Safe Locking for Multi-threaded Java

Einar Broch Johnsen, Thi Mai Thuong Tran, Olaf Owe, Martin Steffen

University of Oslo, Norway

NWPT'10

10-12 November, 2010

Motivation

- Concurrency control mechanisms for high-level programming languages, such as Java
  - lexical scope: synchronized-methods/blocks
  - non-lexical scope: lock and unlock operators to acquire and release a lock in non-lexical scope.
- Runtime errors and unwanted behaviors.

```
import java.util.concurrent.locks;
public class ConditionTest {
....................
  private final Thread producer, consumer;
  private final ReentrantLock l;
  class Consumer implements Runnable {...}
  class Producer implements Runnable {
        ....................
    public void put(Integer key, Boolean value) {
      l.lock();
      l.lock(); // 2 time lock
      try { collection.put(key, value);
      ..................................
      } finally { l.unlock(); } // 1 times unlock
    ...
}
```

Consumer is hanging

```
Producer: adding 1 to collection.
Consumer: waiting 10 seconds for 2345 to arrive ...
Producer: adding 4 to collection.
Producer: adding 66 to collection.
Producer: adding 9 to collection.
Producer: adding 2435 to collection.
Producer: exiting.
```

Lock Handling in Java: release a free lock

```
import java.util.concurrent.locks;
public class ConditionTest {
....................
  private final Thread producer, consumer;
  private final ReentrantLock l;
  class Consumer implements Runnable {...}
  class Producer implements Runnable {
        ...............
    public void put(Integer key, Boolean value) {

      l.lock(); // 1 time lock
      try { collection.put(key, value);
          ...........................
      } finally {
                  l.unlock();
                  l.unlock();
                } // 2 times unlock
    ...
}
```

Lock Handling in Java: Report of lock errors at run-time

```
Producer: adding 1 to collection.
.................................
Exception in ... java.lang.IllegalMonitorStateException
  at ...ReentrantLockSync.tryRelease(ReentrantLock.java:127)
  at ...release(AbstractQueuedSynchronizer.java:1239)
  at ...ReentrantLock.unlock(ReentrantLock.java:431)
  at ...ConditionTestProducer.put(ConditionTest.java:110)
  ............
  at java.lang.Thread.run(Thread.java:662)
.......
Consumer: exiting.
```

Goals of our work

- Semantics for lock handling as in Java.
- Static type and effect system for safe usage of re-entrant locks.
- Soundness proof
- Details and proofs: See our technical report[Johnsen et al., 2010]

Challenges

- Dynamic creation of objects, threads, and especially locks.
- Identities of locks are available at user-level
- Locks are re-entrant
- Aliasing
- Passing locks between threads
- Multi-threading/concurrency

$$
\begin{array}{rcl}
D \in \mathit{Classes} & ::= & \mathsf{class}\ \ C(\vec{f}{:}\vec{T})\{\vec{f}{:}\vec{T}; \vec{M}\} \\
M \in \mathit{Methods} & ::= & m(\vec{x}{:}\vec{T})\{t\} : T \\
t \in \mathit{ThreadSeq} & ::= & \mathtt{stop} \mid \mathtt{error} \mid v \mid \mathtt{let}\ x{:}T = e\ \mathtt{in}\ t \\
e \in \mathit{Exp} & ::= & t \mid \mathtt{if}\ v\ \mathtt{then}\ \ e\ \mathtt{else}\ e \mid v.f \mid v.f := v \mid v.m(\vec{v}) \\
 & \mid & \mathtt{new}\ \ C(\vec{v}) \mid \mathtt{spawn}\ t \mid \mathtt{new}\ L \mid v.\ \mathtt{lock} \\
 & \mid & v.\ \mathtt{unlock} \mid \mathtt{if}\ v.\ \mathtt{trylock}\ \mathtt{then}\ e\ \mathtt{else}\ e \\
v \in \mathit{Value} & ::= & r \mid x \mid () \\
x, y \in \mathit{Var} & & \\
S, T \in \mathit{Type} & ::= & C \mid B \mid \mathtt{Unit} \mid \ \mathtt{L}
\end{array}
$$

