A Proposed Semantics for Multithreaded Java

Bill Pugh
Basic Framework

• Operational semantics
• Actions occur in a global order
  – consistent with original order in each thread
    • except for prescient writes
• If program not correctly synchronized
  – reads non-deterministically choose which value to return from set of candidate writes
Terms

• Variable
  – a field or array element

• Value
  – a primitive type or reference to an object

• Local
  – a value stored in a local or on the stack
Write Sets

• Sets of writes
  – a write is a variable/value pair
• allWrites: all writes performed
• Threads/monitors/voltiles have/know:
  – overwritten: a set of writes known to be overwritten
  – previous: a set of writes known to be in the past
Multimap basics

• These are all monotonic multimaps
  – they only grow
• Standard set operations apply
• Applying a multimap to a variable:
  – $M(v) = \{ w \mid <v, w> \text{ in } M \}$
• Writes have hidden GUID
  – two writes of 42 are distinct
• Subscripts used to indicate thread, monitor or volatile that “owns/knows” a multimap
Read/Write Semantics  
(in thread $t$)

- **ReadNormal(Variable $v$)**
  \[
  w = \text{choose}( \text{allWrites}(v) - \text{overwritten}_t(v) ) \\
  \text{return } w
  \]

- **WriteNormal(Variable $v$, Value $w$)**
  \[
  \text{overwritten}_t(v) \cup= \text{previous}_t(v) \\
  \text{previous}_t(v) += w \\
  \text{allWrites}(v) += w
  \]
Invariants

- overwritten$_t \subset$ previous$_t \subseteq$ allWrites

- For correctly synchronized code, at point where you access variable $v$:
  - previous$_t(v) = \text{allWrites}(v)$
  - $| \text{allWrites}(v) - \text{overwritten}_t(v) | = 1$
Initial write of default value

- When a variable $v$ is created, all threads $t$ know the initial write $w$ of the default value to that variable is previous
  - $\text{allWrites}(v) = \{ w \}$
  - $\text{previous}_t(v) = \{ w \}$
  - $\text{overwritten}_t(v) = \{ \}$
Lock/Unlock Semantics

- **Lock(Monitor \( m \))**
  
  wait until lock on \( m \) has been acquired
  
  overwritten\( _t \) \( \cup \) overwritten\( _m \)
  
  previous\( _t \) \( \cup \) previous\( _m \)

- **Unlock(Monitor \( m \))**
  
  overwritten\( _m \) \( \cup \) overwritten\( _t \)
  
  previous\( _m \) \( \cup \) previous\( _t \)
  
  release lock
Example

\[ x = 1 \]

unlock A

\begin{align*}
\text{overwritten}_A(x) &= \{0\} \\
\text{previous}_A(x) &= \{0,1\}
\end{align*}

\[ x = 2 \]

unlock A

\begin{align*}
\text{overwritten}_A(x) &= \{0,1\} \\
\text{previous}_A(x) &= \{0,1,2\}
\end{align*}

x can print 2 or 3, but not 0 or 1

\[ x = 3 \]

unlock B

\begin{align*}
\text{overwritten}_B(x) &= \{0\} \\
\text{previous}_B(x) &= \{0,3\}
\end{align*}

lock A

\[ x = 1 \]

unlock A

\begin{align*}
\text{overwritten}_A(x) &= \{0,1\} \\
\text{previous}_A(x) &= \{0,1\}
\end{align*}

\[ x = 2 \]

unlock A

\begin{align*}
\text{overwritten}_A(x) &= \{0,1\} \\
\text{previous}_A(x) &= \{0,1,2\}
\end{align*}

\[ x = 4 \]

print x

\begin{align*}
\text{overwritten}_B(x) &= \{0\} \\
\text{previous}_B(x) &= \{0,3\}
\end{align*}

must print 4

\[ x = 4 \]

print x

\begin{align*}
\text{overwritten}_B(x) &= \{0,1,2\} \\
\text{previous}_B(x) &= \{0,1,2, 3\}
\end{align*}
Happens-before relationships

previous: reachable backwards
overwritten: exists a backwards paths where it is overwritten
Volatile Semantics

• Very similar to monitors

• ReadVolatile(Variable $v$)
  overwritten$_t \cup = \text{overwritten}_v$
  previous$_t \cup = \text{previous}_v$
  return volatileValue$_v$

• WriteVolatile(Variable $v$, Value $w$)
  overwritten$_v \cup = \text{overwritten}_t$
  previous$_v \cup = \text{previous}_t$
  volatileValue$_v = w$
Synchronization optimizations

- Thread local monitors are no-ops
  - Information known by monitor must be subset of information known by thread
- Thread local volatile fields can be treated as non-volatile fields
- Recursive locks are no-ops
  - recursive lock can’t reveal any new information
  - recursive unlock won’t be read
Lock Coarsening

• If you guarantee that no other thread acquires a lock between a unlock and lock
  – information written by unlock in monitor will not be read by any other thread
  – lock will not acquire any new information
  – Unlock and locks have no effect
Problem

\[ x = y = 0 \]

Thread 1

\[ j = y \]
\[ i = 1 \]

Thread 2

\[ i = x \]
\[ j = 1 \]

Can this result in \( i = 1 \) and \( j = 1 \)?
Need Prescient Writes

- A thread may perform a write early only if the following conditions hold
  - The write is guaranteed to happen
  - The variable written to and the value written are fixed
    - including across non-deterministic values returned by reads
  - it is not moved past another access to that variable
Prescient writes (continued)

• A Prescient write may not be reordered with a preceding lock action unless the previous unlock on that monitor (if any) is guaranteed to have been done by the same thread
  – circularity problem?
Prescient Reads?

