantics

- Configuration structure $[\![T]\!]$ for program $T$
  - for each operational computation configurations $C$ of events
    - events generated by [var], [pre], [syn1], [syn3]
    - downwards closure
- $[\![T]\!]$ satisfies Java axioms

# Conclusions and Future Work

- Integration of “new” Java memory model with operational semantics
  - Axioms for memory based on configuration theories
  - Dependencies for causality
- How to capture global static analyses?
- Transactional Java?