Semantics (locks): Global level

$$\sigma \in Heap \quad ::= \quad \bullet \mid \sigma, o : C(\vec{v}) \mid \sigma, l : 0 \mid \sigma, l : p(n)$$

Global configuration: $\sigma \vdash P$, so global step:

$$\sigma \vdash P \rightarrow \sigma' \vdash P' \; . \qquad (1)$$

$$\text{where } P \quad ::= \quad \mathbf{0} \mid P \parallel P \mid p\langle t \rangle$$

$$\frac{\sigma(l) = p'(n) \qquad p \neq p'}{\sigma \vdash p\langle \texttt{let } x : T = l.\,\texttt{unlock in } t \rangle \rightarrow \sigma \vdash p\langle \texttt{error} \rangle} \text{ R-ERROR}_1$$

$$\frac{\sigma(l) = 0}{\sigma \vdash p\langle \texttt{let } x : T = l.\,\texttt{unlock in } t \rangle \rightarrow \sigma \vdash p\langle \texttt{error} \rangle} \text{ R-ERROR}_2$$

Type and effect system

At the local level, the judgment of the expression e

$$\sigma; \Gamma; \Delta_1 \vdash e : T :: \Delta_2 \qquad (2)$$

- Under the environment Γ the expression e has the type T
- Executing e leads to the effect changing from Δ_1 to Δ_2

$$\frac{\sigma; \Gamma \vdash v : L \qquad \Delta \vdash v}{\sigma; \Gamma; \Delta \vdash v.\ \texttt{lock}\colon L :: \Delta + v}\ \textsc{T-Lock} \qquad \frac{\sigma; \Gamma \vdash v : L \qquad \Delta \vdash v : n+1}{\sigma; \Gamma; \Delta \vdash v.\ \texttt{unlock}\colon L :: \Delta - v}\ \textsc{T-Unlock}$$

$$\begin{array}{rcl}
\sigma \in Heap & ::= & \bullet \mid \sigma, o : C(\vec{v}) \mid \sigma, l : 0 \mid \sigma, l : p(n) \\
\Gamma \in TypeEnv & ::= & \bullet \mid \Gamma, x : T \\
\Delta \in LockEnv & ::= & \bullet \mid \Delta, l : n \mid \Delta, x : n
\end{array}$$

Definition (Projection)

Assume a heap σ with $\vdash \sigma : ok$ and a thread p. The *projection* of σ onto p ($\sigma \downarrow_p$) inductively defined:

$$\begin{array}{rcll}
\bullet \downarrow_p & = & \bullet & \\
(\sigma, l{:}0) \downarrow_p & = & \sigma \downarrow_p, l{:}0 & \\
(\sigma, l{:}p(n)) \downarrow_p & = & \sigma \downarrow_p, l{:}n & \\
(\sigma, l{:}p'(n)) \downarrow_p & = & \sigma \downarrow_p, l{:}0 & \text{if } p \neq p' \\
(\sigma, o{:}C(\vec{v})) \downarrow_p & = & \sigma \downarrow_p & .
\end{array}$$

Type rules (T-Call)

At the global level, the judgment of the form: $\sigma \vdash P : ok$

$$\frac{\sigma;\Gamma \vdash \vec{v} : \vec{T} \qquad \sigma;\Gamma \vdash v : C \qquad \vdash C.m : \vec{T} \to T :: \Delta'_1 \to \Delta'_2 \\ \vdash C.m = \lambda\vec{x}.t \qquad \Delta_1 \geq \Delta'_1[\vec{v}/\vec{x}] \qquad \Delta_2 = \Delta_1 + (\Delta'_2 - \Delta'_1)[\vec{v}/\vec{x}]}{\sigma;\Gamma;\Delta_1 \vdash v.m(\vec{v}) : T :: \Delta_2} \text{ T-CALL}$$

