Safe Locking for Multi-Threaded Java*

September 27, 2010

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

Institute of Informatics, University of Oslo, Norway

1 Motivation

Many concurrency models have been developed for high-level programming languages such as Java. A trend here is more flexible concurrency control protocols, going beyond the original Java multi-threading treatment based on lexically-scoped locking. A recent proposal for Java supports flexible, *non-lexical* lock-handling via the `Lock`-classes in Java 5. The added flexibility of non-lexical use of the corresponding concurrency operators comes at a price: improper usage leads to run-time exceptions and unwanted behavior. This is in contrast with the more disciplined use under a lexically scoped regime, where each entrance to a critical region is syntactically accompanied by a corresponding exit as with traditional `synchronized` methods.

2 A Type and Effect System for Safe Locking

To assure safe use of locking we present a static type and effect system which assures that e.g., no lock is released more often than it is being held, or released by a thread which does not hold it. We call such erroneous situations *lock errors*. To prevent them, we basically keep track per thread of the number of locking and unlocking on the individual locks. As straightforwards as it sounds, there are three main challenges

Dynamic lock creation: locks can be dynamically created as instances of a lock class.

Aliasing: As locks are accessed via references to a lock instance, they are subject to aliasing: two different variables may refer to the same lock.

Lock passing: Lock references can be passed via method calls (between objects) and via instance fields (between threads).

The type and effect system resembles partly an earlier one for safe transaction handling [6], but the mentioned challenges such as aliasing and passing of identities arise from the crucial difference between transactions and locks: the latter have an identity at the level of the programming language, whereas transactions have not (see also [5] or [1] for a comparison of the differences between the two concurrency control mechanisms as far as static typing is concerned).

* The authors are listed in alphabetical order. The work has been partly supported by the EU-project FP7-231620 HATS.

To capture effects related to locks, the general form of judgments for a single expression, i.e., inside one thread, is of the form

$$\sigma; \Gamma; \Delta_1 \vdash e : T :: \Delta_2 \ . \tag{1}$$

It is read as "given the heap σ and under the lock assumptions Δ_1 and type assumptions Γ, expression e has type T and some effect which changes Δ_1 into Δ_2". The *lock environment* keeps the assumptions for locks, i.e. whether a lock is free (denoted by 0), or taken by some thread in which the environment need to remember how many times it is taken, to capture re-entrance.

The following rules sketch 4 typical effect rules for expressions, concentrating on the aspects of lock handling and multi-threading:

$$\frac{\sigma; \Gamma; \bullet \vdash e : T :: \Delta' \qquad \Delta' \vdash \mathit{free}}{\sigma; \Gamma \vdash \mathsf{spawn}\ e : \mathtt{Unit}}\ \text{T-Spawn} \qquad \frac{\Delta \not\vdash l}{\sigma; \Gamma; \Delta \vdash \mathsf{new}\ L : L :: \Delta, (l{:}0)}\ \text{T-NewL}$$

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

The type and effect system is not only concerned with checking expressions, the declarations of methods are generalized, as well. We do not require that method bodies are balanced wrt. lock usage: a method may just take one lock without releasing it, or vice versa. To ensure, however, that this flexibility does not lead to lock errors, the declaration of a method does not only contains the expected balances for all lock parameters but also a requirement on where that method can be used as a form of precondition for the actual parameters. So the *specification* of a method, as far as its effects are concerned, is of the form $m(\vec{x} : \vec{T})\{e\} : \Delta_1 \to \Delta_2$, and the corresponding rule looks as follows:

$$\frac{\vdash C.m : \vec{T} \to T :: \Delta_1 \to \Delta_2 \qquad \bullet; \vec{x}{:}\vec{T}, \mathsf{this}{:}C; \Delta_1 \vdash e : T :: \Delta_2}{\vdash C.m(\vec{x}{:}\vec{T})\{e\} : ok}\ \text{T-Meth}$$

The formal system must make sure that for all client code of the form $o.m(\vec{v})$, where m is the mentioned method, the call is issued only at a location, where the balance is *at least* Δ_1:

$$\frac{\begin{array}{ccccc}\sigma; \Gamma \vdash o : C & \vdash C.m : \vec{T} \to T :: \Delta_1' \to \Delta_2' & \sigma; \Gamma \vdash \vec{v} : \vec{T} \\ \hat{\Delta}_1 = \Delta_1 \downarrow_{\vec{v}} & \vec{v}' = \mathit{locks}_\sigma(\vec{v}) & \vec{x} = \mathit{dom}(\Delta_1') & \hat{\Delta}_1 \geq \Delta_1'[\vec{v}'/\vec{x}] \\ & \Delta_2 = \Delta_1 + (\Delta_2' - \Delta_1')[\vec{v}'/\vec{x}] & \end{array}}{\sigma; \Gamma; \Delta_1 \vdash o.m(\vec{v}) : T :: \Delta_2}\ \text{T-Call}$$

The mentioned pre-condition for method calls corresponds to the premise $\hat{\Delta}_1 \geq \Delta_1'[\vec{v}'/\vec{x}]$ in the premise of T-Meth. Further premises of that rule handled well-typedness, but especially make sure that the lock environment Δ_2 *after* the

call is calculated from Δ_1 before the call by taking the declared net effect of the method $\Delta'_2 - \Delta'_1$ into account.

Since the locks assure mutual exclusion, a lock can be taken only by one thread at a time, which means the static analysis does not need to take into account interference between concurrent threads when analysing these balances.

3 Results

Our contributions are:

Semantics: We present an operational semantics for a multi-threaded variant of Featherweight Java [2] supporting non-lexical use of locks.

Type and effect system: For that calculus, we present a type and effect system in terms of a formal derivation system following the ideas sketched above, to avoid improper use of lock operations.

Soundness : Based on the operational semantics, we prove the soundness of our formal system by standard subject reduction.

The type and effect system also gives insight about "good" usage of locks in practice which says that users should pass locks via method calls rather than via instance fields.

References

1. C. Blundell, E. C. Lewis, and M. K. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.
2. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '99*, pages 132–146. ACM, 1999. In *SIGPLAN Notices*.
3. E. B. Johnsen, T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe locking for multi-threaded Java. In *Submitted for conference publication*, 2010.
4. E. B. Johnsen, T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe locking for multi-threaded Java. Technical report, University of Oslo, Dept. of Computer Science, Oct. 2010. to appear.
5. T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe typing for transactional vs. lock-based concurrency in multi-threaded Java. In *Proceedings of the Second International Conference on Knowledge and Systems Engineering, KSE 2010*. IEEE Computer Society, Oct. 2010. Accepted for publication.
6. T. Mai Thuong Tran and M. Steffen. Safe commits for Transactional Featherweight Java. In D. Méry and S. Merz, editors, *Proceedings of the 8th International Conference on Integrated Formal Methods (iFM 2010)*, volume 6396 of *Lecture Notes in Computer Science*, pages 290–304 (15 pages). Springer-Verlag, Oct. 2010. An earlier and longer version has appeared as UiO, Dept. of Comp. Science Technical Report 392, Oct. 2009 and appeared as extended abstract in the Proceedings of NWPT'09.
7. M. Steffen and T. M. T. Tran. Safe commits for Transactional Featherweight Java. Technical Report 392, University of Oslo, Dept. of Computer Science, Oct. 2009. 23 pages. A shorter version (extended abstract) has been presented at the NWPT'09.