• Prescient Reads are not needed
• Reads can be done early
  – so long as value read is guaranteed to not be in overwritten set at original point of read
Very Prescient Reads

- Can even do forward substitution across lock
  - At point of lock (and of read), no other thread knows \( x=1 \) to be previous
  - cannot learn that \( x=1 \) is overwritten at lock
Requires G-CRF

Can this result in $i = 2$ and $j = 1$?
Example Execution

- **T1**: $a = 1$
  \[aW = \{0, 1\}; o_1 = \{0\}; p_1 = \{0,1\}\]

- **T2**: $a = 2$
  \[aW = \{0, 1, 2\}; o_2 = \{0\}; p_2 = \{0,1\}\]

- **T1**: $i = a$
  choose 2 from $\{0, 1, 2\} - \{0\}$

- **T2**: $j = a$
  choose 1 from $\{0, 1, 2\} - \{1\}$
Final fields

• Have to track data dependence
• Attach overwritten information to final fields and to local values
  – don’t need previous; sync arising from final should not be used for writes
• A local value consists of a <value, overwritten> tuple
Changes

• Changes semantics for reads/writes of normal fields and final fields
  – Operations now take an address (a local value) and a field
    • arrays treated as records

• Constructor termination freezes the appropriate final fields
  – details with constructor chaining
Read/Write Semantics

• ReadNormal(Value <a, oF>, Field f)
  Let v be variable referenced by a.f
  \( w = \text{choose}(\text{allWrites}(v) \setminus \text{overwritten}_t(v) \setminus oF) \)
  return <w, oF>

• WriteNormal(Value <a, —>, Field f, Value <w, —>)
  Let v be variable referenced by a.f
  overwritten\(_t(v) \cup= \text{previous}_t(v)\)
  previous\(_t(v) += w\)
  allWrites(v) += w
Final Semantics

• ReadFinal(Value \(<a, oF>\), Field \(f\))
  Let \(v\) be final variable referenced by \(a.f\)
  \(oF' = \text{overwritten}_v\)
  return \(<\text{finalValue}_v, oF \cup oF'>\)

• WriteFinal(Value \(<a, \rightarrow>\), Field \(f\), Value \(<w, \rightarrow>\))
  Let \(v\) be variable referenced by \(a.f\)
  \(\text{finalValue}_v = w\)

• FreezeFinal(Value \(<a, \rightarrow>\), Field \(f\))
  Let \(v\) be variable referenced by \(a.f\)
  \(\text{overwritten}_v = \text{overwritten}_t\)
Pseudo-Final fields

• If you store a reference to an object $b$ into the heap before the B constructor for $b$ terminates
• Another threads loads that reference
• And synchronization doesn’t guarantee that the load occurs after the B constructor terminates
  – all final fields of B in $b$ become pseudo-final
Pseudo-Final fields

• Each read of a pseudo-final variable $v$ non-deterministically returns either the default value or finalValue$_v$
  – overwritten$_v$ is ignored
On pseudo-final fields

• In reality, having one improperly synchronized reference to an object shouldn’t affect reads of final fields through properly synchronized references

• But I couldn’t make the semantics work
Comparison with other models

• Post-hoc models
  – only tell you if a particular execution is legal
    • circularity issues

• Other operational models
  – impose weird little constraints not needed to enforce SC for correctly synchronized programs (or for safety reasons)
  – only arise in contrived cases
Simple memory models

• Some models have a simple global/cache memory model
  – one global memory
  – one cache per thread
• Actions get committed to global memory in some total order
• Updates applied to local cache in some total order
Models based on reordering

• A model based on reordering depends on rules for reordering
  – can you reorder read of t3.x?
    t2 = t1.x;
    t3 = A.p;
    t4 = t3.x;
  – For example, to
    t2 = t1.x;
    t3 = A.p;
    if (t3 == t1) t4 = t2
    else t4 = t3.x
Behavior prohibited if dependent reads can’t be reordered

- Initially
  \[ p.next = \text{null} \]
- Thread 1:
  \[ p.next = p \]
- Thread 2:
  \[ \text{List tmp = } p.next; \]
  \[ \text{if (tmp == p && tmp.next == null) } \{ \]
  \[ \text{    // Can’t happen under CRF} \]
  \[ \text{\} } \]
Do we care about behaviors no one cares about?

• What if memory model prohibits a weird behavior
  – don’t know of any compiler optimizations that would perform it
  – don’t know of any architectures that would perform it
• is this a problem?
Why we should care

• If behavior is prohibited, need to prove:
  – architecture doesn’t allow it
  – compiler optimizations don’t allow it

• Even if a compiler doesn’t allow it, proving that is a burden
Challenge

• I don’t know of any examples of behaviors prohibited by my approach
  – except for those we must prohibit
  – and edge cases of final fields of objects escaping their constructors

• but I need outside eyes