Definition (Operators on lock environments)

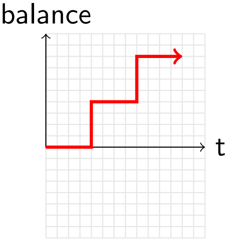
1. Let $\Delta = \Delta_1 + \Delta_2$, then
   - $\Delta \vdash l : n_1 + n_2$ if $\Delta_1 \vdash l : n_1 \wedge \Delta_2 \vdash l : n_2$.
   - $\Delta \vdash l : n_1$ if $\Delta_1 \vdash l : n_1 \wedge \Delta_2 \not\vdash l$ (and symmetrically).
2. $\Delta_1 \geq \Delta_2$ if $dom(\Delta_1) \supseteq dom(\Delta_2) \wedge \forall l \in dom(\Delta_2)$: $n_1 \geq n_2$, where $(\Delta_1 \vdash l : n_1) \wedge (\Delta_2 \vdash l : n_2)$.
3. $\Delta_1 - \Delta_2$ for $\Delta_1 \geq \Delta_2$, analogously.

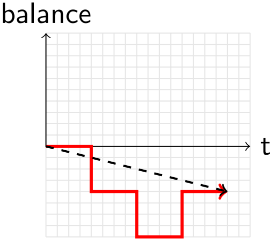
An illustration of T-Call

Two methods m and n operating on a single lock:

$m()\{l.lock;\ l.lock\ x.n(); \ldots\}$ where

$n()\{l.unlock;\ l.unlock;\ l.lock\}$



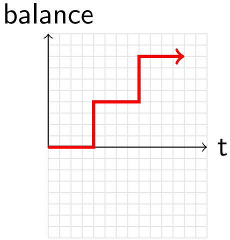


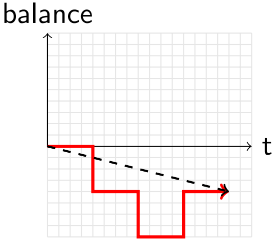
An illustration of T-Call

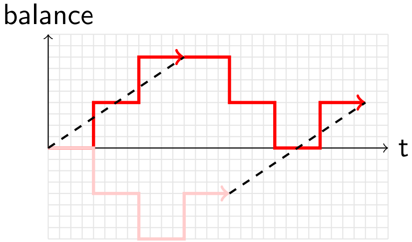
Two methods *m* and *n* operating on a single lock:

$m()\{l.lock;\ l.lock\ x.n();\ldots\}$ where

$n()\{l.unlock;\ l.unlock;\ l.lock\}$







Examples of Aliasing

Method with 2 formal parameters

```
m(x1 : L,  x2 : L) {
  x1.unlock; x2.unlock
}
```

$$\Delta_1 = x_1{:}1, x_2{:}1 \tag{2}$$

$$o.m(l_1, l_2) \;:\; \Delta_1' = \Delta_1[l_1/x_1][l_2/x_2] = l_1{:}1, l_2{:}1 \tag{3}$$

$$o.m(l, l) \;:\; \Delta_1' = \Delta_1[l/x_1][l/x_2] = l{:}(1+1) \tag{4}$$

Definition (Substitution for lock environments:$\Delta[l/x]$)

Given $\Delta = v_1{:}n_1, \ldots, v_k{:}n_k,\ \Delta' = \Delta[l/x]$.

1. $\Delta' = \Delta'', l{:}(n_l + n_x)$ If $\Delta = \Delta'', l{:}n_l, x{:}n_x$
2. $\Delta' = \Delta'', l{:}n_x$ If $\Delta = \Delta'', x{:}n_x \wedge l \notin dom(\Delta'')$
3. $\Delta' = \Delta$, otherwise.

