Safe Locking for Multi-threaded Java

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

University of Oslo, Norway

FSEN'11

20-22 April, 2011

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.

Lock Handling in Java: not release the lock after finishing

```
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);
      ................................
      l.lock();                          // 2 times lock
      } finally { l.unlock(); } // 1 time 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);
            ..........................
            l.unlock();
      } finally {
                    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

Statically avoid

- hanging locks
- lock exceptions

Solution

- Semantics for lock handling as in Java.
- Static type & effect system for safe usage of re-entrant locks.
- Soundness of our system: subject reduction.

Challenges

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

A Concurrent Calculus

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

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

$$\sigma \vdash P \rightarrow \sigma' \vdash P' \ . \tag{1}$$

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

$$\begin{array}{rcll} \sigma \in \mathit{Heap} & ::= & \bullet & \text{empty heap} \\ & \mid & \sigma, o \mapsto C(\vec{v}) & \text{object with instance state } C(\vec{v}) \\ & \mid & \sigma, l \mapsto 0 & \text{free lock} \\ & \mid & \sigma, l \mapsto p(n) & \text{lock taken } n \text{ times by } p \end{array}$$

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

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

Type and effect system

The judgment of the expression e

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

$$\begin{array}{rcl} \Gamma \in \mathit{TypeEnv} & ::= & \bullet \mid \Gamma, x : T \\ \Delta \in \mathit{LockEnv} & ::= & \bullet \mid \Delta, l : n \mid \Delta, x : n \end{array}$$

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

$$\frac{\sigma; \Gamma \vdash v : L \qquad \Delta \vdash v}{\sigma; \Gamma; \Delta \vdash v.\,\texttt{lock}: L :: \Delta + v}\ \text{T-LOCK} \qquad \frac{\sigma; \Gamma \vdash v : L \qquad \Delta \vdash v : n+1}{\sigma; \Gamma; \Delta \vdash v.\,\texttt{unlock}: L :: \Delta - v}\ \text{T-UNLOCK}$$

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

Definition (Operators on lock environments)

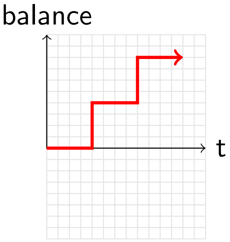
1. Let $\Delta = \Delta_1 + \Delta_2$, then
   - $\Delta \vdash l : n_1 + n_2$ if $\Delta_1 \vdash l : n_1 \wedge \Delta_2 \vdash l : n_2$.
   - $\Delta \vdash l : n_1$ if $\Delta_1 \vdash l : n_1 \wedge \Delta_2 \not\vdash l$ (and symmetrically).
2. $\Delta_1 \geq \Delta_2$ if $dom(\Delta_1) \supseteq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

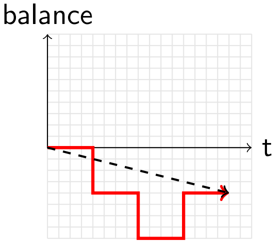
An illustration of T-Call

Two methods m and n operating on a single lock:

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

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



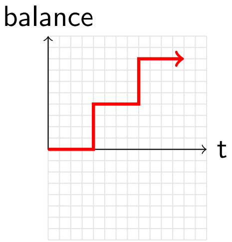


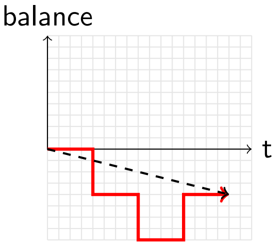
An illustration of T-Call

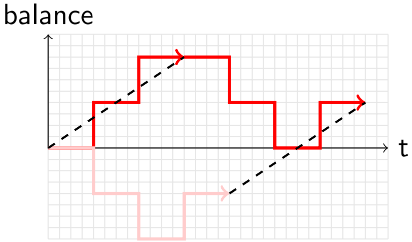
Two methods m and n operating on a single lock:

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

$$\Delta_1' = x_1{:}1, x_2{:}1 \tag{3}$$

$$o.m(l_1, l_2) \ : \ \Delta_1 = \Delta_1'[l_1/x_1][l_2/x_2] = l_1{:}1, l_2{:}1 \tag{4}$$

$$o.m(l, l) \ : \ \Delta_1 = \Delta_1'[l/x_1][l/x_2] = l{:}(1+1) \tag{5}$$

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

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

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

Examples of Aliasing

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!

Examples of Aliasing

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);
```

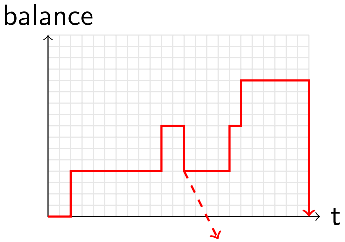
Nothing is wrong here!

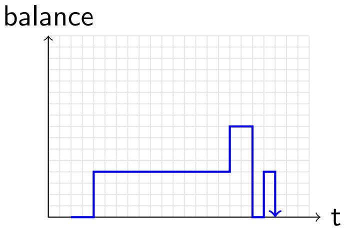
Listing 8: Method call, aliasing

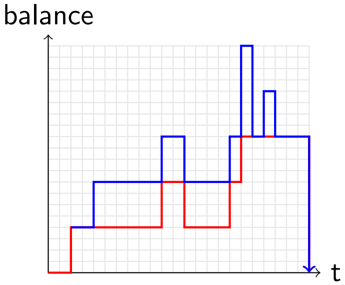
```
f1 := new L;
f2 := f1;     // f1 and f2: aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

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

Illustration of aliasing and non-aliasing locks







Definition (Hanging lock)

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* $\sigma_0 \vdash P_0 \longrightarrow^* \sigma' \vdash P \parallel p\langle \texttt{error} \rangle$.

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:

- Exception handling
- Higher order functions
- Type inference
- Implementation
- Case studies

[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.

[Mai Thuong Tran et al., 2010] Mai Thuong Tran, T., Owe, O., and Steffen, M. (2010).
Safe typing for transactional vs. lock-based concurrency in multi-threaded Java.
In Pham, S. B., Hoang, T.-H., McKay, B., and Hirota, K., editors, *Proceedings of the Second International Conference on Knowledge and Systems Engineering, KSE 2010*, pages 188–193. IEEE Computer Society.

[Mai Thuong Tran and Steffen, 2010] Mai Thuong Tran, T. and Steffen, M. (2010).
Safe commits for Transactional Featherweight Java.
In Méry, D. and Merz, S., editors, *Proceedings of the 8th International Conference on Integrated Formal Methods (iFM 2010)*, volume 6396 of *Lecture Notes in Computer Science*, pages 290–304. Springer Verlag.
An earlier and longer version has appeared as UiO, Dept. of Informatics Technical Report 392, Oct. 2009.

[Terauchi, 2008] Terauchi, T. (2008).
Checking race freedom via linear programming.

In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–10. ACM.