Listing 1: Method call, no aliasing

```
f1 := new L;
f2 := new L;          // f1 and f2: no aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

Nothing is wrong here!

Listing 2: Method call, aliasing

```
f1 := new L;
f2 := f1;     // f1 and f2: aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

Again, there is *no* run-time error!

Listing 3: Method call, no aliasing

```
f1 := new L;
f2 := new L;          // f1 and f2: no aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

Nothing is wrong here!

Listing 4: Method call, aliasing

```
f1 := new L;
f2 := f1;      // f1 and f2: aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

Again, there is *no* run-time error!

Listing 5: Method call, no aliasing

```
f1 := new L;
f2 := new L;          // f1 and f2: no aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

Nothing is wrong here!

Listing 6: Method call, aliasing

```
f1 := new L;
f2 := f1;     // f1 and f2: aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

Again, there is *no* run-time error!

Examples of Aliasing

Listing 7: Method call, no aliasing

```
f1 := new L;
f2 := new L;          // f1 and f2: no aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

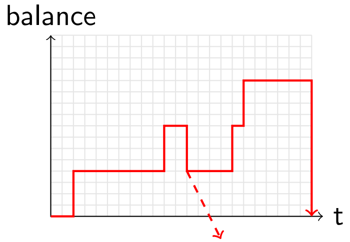
Nothing is wrong here!

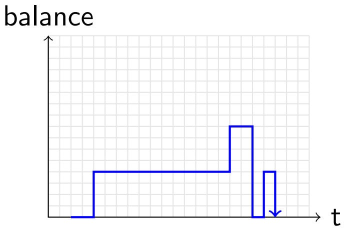
Listing 8: Method call, aliasing

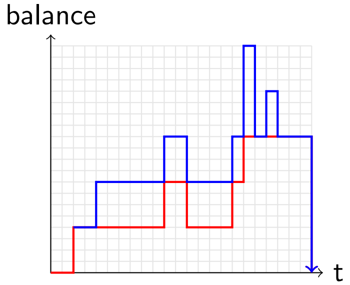
```
f1 := new L;
f2 := f1;     // f1 and f2: aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

Again, there is *no* run-time error!

Illustration of aliasing and non-aliasing locks







Examples of Aliasing

Listing 9: Method call, midway assignment

```
f1 := new L;
f2 := new L;   /
f1.lock; f2.lock;
f2 := f1;      // f1 and f2 are now aliases
o.m(f1,f2);
```

Obviously, the code leads to a *lock-error*!

$$\frac{\sigma;\Gamma \vdash v_1 : C \qquad \vdash C.f_i : T_i \qquad \sigma;\Gamma \vdash v_2 : T_i \qquad T_i \neq L}{\sigma;\Gamma;\Delta \vdash v_1.f_i := v_2 : T_i :: \Delta} \text{ T-ASSIGN}$$

Examples of Aliasing

Listing 10: Method call, midway assignment

```
f1 := new L;
f2 := new L;   /
f1.lock; f2.lock;
f2 := f1;      // f1 and f2 are now aliases
o.m(f1,f2);
```

Obviously, the code leads to a *lock-error*!

$$\frac{\sigma;\Gamma \vdash v_1 : C \qquad \vdash C.f_i : T_i \qquad \sigma;\Gamma \vdash v_2 : T_i \qquad T_i \neq L}{\sigma;\Gamma;\Delta \vdash v_1.f_i := v_2 : T_i :: \Delta} \text{ T-ASSIGN}$$

Examples of Aliasing

Listing 11: Method call, midway assignment

```
f1 := new L;
f2 := new L;   /
f1.lock; f2.lock;
f2 := f1;      // f1 and f2 are now aliases
o.m(f1,f2);
```

Obviously, the code leads to a *lock-error*!

$$\frac{\sigma;\Gamma \vdash v_1 : C \qquad \vdash C.f_i : T_i \qquad \sigma;\Gamma \vdash v_2 : T_i \qquad T_i \neq L}{\sigma;\Gamma;\Delta \vdash v_1.f_i := v_2 : T_i :: \Delta} \ \textsc{T-Assign}$$

Two core observations

- Aliasing does not hurt in our setting.
- locking assures interference-freedom

Type rules

$$\frac{\Delta_1 = \sigma \downarrow_p \qquad \sigma; \bullet; \Delta_1 \vdash t : T :: \Delta_2 \qquad t \neq \texttt{error} \qquad \Delta_2 \vdash \mathit{free}}{\sigma \vdash p\langle t \rangle : ok} \ \textsc{T-Thread}$$

$$\frac{\sigma \vdash P_1 : ok \qquad \sigma \vdash P_2 : ok}{\sigma \vdash P_1 \parallel P_2 : ok} \ \textsc{T-Par}$$

Soundness: proof by subject reduction

Definition (Hanging lock)

A configuration $\sigma \vdash P$ has a *hanging lock* if $P = P' \parallel p\langle \texttt{stop} \rangle$ where $\sigma(l) = p(n)$ with $n \geq 1$.

Theorem (Well-typed programs have no hanging locks)

Given an initial configuration $\sigma_0 \vdash P_0 : ok$. Then it's not the case that $\sigma_0 \vdash P_0 \longrightarrow^* \sigma' \vdash P'$, where $\sigma' \vdash P'$ has a hanging lock.

Theorem (Well-typed programs are lock-error free)

Given an initial configuration $\sigma_0 \vdash P_0 : ok$. Then it's not the case that $\sigma_0 \vdash P_0 \longrightarrow^* \sigma' \vdash P \parallel p\langle \texttt{error} \rangle$.

Related work

- [Igarashi and Kobayashi, 2005]: the resource usage analysis problem. The language, however, is sequential.
- [Gerakios et al., 2010]: a uniform treatment of region-based management and locks
- [Bigliardi and Laneve, 2000][Laneve, 2003]: a type system for statically assuring proper lock handling for the JVM, *structured locking*.
- [Iwama and Kobayashi, 2002]: a type system for multi-threaded Java program for JVM in non-lexical locking.

Summary, current and future work

Summary:

- A calculus supporting lock handling as in Java with operational semantics
- Usage of locks in non-lexical scope can be typed checked
  - Type and effect system
  - Soundness proof: subject reduction
- Aliasing, passing locks between threads, dynamic creation of objects, threads and especially locks.

Current and future work:

- Implementing a type checker.
- Adding exceptions (under work)

More details on the technical report
[Johnsen et al., 2010](http://heim.ifi.uio.no/ msteffen)

[Bigliardi and Laneve, 2000] Bigliardi, G. and Laneve, C. (2000).
A type system for JVM threads.
In *In Proceedings of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000*, page 2003.

[Gerakios et al., 2010] Gerakios, P., Papaspyrou, N., and Sagonas, K. (2010).
A concurrent language with a uniform treatment of regions and locks.
In *Programming Language Approaches to Concurrency and Communication-eCentric Software EPTCS 17*, pages 79–93.

[Igarashi and Kobayashi, 2005] Igarashi, A. and Kobayashi, N. (2005).
Resource usage analysis.
ACM Transactions on Programming Languages and Systems, 27(2):264–313.

[Iwama and Kobayashi, 2002] Iwama, F. and Kobayashi, N. (2002).
A new type system for JVM lock primitives.
In *ASIA-PEPM '02: Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, pages 71–82, New York, NY, USA. ACM.

[Johnsen et al., 2010] Johnsen, E. B., Mai Thuong Tran, T., Owe, O., and Steffen, M. (2010).
Safe locking for multi-threaded Java.
Technical Report 402, University of Oslo, Dept. of Computer Science. `www.ifi.uio.no/~msteffen/publications.html#techreports`. A shorter version (extended abstract) has been presented at the NWPT'10.

[Laneve, 2003] Laneve, C. (2003).
A type system for JVM threads.

Theoretical Computer Science, 290(1):741 – 778.